°Master of Science

Image Processing Pipeline and 3D Interactive Visualization Methods for Teravoxel Volumes

by

Akanksha Ashwini

A dissertation submitted in partial satisfaction of the
requirements for the degree of

in

Computer Engineering

in the

Graduate Division

of the

Kettering University, Flint

Committee in charge:

Professor Jaerock Kwon, Chair
Professor Guiseppi Turini
Associate Professor Girma Tewolde

Fall 2017

The dissertation of Akanksha Ashwini, titled Image Processing Pipeline and 3D Interactive Visualization Methods for Teravoxel Volumes, is approved:

Chair		Date	
		Date	
		Date	

Kettering University, Flint

Image Processing Pipeline and 3D Interactive Visualization Methods for Teravoxel Volumes

Abstract

Image Processing Pipeline and 3D Interactive Visualization Methods for Teravoxel Volumes

by

Akanksha Ashwini

in Computer Engineering

Kettering University, Flint

Professor Jaerock Kwon, Chair

Invasive brag; forbearance.

To Ossie Bernosky

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank Prof. Jaerock Kwon......

Part I

Introduction

Chapter 1

Introduction

Today researchers have implemented numerous ways to study the mammalian brain. It is believed that many of the human brain illness and diseases can be cured by understanding the abnormalities in the brain morphology. Accurate microvascular morphometric information from the brain has significant implications in fields including the quantification of angiogenesis in cancer research, the study of immune response for neural prosthetics, and predicting the nature of blood flow as it relates to stroke.

For decades, scientists have routinely used rats as a primary model for brain research. Rodents can be a good model for humans because a lot of the structure and connectivity that exists in human brains also exists in rodents. It is said that Rodents are genetically similar to humans, they have shorter lifespans, enabling scientists to study their brain structure across generations if desired. Therefore, a lot of research performed today for the study of the human brain uses a mouse brain for experiments.

Connectomics aims to map the full connection matrix of the brain. A vast variety of methods and approaches have been developed for vascular extraction, analysis, and modeling with increasing complexity[14]. Knife-Edge Scanning Microscopy (KESM) being the first instrument of this technique, allows imaging of the whole mouse brain microvascular system at resolutions sufficient to perform accurate morphometry. With such high-throughput and high-resolution imaging technologies, it is possible to acquire teravoxel sized three-dimensional neuronal and microvascular images of the whole mouse brain with sub-micrometer resolution. Besides having such efficient image acquisition system available. It is also imperative to be able to visualize and share these teravoxel volumes efficiently, to facilitate group efforts from research communities. However, due to the immense size of the data sets, effectively visualizing, sharing and managing them have always been a big challenge.

This thesis describes an Image Processing Pipeline for handling such huge teravoxel volumes, a web-based real-time 3D visualization framework that allows research groups to work in collaboration, and a virtual reality framework for fully-interactive scientific visualization of the mouse brain data. The proposed work can visualize and share terabyte-sized three-dimensional images for study and analysis of mammalian brain morphology. Although the image processing pipeline used a KESM data set to show the feasibility of it, the proposed system can also be used for other larger biomedical data sets. The virtual reality application provides researchers a novel way to be able to step into the 3D visualization framework to explore and fully interact with the mouse brain data.

1.1 Background

Knife-Edge Scanning Microscopy

KESM is one of the first instruments to achieve whole-mouse-brain-scale imaging at sub-micrometer resolution [19][12][17]. KESM is the technique of concurrently slicing and imaging tissue samples at sub-micrometer resolution with a high resolution and high sensitivity digital line camera.

Three-dimensional light microscopy in medical imaging needs continuous automated sectioning method to automate the process. This can be achieved by either optical sectioning or physical sectioning using a suitable sample and movement stage. However, to obtain high-resolution volumetric tissue structure data at high throughput, it is preferred to use physical sectioning. Optical sectioning is disadvantageous because of the depth resolution limitation and the trade-off between signal quality and disruptive background noise such as tissue data from out-of-focus imaging planes.

Since the slicing and imaging happens simultaneously in KESM, the overall throughput of the system is high. Each slice is an aggregation of multiple line images. It also preserves image registration throughout the depth of the tissue block and eliminates undesirable events such as back-scattering of light and bleaching of fluorescent-stained tissue below the knife.

The tissue is stained and embedded in either LR-White or Araldite to make it stiff, as it is important to have the tissue rigid and wrinkle-free during cutting to achieve sub-micrometer thickness. The tissue sample being sliced by the knife is imaged just above the knife edge by a powerful line scan camera (See Fig.**??**). The image capture mechanism is triggered based on the encoded position of the tissue being sliced, which ensures that every sectioning results in an image capture. The KESM can image a $1cm^3$ tissue block in approximately 50 hours at a resolution of $0.6\mu\text{m}\times0.7\mu\text{m}\times1.0\mu\text{m}$. While sectioning and imaging, the width of the tissue slice is not exactly as the field of view of the objective. Thus, there are additional non-tissue areas that appear as dark regions on either side of the tissue in every image. The additional region causes a significant increase of memory required to store the images and process them. Each tissue sample imaged by the KESM can generate up to around 80,000 images (the tissue is laterally sectioned several times, and each column has around 10,000 images). Thus, to extract tissue region manually requires a lot of time and effort and is inefficient. With an aim to automate this process, a template-matching based method was proposed for tissue extraction from the KESM image stacks [5] and was later improvised [24]. After the tissue region extraction, intensity levels are normalized, and the clean images are stored in the column stacks.

Internet Enabled Robotic Microscopy

The Internet Enabled Robotic Microscope (IEROM), the second generation of microscope based on KESM, is a low-cost and more robust version of the first prototype. It aims to overcome the limitations of the first generation prototype by making the instrument less expensive, flexible, less bulky and occupy a smaller footprint. The cost is reduced by changes such as using LED illumination instead of laser illumination. The optics train design is modular and flexible. This provides an easily operable, flexible platform for biomedical researchers across different domains such as neuroscience and vascular research. The IEROM promises to be commercially viable and indispensable to biomedical researchers.

1.2 Related Work

The Mouse Atlas Project (MAP)

Mackenzie Graham developed a probabilistic atlas of the adult and developing C57BL/6J mouse. The MAP consists of not only data from Magnetic Resonance Microscopy (MRM) and histological atlases, but also a suite of tools for image processing, volume registration, volume browsing, and annotation. The MAP will produce an imaging framework to house and correlate gene expression with anatomic and molecular information drawn from traditional and novel imaging technologies. This digital atlas of the C57BL/6J mouse brain is composed of volumes of data acquired from μMRI, block-face imaging, histology, and immunohistochemistry. MAP technology provides the infrastructure for the development of the Allen Brain Atlas [16]. Also, see the related Mouse BIRN (Biomedical Informatics Research Network).

Allen Brain Atlas

The Allen Brain Atlas contains detailed gene expression maps for $\approx 20,000$ genes in the C57BL/6J mouse [13]. A semi-automated procedure was used to conduct in situ hybridization and data acquisition on 25μm thick sections (z-axis) of the mouse brain. The x-y axis resolution of the images ranges from 0.95μm to 8μm. The Allen Brain Atlas is the first comprehensive gene expression map at the whole-brain level and is currently accessed over 4 million times per month, with over 250 scientists browsing the data on a daily basis.

The Mouse Brain Library (MBL)

MBL is developing methods to construct atlases from celloidin-embedded tissue to guide registration of MBL data into a standard coordinate system, by segmenting each brain in its collection into 1,200 standard anatomical structures at a resolution of 36μm [26]. Algorithms are to be designed to segment each brain in the MBL into a set of standard anatomical structures like those defined in the rat atlas produced by Computer Vision Laboratory for

Vertebrate Brain Mapping at Drexel College of Medicine, whose computerized 3D atlas was built from stained sections for the mouse brain that reconstructs Nissl-stained sectional material, a 17.9 μm isotropic 3D data set, from a freshly frozen brain of an adult male C57BL/6J mouse.

BrainMaps.org

Brain Maps.org is an internet-enabled, high-resolution brain map [20]. The map contains over 10 million megapixels (35terabytes) of scanned data, at a typical resolution of $\approx 0.46\mu$m/pixel (in the x-y plane). The atlas provides an intuitive Web-based interface for easy and bandwidth efficient navigation, through the use of a series of sub-sampled (zoomed out) views of the data sets, similar to the Google Maps interface. Even though the x-y plane resolution is below 1μm, the z-axis resolution is orders of magnitude lower (for example, one coronal brain set has 234 slides in it, corresponding to a sectional thickness of 25μm). The database also serves serial sections from electron microscopy, cryosections, and immunohistochemistry, and hosts a total of 135 data sets (as of March 2, 2011).

Whole-Brain Catalog (WBC)

WBC is a 3D virtual environment for exploring multiple sources of brain data (including mouse brain data), e.g., Cell Centered Database (CCDB), Neuroscience Information Framework (NIF), and the Allen Brain Atlas (see above). WBC has native support for registering to the Waxholm Space, a rodent standard atlas space [11]. It supports multiple functionalities including visualization, slicing, animations, and simulations. In summary, there are several mouse brain atlases available, with data from different imaging modalities, but their resolution is not high enough in one or more of the *x, y,* or *z* axes to show the morphological detail of neurons.

Knife-Edge Scanning Microscopy Brain Atlas (KESMBA)

KESMBA [6] framework has been designed and implemented to allow the widest dissemination of KESM mouse brain circuit data by overlaying transparent layers of images with distance attenuation. Overlaying image stacks containing two intertwining objects to get minimum intensity projection results in the loss of 3D information. Although, interleaving each image with semi-opaque blank images brings out the 3D information. Still, KESMBA provides a pseudo 3D visualization as it stacks the semitransparent image slices for not more than 30 layers at once.

Terafly - Vaa3D plugin

Terafly is a Vaa3D plugin [2] for real-time 3D visualization of terabyte sized volumetric images. Vaa3D, which is an open-source, cross-platform system, extended its powerful 3D visualization and analysis capabilities to images of potentially unlimited size with this plugin. When used with large volumetric images up to 2.5 Terabyte in size, Vaa3D-TeraFly exhibited real-time (sub-second) performance that consistently scaled on image size. TeraFly can generate a 3D region of interest (ROI) by subsequent fetching and rendering of image data at higher resolutions, thus enabling fast (sub-second) visualization of Terabyte-size images. It exhibits real-time performance regardless of image size when used on both high and medium-end computers. However, the performance is constrained on a local computer and cannot be directly used to share 3D visualization results of terabyte-sized data sets among research groups.

1.3 Overview and Thesis Structure

Modern high-throughput and high-resolution 3D bioimaging technologies such as Knife-Edge Scanning Microscopy (KESM) [17] have enabled imaging and reconstruction of the whole mouse brain architecture at sub-micrometer resolution. KESM performs simultaneous serial sectioning and imaging of the whole mouse brain and generates data sets that include: neuronal circuits (Golgi stained), soma distribution (Nissl stained), and vascular networks (India ink-stained). The data sets are multi-scaled images, ranging from sub-cellular ($< 1\mu$m) to the whole organ scale ($\approx$ 1cm). The KESM scans a $1cm^3$ tissue block in approximately 50 hours at a resolution of $0.6\mu\text{m}\times 0.7\mu\text{m}\times 1.0\mu\text{m}$. It then stores the scanned biological tissue data digitally in the form of stacked 2D images, the size for which is $\approx$ 2TB. Then, through a processing pipeline, these stacked 2D images are converted into volumes composed of terabytes of voxels, referred as *Teravoxel Volumes*. Due to immense size and multi-scale nature of the data set, efficiently visualizing and sharing them among research communities for analytical studies have always imposed challenges on researchers.

3D visualization helps users to understand the morphology of the biological organ. At the same time, efficiently sharing the data across research communities is also essential for review and feedback. Terafly [23][2] is an open-source Vaa3D [15] plug-in to support 3D visualization of immensely sized biomedical images. Although the tool is efficient in visualizing terabyte-scale images, the plug-in toolkit is a stand-alone software package that is required to be installed on local computers. Also teravoxel data must reside on the same computer or local network resources. This hinders the research communities to work in collaboration with data sets. On the other hand, there have been efforts to develop web-based applications to visualize neuronal circuits and microvascular data sets (e.g., The Mouse Atlas Project, Allen Brain Atlas [13] and Knife-Edge Scanning Microscopy Brain Atlas (KESMBA) [6]). These applications allow centralization of data sets and facilitate sharing of visualization results. Yet they are not efficient real-time 3D visualization methods. For instance, Allen Brain Atlas based on the Mouse Atlas Project does not support high-resolution data visualization (details will be in the next section). KESMBA provides pseudo-3D visualization through stacking of semitransparent image slices. The maximum number of layers is 30. It takes time for KESMBA load all the layers before it displays. Also, it is just layered and attenuated

2D image stacks. This creates a need for the development of technologies which facilitate efficient 3D visualization along with centralization of teravoxel volumes.

In this thesis, we discuss the implementation of an image processing pipeline for a Web-based real-time framework for 3D visualization of teravoxel volumes, such as the microvascular data set obtained by the KESM. This pipeline is capable of visualizing both terabyte volumes in real-time. Since the proposed framework is Web-based, visualization is entirely independent of the underlying operating system. Through the framework, 3D visualized data sets can be accessed easily so that it facilitates sharing of results across research communities. It even overthrows the necessity of downloading large data sets or installing any software. Out of the several other advantages of using Web technology, one of the most important features is the great level of interoperability achieved that results in faster switching and visualization of multi-resolution volumes. The web-based framework is currently implemented using a mouse brain vascular dataset from KESM. Also, since we intend to share and study KESM dataset only, we do not require any spatial database to manage them. However, the approach can be implemented for any other biomedical dataset of any size. The visualization of a volume of a region of interest (ROI) currently depends on the manual selection input from the user. But the process of switching between different resolution volumes through user's mouse scroll input can be automated later as a part of future work. The graphical user interface is designed to display a polygon mesh from a $256 \times 256 \times 256$ volume. The user can explore the dataset by selecting a region of interest to display at a particular resolution. Thus, different sections of the whole-mouse-brain at various resolutions can be visualized in real-time. This kind of visualization will be similar to one obtained in the Vaa3D [15] plugin: Terafly [23][2] or the Google Earth application.

Another major part of this thesis discusses the implementation of a Virtual Reality Framework for a fully-interactive and immersive 3D visualization experience with the mouse brain. Effective data visualization is the demand of the era of *big data* and immersive virtual reality provides benefits beyond the traditional desktop visualization tools. Understanding the complex microvascular network of the brain in such a platform leads to better perception of the morphology, more intuitive data understanding and a better retention of the perceived relationships in the data structure. With the VR framework designed for the KESM data, it is possible to navigate inside the mouse brain structure and load different resolution data set prepared by the image processing pipeline.

This thesis has been divided into three logical parts. The first part describes the Image Processing Pipeline implemented, the second part discusses the web-based 3D visualization framework design, and the third part introduces the virtual reality framework for the KESM data set. The mouse brain data-set used in this thesis are those imaged by the Brain Tissue Scanner in the Brain Networks Laboratory at Texas A&M University in 2008 and is from a C57/BL6 mouse specimen. It is a *India Ink* stained vascular network data set, labeled with the mouse brain id: `MOU1_BRA_IND_2008_04`.

Part II

Image Processing Pipeline : Principles and Methods

Chapter 2

Image Processing Pipeline for IEROM Dataset

2.1 Overview

The imaging techniques implemented in the KESM can reconstruct the mouse brain with microscopic resolution. However, the architecture of a Mammalian Brain cannot be understood properly in a single-cell resolution. So, we design the Image Processing Pipeline for IEROM to create multi-resolution unit volumes which can help visualize and study the structure of the biological organ efficiently.

During the image acquisition stage, the KESM scans a $1cm^3$ tissue block in approximately 50 hours at a resolution of 0.6μm$\times 0.7\mu$m$\times 1.0\mu$m. It then stores the scanned biological tissue data digitally in the form of stacked 2D images, the size for which is $\approx$ 2TB. However, the dataset obtained from the automated microscope cannot be directly used for visualization or feature extraction. The images contain background noise and additional artifacts specific to the KESM imaging method that needs to be alleviated before we can clearly visualize and process them. Therefore, after the acquisition of images, initial preprocessing steps are executed to remove any unwanted noise or artifacts from the images. [24].

After the preprocessing of images, we obtain 2D column stacks of clean images containing only tissue areas from the mouse brain. Through the image processing pipeline explained in this chapter, these stacked 2D images are then converted into volumes composed of terabytes of voxels, referred as *Teravoxel Volumes*.

2.2 Image Processing Pipeline Design

The complete implementation of the image processing pipeline is divided into four stages:

- Stitching of 2D images across column stacks to create a single 2D image stack of microscopic resolution images.

- Sub-sampling of the images in the stack (created after stitching), to generate multiple resolution image stacks.
- Creating unit-image stacks of 256 images, where a unit is defined as a resolution of 256×256. These unit-image stacks are pre-requisite for unit-volumes.
- Making 3D models of different formats from the unit-image stacks.

Image Stitcher Block

The images received after the acquisition and preprocessing steps are clean, without any artifacts and noise, with only tissue areas, but are in the form of column stacks. Each column store images for a specific section of the tissue. Since during KESM serial sectioning, the images scanned for a particular part of the tissue are stored in a single column stack. So each column stack is from a different starting point along the *y-axis.* Although, the width and height of the images in each column stack remains same i.e., the slice size 2400×12000.

To experiment this pipeline, we used the scanned data set created by the KESM in the year 2008. The scanned files are saved in the column stacks and named in such a manner that they provide information about the year and date of scanning, x-position, y-position, z-position, time, the speed of sectioning, and orientation. As a part of the preprocessing step, the images in the columns are filtered for the non-tissue areas and the files with no tissue areas are removed. This process reduced the number of column stacks eventually, from six to four.

The Image Stitcher block will stitch the images across the columns along the *z-axis.* The image files with the same *z-position* value are tiles together to form a larger image. So in our case, the sliced images of resolution 2400×12000 from each column when tiled together for a particular *z value,* creates an image of resolution 9600×12000.

Sub-Sampler Block

This block will create multi-resolution image stacks from the original microscopic resolution image stack. The image stack created by the Image Stitcher block has images of the highest possible resolution for the images acquired by the KESM, i.e., 9600×12000. The only way to create multiple resolution images without hampering the quality of images is to sub-sample them.

The sub-sampler block will create image stacks of resolution half of the input image stack. For example, if we sub-sample the original stitched images, we will end up in an image stack with images of resolution 4800×6000. The sub-sampler block will continuously sub-sample the image stack till the smallest resolution image stack is achieved. The sub-sampling was performed five times to create six different resolution image stacks.

Unit Volume Creator Block

The higher resolution volumes are difficult to visualize together, due to their immense size. Breaking such large volume data set into smaller volumes makes it easier and efficient for the user to visualize and study the section of mouse brain.

This block basically creates unit-image stacks where a unit is a predefined chosen size of 256. The idea is to create image stacks which can be used to directly create volumes of equal dimensions across all three axes, i.e., $256 \times 256 \times 256$. So the unit-image stacks have 256 image files, each of resolution 256×256. The unit-image stacks are extracted and cropped out from all the different resolution image stacks.

3D Model Maker Block

As the name suggests this block creates the 3D models out of the unit-volumes. The methods implemented so far is capable of creating 3D models and meshes in .tiff, .stl, and .vtk formats. However, the approach can be extended to create other visualization formats as well.

2.3 Tools and Software used to design the Image Processing Pipeline

The Insight Segmentation and Registration Toolkit (ITK)

ITK is an open-source software toolkit, implemented in C++, that provides algorithms for performing registration and segmentation to multidimensional data. Segmentation is the process of identifying and classifying data found in a digitally sampled representation. Registration is the task of aligning or developing correspondences between data. ITK is widely used for medical image processing. However, It does not include methods for displaying images, nor a development environment or an end user application for exploring the implemented algorithms.

The Visualization Toolkit (VTK)

VTK is an open-source, freely available software system for scientific visualization, information visualization, 3D computer graphics, modeling, image processing, and volume rendering. VTK is implemented as a C++ toolkit, requiring users to build applications by combining various objects into an application.

2.4 Controller-Worker Model

All the above methods were implemented using a *controller-worker* model. Each block explained above has its own *controller* and *worker*. A *worker* is the one which make changes;

execute functions utilizing ITK and VTK libraries. Whereas, a *controller* is the one which decides what files are to be passed to the *worker* and even controls the *worker* execution. This model has been applied to avoid time delays, generally caused due to looping over the same task. Here, each time a task is to be executed, the *controller* code simply calls the *worker* application which performs that task. Since all the *controllers* are written in Qt and do not use any ITK/VTK libraries, the complete design is a cross-platform solution.

Chapter 3

Stitching of Column Images

3.1 Overview

After the serial sectioning performed in KESM, the raw data set of 2D images is stored in the form of stacks and columns. The tissue area from each raw image is automatically cropped and saved, so each column contains images of only tissue area from the mouse brain. The images in each column are further processed; the noise is removed, and image intensity is normalized for each cropped image. Then the images at the same z coordinate are stitched across the columns in an image sheet. ITK filters are utilized to perform the stitching algorithm. Whereas, we implemented the image intensity normalization algorithm previously proposed for KESM image stacks[].The outcome of this process is a single stack of 2D images with clean prominent tissue areas. For this experiment, the raw images we had were initially divided into four columns or four stacks of images. The resolution of each image in each column stack was 2400×12000. After the stitching, we finally got a stack of 9626 images each of resolution 9600×12000 (See Fig.3.1).

3.2 Input for Image Stitcher

The IEROM performs serial sectioning of the whole mouse-brain tissue. The thin slices of the tissue are simultaneously cut and imaged. All the serially scanned images are then stored digitally in the form of stacks and columns. Each serially cut section is of the same width as that of the knife edge. The line scan camera scans and stores the images according to their x and y coordinate information. The same xy information is stored in the same column. This leads to the formation of stacks of images in each column. The images are stored across five column stacks, where each image is of resolution 2400×12000. Column directories are named as Vasculature_col0001, Vasculature_col0002, Vasculature_col0003, Vasculature_col0004. These images are cropped and normalized. The input for the Image Sticther block is those four column directories with jpeg files, and a file with all the metadata information generated by the IEROM during the sectioning and imaging process.

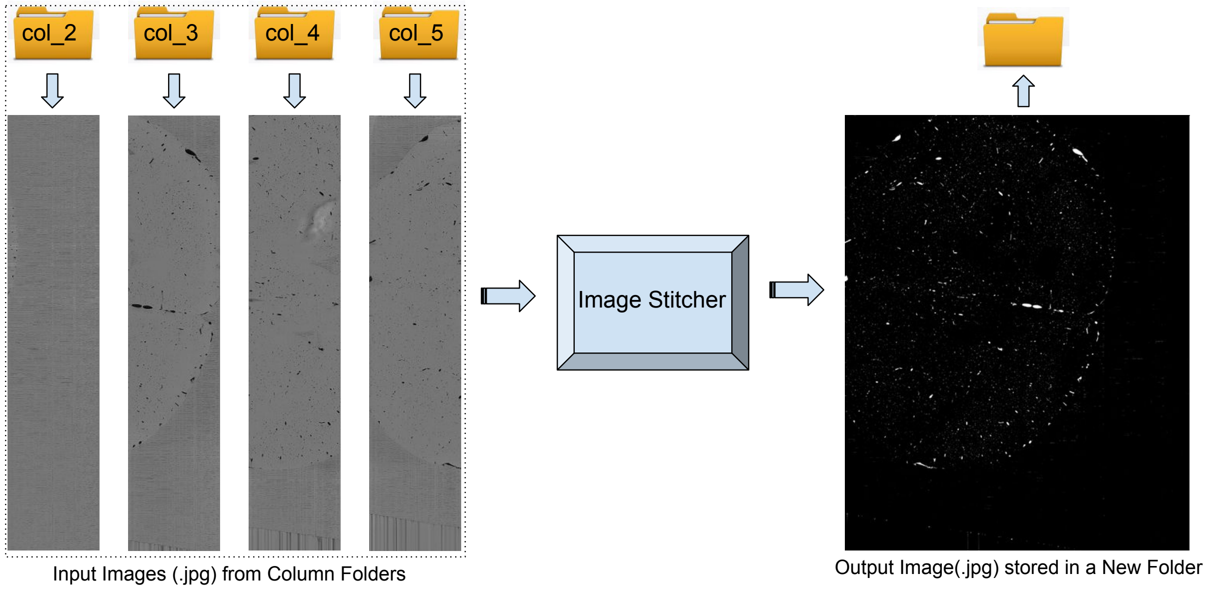
col_2
col_3
col_4
col_5
Image Stitcher
Input Images (.jpg) from Column Folders
Output Image(.jpg) stored in a New Folder

Figure 3.1: Caption

The name of a file stored in any column directory provides information about the date of creation, x coordinate, y coordinate, z coordinate, staining, etc.

A Metadata file is created by the KESM which specifies the information about the tissue sample used, specimen, organ, staining performed, sectioning plane, slice size, knife edge orientation, voxel resolution and the number of columns created. According to this file, we know that the width of each image in a column is 2400 pixels and there are six such column stacks formed. After the preprocessing step, images with non-tissue areas are removed which reduces the number of column stacks to four.

To minimize the artifacts in the output images, after the cropping and normalization process, files containing images of only non-tissue areas are deleted. Such images do not retain the original architecture of the mouse brain and adversely increase the amount of processing dataset. As a result, the column directories now do not have the same number of image files.

3.3 Image Sticher

For each output image, this block picks one image from each column directory, stitches them together and then applies thresholding values to each pixel in the image.

Image Stitcher Controller

The *Controller* decides what files are to be passed on to its *Worker*. Since the number of image files in the column directories is unequal, the *Controller* code needs to perform some searching algorithm to find the right combination of data to be stitched. The idea is to stitch together the image files extracted from the same z coordinate (which is physically the depth level inside the tissue).

First, *Image Stitcher Controller* finds a column directory with maximum number of files, and makes it the base directory for all the z coordinate references. In my case, Column 4 had the maximum number of files, which means it had the maximum number of z coordinate levels.

Second, for each file in the *base directory*, the *controller* starts searching for files with the same z coordinate in other column directories. Once it finds all the images for a particular z coordinate, it calls the *Image Stitcher Worker,* and passes all the four images obtained for that z coordinate.

Image Stitcher Worker

The *worker* tiles the input images received from the *controller*, side-by-side using a corresponding ITK library filter; **TileImageFilter()**. The input for this filter is four images each of resolution 2400×12000, and the output is one tiled image of resolution 9600×12000. The layout for the filter is specified as [4,0] which means it tiles four input images along with their length. The input images are not tiled in the same order as their column numbers. Instead, the image from the last column is the first one to be placed in the image sheet (See Fig.**??**)

The *worker* then binarizes the tiled image using Thresholding method in ITK; **Binary-ThresholdImageFilter()**. The tissue areas in the image are set to a threshold intensity value of 255 that corresponds to *white* and the non-tissue areas are set to a threshold intensity value of 0 which corresponds to *black.* The images are binarized to create a sharp contrast between the background and the tissue imprints in the foreground (See Fig.**??**). This process also eliminates knife chatter or artifact in the image, if present.

The *worker* also creates an output directory for storing the stitched images, in a path specified by the user. The metadata file information is used for labeling the directory.

3.4 Output of Image Stitcher

The stitching operation results in an output directory with 9,626 images. The label of the output directory indicates the ID of the Mouse Brain used for the experiment, and the resolution of the stitched images; `MOU1_BRA_IND_2008_04_9600x12000`.Each output image is saved as a jpeg file and is labeled with its z coordinate information, e.g., `20160414_z5.0670.jpg` (See Fig.).

3.5 Challenge

Unequal Number of Images in Column Stacks

All the serially scanned images are stored digitally in the form of stacks and columns. After that, the tissue areas are cropped and saved from each image. Since, the mouse brain is embedded in plastic, the serial sectioning and imaging data set also include images of non-tissue areas. After the removal of such images from the column stacks, we end up with unequal number of images in the column directories. For reference, we call these images as *relevant images*. It is observed that the mouse brain is embedded in the center of the plastic volume, so there are more number of images in the center column stacks than in the end column stacks. If the images were to be stitched directly, it would result in an unequal width of images in the final output image stack.

Thus, to maintain a constant width, we need to stitch the *relevant images* with *dummy images*, in cases when we do not have the corresponding images for a z level in other columns. A *dummy image* has a constant intensity value throughout, which is equal to the background intensity value of the *relevant image*. In our case, the background intensity value is *0*, which results in a complete black *dummy image* (see Fig.**??**).

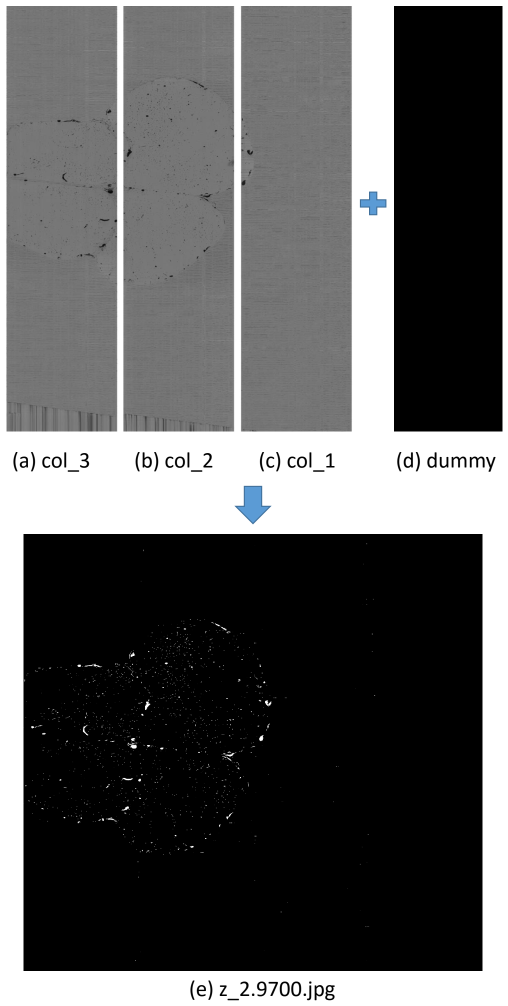

Figure 3.2: Dummy Image appended after stitching of the images from different columns. (a), (b) and (c) represent images from three columns with *relevant images* for the z coordinate. These images are processed to invert the intensity of the foreground and set a constant intensity value for the background. (d) *Dummy Image* of the same resolution as that of the column image, but having the background intensity value throughout the image. (e) Stitched image of resolution 9600×12000.

Chapter 4

Sub-Sampling of Image Stacks

4.1 Overview

For studying the architecture of any biological organ, it is required to develop multidimensional microscopic data resources. In this chapter, we will create those data resources. To create multi-resolution image stacks, we need to sub-sample the original 2D image stack obtained after the stitching process. I used ITK filters to create output image stacks, which are half the resolution of the input image stack. We shrink the 2D images in the x-y plane, and choose to keep alternate files from the original stack in the z direction. So, if the resolution of the image stack created after stitching is A, the sub-sampled image stack will be of resolution A/2. The method is used to create a series of image stacks of resolution: A, A/2, A/4, A/8, A/16, A/32. For example, the original stitched image stack with 9626 images each of resolution 9600×12000 when subsampled, creates a stack with 4813 images each of resolution 4800×6000. We continuously subsample the images stacks until an image resolution lesser than 512×512 is achieved. Finally, we end up creating image stacks of resolution: 9600×12000, 4800×6000, 2400×3000, 1200×1500, 600×750, 300×375 (See Fig.3). The sub-sampled image stacks are stored in directories labeled with information about the stored image resolution and the mouse brain ID used for the experiment.

4.2 Input for Sub-Sampler

The cutting and imaging process in IEROM produces images with fixed width and height. The Image Stitcher block creates a stack of images with the highest resolution possible for any specimen. The highest resolution of images in my case is 9600×12000. Input for the Sub-Sampler block is the image stack directory created by the Image Stitcher block in the previous chapter i.e. `MOU1_BRA_IND_2008_04_9600x12000` , which will be referred as *original stack* in this chapter for the ease of understanding.

4.3 Sub-Sampler

For any input image stack, this block will create a new image stack with images half the resolution of input images selected alternatively from the stack.

Sub-Sampler Controller

This *Controller* simply picks every alternate file from the input image stack and calls the *worker* to down-scale that image. It also passes the destination directory path to the *worker* for saving the down-scaled images. The *Controller* creates a destination directory, and labels it with the Mouse Brain ID information (taken from the input stack) and resolution of the output images. For e.g, if the input stack is `MOU1_BRA_IND_2008_04_9600x12000` then the output image stack will be labeled as `MOU1_BRA_IND_2008_04_4800x6000`.

Sub-Sampler Worker

The *Worker* receives an input image from the *Controller* and shrinks it using an ITK image filter called **ShrinkImageFilter()**. It then saves the output image in the destination directory specified by the *Controller*. The shrink factor for this filter is set to a value of 2, which means the width and height of the output image are half that of the input image. The output image size in each dimension is given by:

outputSize[j] = max(std::floor(inputSize[j]/shrinkFactor[j]), 1);

When the shrink factor is 2, starting from the first pixel, alternate pixels are selected from both x & y dimensions of the input image to make the output image. Note that the physical centers of the input image and output image will be the same, due to which the origin of the output image may not be the same as the origin of the input image.

One cycle of the *Sub-Sampler Controller and Worker* creates half resolution image stack. The cycle repeats until we achieve an image stack of resolution lesser than 512×512. This restriction of selecting the smallest resolution is to retain the entire organ structure in the smallest resolution image stack. Down-scaling this resolution by half will end up with a resolution of 256×256, which is the size of a unit image (will be explained in the next chapter).

4.4 Output of the Sub-Sampler

The idea is to retain the complete organ structure in the smallest resolution, so in our case, we end up with a resolution of 300×375. Further sub-sampling will decimate some tissue traces from the images. So, that puts a limit on the number of sub-sampling processes. Thus, the output of the sub-sampler block is a set of image directories mentioned in the table below:

Subsampled Directories Created	*No. of files*
`MOU1_BRA_IND_2008_04_4800x6000`	4813
`MOU1_BRA_IND_2008_04_2400x3000`	2407
`MOU1_BRA_IND_2008_04_1200x1500`	1204
`MOU1_BRA_IND_2008_04_600x750`	602
`MOU1_BRA_IND_2008_04_300x375`	301

All these directories cumulatively occupy a hard disk space of ≈70 GB.

4.5 Challenges

Overcrowding of Data caused by Sub-Sampling:

After performing the sub-sampling of image stacks, the sub-sampled images of the lower resolution stacks suffer from overcrowding of data. This overcrowding could be very well displayed in the *unit-volume meshes* created later. A lot of points in the *volume mesh* were observed to be unconnected, and they simply appeared as chunks of dots. So, we had to process these images either using some ITK filters or by adjusting their brightness values. Due to the shrinking operation performed, there is a large difference between the intensities of the background and foreground data. Adjusting the brightness of these images, removed the background data and reduced the overcrowding resulting in cleaner *Volumetric Images.* (See Fig .**??**).

Chapter 5

Unit-Volume Creation

5.1 Overview

The Unit-Volume Creator block crops out *unit-image* stacks or *unit-stacks* from the image stack directories created in the previous chapters. In short, they prepare the dataset for generating 3D models.

Handling 3D models created from the higher resolution image stacks to develop a 3D visualization method for the detailed study of the whole mouse brain structure, could be computationally challenging. However, proper division and categorization of data sets can resolve data management issues; making the process more flexible. In this chapter, we explain a cropping mechanism implemented to extract unit volumes of size $256 \times 256 \times 256$ from the 2D image stacks. The value for the resolution standard is selected to be able to view the tissue details clearly in every volume. The result of the process is a set of multi-resolution unit volume meshes. It is important to create such multiscale models, to visualize and analyze different sections of the mouse brain vascular data at various resolutions, facilitating a better understanding of the morphology.

To create *unit-volumes*, we crop out stacks of 256 images each of resolution 256×256 from an original image stack of a particular resolution. Consecutive 2D images of resolution 256×256, referenced as *unit-images* are cropped and extracted from each image of an input image stack. Each *unit-image* is labeled with its *unit-x* and *unit-y* coordinates. Where, *unit-x* and *unit-y* mark 256 pixels in x and y direction. Each image in an input image stack points to a coordinate in the z direction, which is used to name the directory storing all the *unit-images* for that image. Now, these directories save the *unit-images* temporarily, until they are moved to their respective *unit-volume* image stack directories (explained in sections below). Every 256 images in the input image stack mark a unit in z direction. Starting from the first image in the input image stack, a set of 256 images are taken in sequence, to create 256 temporary directories. These 256 temporary directories will create *unit-volume* image stacks of the same *unit-z* coordinate. This process is repeated for all the images in the input image stack, taken in a set of 256 files at once.

Terms	**Definition**
Unit	A unit is defined as 256 pixels
Unit-Image	An Image file having a resolution of 256 × 256
Layer	A unit along the Z direction or a set of 256 image files
Unit-Stack	A directory with 256 unit-images
Unit-Volume	A Volume file having a resolution of 256 × 256 × 256

5.2 Input for Unit-Volume Creator

The image directories created by the Image Stitcher and Sub-Sampler blocks are the input to the Unit-Volume Creator. The Unit-Volume Creator is executed once for each image directory, starting from the highest resolution.

5.3 Unit-Volume Creator

For any input image stack, this block will create *unit-image* stacks which are a pre-requisite for the next chapter where we start building the *3D Models* or *unit-volumes* from the *unit-image stacks*. Let's illustrate the process of building *unit-image stacks* using the original image stack of 9626 images of resolution 9600 × 12000 each. Starting from the highest resolution, the *Unit-Volume Controller* is called once for each sub-sampled image stack.

However, we do not create *unit-volume* image stacks for the lowest resolution image stack which contains 300 images of resolution 300 × 375. If we sub-sample this image stack, we will create an image stack of dimension 150 × 187 × 150, which is even smaller than the *unit-volume* dimension i.e., 256 × 256 × 256. The framework application requires that we visualize the whole-mouse-brain structure at first, which can only be done in the smallest resolution. Thus, for clear visualization, we consider the smallest resolution image stack of dimension 300 × 375 × 300 as one complete *unit-volume* stack.

Unit-Volume Controller

Firstly, the controller creates a text file to save all the original z coordinate values from the image stack. As the image stack contains 9626 images, corresponding 9626 z coordinate values are indexed in a file, which will be used later to retrieve the original z coordinate values while performing volume rendering (explained in later sections).

Then, the *Controller* calls the *Worker* once for each file in the image stack. For each file in the image stack, the *worker* creates a temporary directory to store the cropped *unit-images* extracted from that file. The temporary directory is labeled with the z coordinate value of the image.

Unit-Volume Worker

The *Worker* receives a single file at a time from an image stack. A file name is identified uniquely by its Z coordinate value or its Z position in the image stack. At first, the *worker* creates a directory labeled with this Z value, to store all the cropped *unit-images* extracted from this image file. A *unit* is defined as 256 pixels. The image is passed through an ITK filter called **ExtractImageFilter()**. This filter crops and extracts *unit-images* of resolution 256×256 pixels. By definition, the **ExtractImageFilter()** decreases or changes the image boundary of an image by removing pixels outside the target region. We define the boundary image width and height as 256, so starting from the first pixel, it will crop out the image after 255 consecutive pixels, in both x and y directions.

In our case, we had image files of resolution 9600×12000. So, the filter divides the 9600 pixels along the x direction and 12000 pixels along the y direction into units of 256 pixels each. This results in 37 *unit-x* and 46 *unit-y* coordinates. The *unit-images* cropped out are labeled with their *unit-x* and *unit-y* coordinates such as: `x0_y0.jpg`, `x0_y1.jpg`, ... `x36_y45.jpg`. All these *unit-images* corresponding to a single image file are stored in the temporary directory created by the *worker* and are labeled with the z coordinate value of the image. This step is repeated for all the images in the image stack, resulting in a set of temporary directories, one for each file in the input image stack. (See Fig.**??**). So, for 9626 images in the input stack, 9626 temporary directories are created.

A *Layer* marks 256 image files or a *unit* along the z direction. So, 9626 images in the input stack are divided into 38 layers thus resulting in 38 *unit-z* coordinates. The temporary directories corresponding to a single layer, creates *unit-stacks* with the same *unit-z* coordinate value. The *unit-images* labeled with same *unit-x* and *unit-y* coordinate values across a set of 256 temporary directories, result in the creation of a *unit-stack* directory. These directories are labeled with their *unit-x*,*unit-y* and *unit-z* co-ordinates such as: `Vol_0_0_0`, `Vol_0_1_0`, `Vol_0_2_0`, ... , `Vol_36_45_37` (See Fig.**??**). For example: all *unit-images* labeled as `x0_y0.jpg`, moved out from the 1st *layer* of temporary directories, will create a *unit-volume* image stack called `Vol_0_0_0`. A *unit-image* moved out of a temporary directory is renamed with its original z coordinate value, when saved in the *unit-stack* directory.

A single temporary directory store *unit-images* for a single image file in the image stack which means, we expect 9626 temporary directories, which would be an immensely sized data set. Now, to save time and disk space while creating the *unit-volume* image stacks, we create only 256 temporary directories at once. As soon as a temporary directory is created, *unit images* corresponding to their respective *unit-stacks*, are moved out and stored in their *unit-stack* directories. While a *unit-stack* is being created, all the *unit-stacks* in the same *layer* are created directly after that. So at once, only 256 temporary directories are created and utilized. After all the *unit-images* are moved out of the temporary directories, these empty directories are deleted. In this way, we avoid creating multiple copies of the images and eventually save processing time and hard disk space.

5.4 Output of the Unit-Volume Creator

The *Unit-Volume* Creator block basically outputs *Unit-Stacks* i.e. directories of 256 *unit-images*, which is a pre-requisite for creating *Unit-Volumes* or *3D Models* (explained in the next chapter).

The *Controller-Worker* model is called for all the sub-sampled image stacks, resulting in a huge pile of multi-resolution *unit-stacks*. Each file in the input image directory contributes to a set of *unit-stacks*. Thus, the output of this complete block is a set of directories, one for each input image directory containing *unit-stacks*. However, we do not create *unit-stacks* for the lowest resolution image stack which contains 300 images of resolution 300 × 375. If we sub-sample this image stack, we will create an image stack of dimension 150 × 187 × 150, which is even smaller than the *unit-volume* dimension i.e., 256 × 256 × 256. The framework application requires that we visualize the whole-mouse-brain structure at first, which can only be done in the smallest resolution. Thus, for clear visualization, we consider the smallest resolution image stack of dimension 300 × 375 × 300 as one complete *unit-stack*.

The *Unit-Volume Creator* block creates the following number of *unit-stacks* for each resolution input image directory:

Resolution for Images in the Source Image Stack	**No. of Unit-Stacks Created**
9600 × 12000	57944
4800 × 6000	7866
2400 × 3000	990
1200 × 1500	80
600 × 750	8
300 × 375	1

All these *unit-stacks* cumulatively occupy a hard disk space of ≈94 GB.

5.5 Challenges

Insufficient Pixels in the Images causing Data Loss:

While cropping and extracting *unit-images* from the image files, we observe that if any of the image resolution parameters (i.e. image width in pixels and image height in pixels) is not a multiple of 256, we face data loss. For larger resolution image stacks, the organ structure traces are in the centre of the image. So, we don't have any tissue areas in the end pixels of the image thus, these pixels can be ignored. However, when we are dealing with smaller sub-sampled resolution image stacks, we might have some tissue area traces in the end pixels of the image files. Hence, we need to create dummy pixels of intensity value equal to the background intensity of the original images.

Insufficient Images in the Image Stack causing Data Loss:

While creating the *unit-volume* image stacks, it was observed that if the number of files in the original image stack was not a multiple of 256. Then, the *unit-stacks* for the last unit or *layer* in the z direction is never created, leading to loss of data. To resolve this issue, we implemented a workaround. First, we calculated the number of files undershot in the last *layer*, to make the total number of files in the original stack as a multiple of 256. By doing this, we know how many files are required to be added. Then, we create *dummy image files* of the same resolution as that of the images in the original stack. These *dummy images* are images with a constant background intensity value as explained in the previous section.

Chapter 6

Making 3D Models and Meshes

6.1 Overview

In this chapter, we convert the *unit-stacks* created earlier into actual volumes files. The aim is to create 3D STL (Standard Tessellation) meshes which can be loaded easily to a Web-based framework. We choose the volume output format as STL because it is supported by the XTK APIs. XTK or the *X Toolkit* is a WebGL based scientific visualization toolkit. We use this platform to create our web-based 3D visualization framework (explained in the next chapter). At first, using the ITK classes, we create 3D volumetric images from the *unit-stacks*. These 3D volumetric images generate *unit-volumes*, and are expected to be isotropic in all the three directions. Iso-surfaces for each *unit-volume* are found using the Marching Cube algorithm in VTK, and are then saved as a 3D mesh in an STL file format. A *STL Mesh* file is labeled with the name of its *unit-stack* directory. So, for the highest resolution image stack, we create *Meshes* from `Vol_0_0_0.stl`, `Vol_0_1_0.stl`, . . . to `Vol_36_45_37.stl`. This process is repeated for all the sub-sampled *unit-stacks*, resulting in a set of multi-resolution *unit-volume meshes*. The *Iso-Surfaces* are the distribution of scalar data in a volumetric image. Marching Cube algorithm uses patterned cubes or isosurfaces to approximate contours in a volumetric image. VTK supports the marching cubes algorithm with *VtkMarchingCubes* class, which requires a volumetric image input as a VTK data object, and creates an output in VTK poly data format. We can specify the threshold value and the number of contours while using VtkMarchingCubes, to generate the 3D surface of the object. 3D Model Maker is the last block in our Image Processing Pipeline.

6.2 Input for the 3D Model Maker

The directories containing *unit-stacks* created in the previous chapter are the source for this block. Each directory contains *unit-stacks* of a particular resolution. We feed these directories one by one to the *3D Model Maker Controller*, so the *Controller-Worker* model is called once for each resolution.

6.3 3D Model Maker

This block creates 3D models or meshes from the *unit-stacks* fed to it. The block is currently capable of producing three different formats of volume files: TIFF (Tagged Image File Format), STL (Standard Tessellation Language), VTK (Visualization Toolkit). The user can select the output file format. We produced different file formats for different use cases.

The 3D Model Maker block utilizes two major toolkits: ITK and VTK. The ITK filters are used to process 2D images and the VTK filters are used for creating the 3D meshes.

3D Model Controller

The *Controller* receives a directory for a single resolution at a time as input. It will call the *worker* once for every *unit-stack* in that directory. A user needs to feed the source directory path and output file format as input arguments to make a call to the *controller*. The *controller* will first create an output directory and set it as the destination directory for the *worker* to save all the output volume files.

3D Model Worker

The *Worker* will create one volume file every time it is called and will save it in the destination directory path provided by the *controller*. The task performed by the *worker* is divided into the following steps:

Create Sequential File Names

The process of generating a volume file from slices or 2D images, requires to have the input image files in an ordered sequence. The *Name Generator* will convert the file names in the *unit-stack* into an ordered sequence of file names. For example: initially a *unit-stack* labeled as `Vol_0_0_0` have files named as `z_0.1325.jpg`,`z_0.1335.jpg`,... `z_0.3875.jpg`. When this *unit-stack* is passed through the *Name Generator* block, it generates file names as `0.jpg`, `1.jpg`,... `255.jpg`.We utilize an ITK filter called **NumericSeriesFileNames** to perform this task.

As a next step the *worker* will convert these sequence of files into 3D Models and Meshes. According to users choice, it can be either a TIFF, STL or VTK file format.

TIFF Model Generation

For generating the TIFF files, we use an ITK filter called **TIFFImageIO**, to combine the sequence files and write an output TIFF volume file. **ImageFileWriter()** is an ITK class that interfaces with this **ImageTIFFIO()** class and writes an out the data to a single file.

STL Mesh Generation

For creating STL files, we first convert the ITK image to a VTK image using an ITK filter called **ImageToVTKImageFilter()**. This filter will convert an ITK data pipeline to a VTK data pipeline and will take care of the details of the connection of the two pipelines. After this step, we end up with a 3D structured point set.

As a second step, we convert these 3D structured point set to one or more isosurfaces using a VTK filter called **vtkMarchingCubes()**. By definition, this filter requires that we specify one or more contour values to generate the isosurfaces. Alternatively, we can specify a min/max scalar range and the number of contours to produce a series of evenly spaced contour values. We call this value as the *iso-surface value*; this value is directly proportional to the scalar values extracted. Higher values of this parameter denote a greater number of scalar data set extraction leading to lesser data loss. However, a higher value also means that the *marching cube algorithm* will repeat the generation of iso-surfaces for the larger number of times, resulting in higher computational time. Therefore a balance is to be maintained. We choose the iso-surface value as 100 for our data set.

At the third step, we compute normals for the polygonal mesh using a VTK filter **vtkPolyDataNormals()**. By definition, this filter computes point and/or cell normals for a polygonal mesh. The user specifies if they would like the point and/or cell normals to be computed by setting the ComputeCellNormals and ComputePointNormals flags. The computed normals (a vtkFloatArray) are set to be the active normals (using SetNormals()) of the PointData and/or the CellData (respectively) of the output PolyData. The filter can reorder polygons to insure consistent orientation across polygon neighbors. Sharp edges can be split and points duplicated with separate normals to give crisp (rendered) surface definition. It is also possible to globally flip the normal orientation. The algorithm works by determining normals for each polygon and then averaging them at shared points. When sharp edges are present, the edges are split and new points generated to prevent blurry edges.

At the fourth step, the polygonal data created is mapped to the graphics primitives using a VTK class called **vtkPolyDataMapper()**. This class maps polygonal data to the rendering/graphics hardware/software. Then we use the **vtkActor()** class to represent the object in a rendered scene. By definition, it inherits functions related to the actors position, and orientation and has scaling and maintains a reference to the defining geometry (i.e., the mapper), rendering properties, and possibly a texture map. vtkActor combines these instance variables into one 4x4 transformation matrix as follows: [x y z 1] = [x y z 1] Translate(-origin) Scale(scale) Rot(y) Rot(x) Rot (z) Trans(origin) Trans(position).

Finally we use a **vtkSTLWriter()** to write stereo lithography (.stl) files in binary form. Stereo lithography files only contain triangles. If polygons with more than three vertices are present, only the first three vertices are written.

6.4 Output of the 3D Model Maker

The *3D Model Maker* is executed for all the different resolution directories, where for each directory it converts all its *unit-stacks* into volume files. So, the output of this complete block is a set of directories with multi-resolution 3D models/meshes.

The iso-surfaces are only generated for volumetric images with tissue areas, as only those areas have scalar data values. Thus, the number of STL Meshes created is not always equal to the number of *unit-volume* image stacks. A *3D-Model-Maker* converts the volumetric images into *STL-Meshes* and the output is displayed in the table below:

Resolution for Images in the Source Image Stack	**No. of STL Meshes Created**
9600×12000	49884
4800×6000	7055
2400×3000	904
1200×1500	74
600×750	8
300×375	1

All the STL's cumulatively occupy a hard disk space of ≈1TB.

6.5 Challenges

Unequal Spacing in the 3D Meshes Created

The three-dimensional serial sectioning performed in KESM is not spaced equally in the *x, y, z* directions. The spacing ratio maintained by the instrument is 0.625: 0.7: 1 in the *x, y and z* directions respectively. This unequal spacing obstructs the construction of volume meshes of exact 256^3 size. The data doesn't align properly in the volume mesh and appears stretched in one direction. To overcome this issue, we set the spacing ratio of the 3D volumetric image as 1: 1: 1 before creating iso-surfaces using marching cube algorithm (See Fig.**??**).

Selecting a Proper Iso-Surface Value

For creating the iso-surfaces with the marching cube algorithm, we need to set the min/max scalar range and the number of contours to generate a series of evenly spaced contour values. While creating the unit-volume STL meshes, we need to be careful in selecting a proper value for the number of contours to be generated. Since the marching cube algorithm will make repeated generation of iso-surfaces for that number of times. So, for higher resolutions, creation of STL meshes will be a time consuming task. Keeping a lower value of contour number will take lesser time to create the meshes. But, we need to consciously choose a value which is neither too low nor too high to maintain the complete data visibility for higher resolutions. However, the same value might not work for lower resolutions as it will lead to data losses in them. So, we need to select different values of *contour number* for creating multi-resolution volume meshes. It has been observed that a contour value of 100 works well for *unit-volumes* image stacks of 9600×12000 resolution, and a value of 180 works well for the *unit-volume* image stack of 300×375 resolution. We somehow maintain values between 100 and 180 for the *unit-volume* of resolutions between these two resolutions.

Part III

Web-Based 3D Visualization Framework : Implementation

Chapter 7

3D Brain Atlas: Web-Based Real-Time 3D Visualization Framework

7.1 Overview

For medical data visualization, most users are required to have an appropriate software installed on their local desktop computer, which may involve a lengthy download or installation process. In this model, software developers have to cater for different operating systems when writing these programs. When they have finished an update, it will again have to be delivered to the user (usually via download) and installed. To circumvent these issues, the goal of this chapter is to provide a medical image viewer that runs in any web browser. This web-based application will aim to provide the same functionality as a desktop solution without being tied to a particular operating system.

3D Brain Atlas is a web-based real-time 3D visualization framework designed to study the morphology of the Mouse Brain Vascular Networks and analyze it at multiple resolutions. With the IEROM Image Processing Pipeline explained in the previous chapters, it is possible to acquire teravoxel sized three-dimensional microvascular images of the whole mouse brain with sub-micrometer resolution. We design this web-based framework, to be able to visualize and share these teravoxel volumes across research communities efficiently. We believe that this novel framework for real-time 3D visualization can facilitate data sharing of terabyte-sized three-dimensional images easily.

In this chapter, we explain the design of the Web-Based Application created for managing visualization and interaction with the 3D meshes. We utilized the X Toolkit (XTK); *a WebGL for scientific visualization* [28], to make the web-based 3D visualization framework for teravoxel volumes. A simple web server is implemented by using *Node.js* [22]. Then through a custom javascript code, we designed the graphical user interface to control the whole functionality of the web application. The graphical user interface (GUI) is designed using a JavaScript controller library called *dat.GUI* [7]. The graphical user interface is used to display the details of the volume loaded and to interact with it: *x,y,z* unit coordinates and the resolution (see Fig.**??**). The URL of this web-based visualization framework for vascular network data set is http://jrkwon.com/3dbrainatlas.

7.2 Tools and Environment Settings

To create an interactive browser-based application which is useful for displaying medical image data, the available tools for creating 3D graphics in a web browser have to be considered. Generally, in the past web browsers have provided several different methods to display 2D and 3D graphics on screen, and only recently with the introduction of HTML5 and WebGL has a widely-conformed standard emerged. At a very basic level, a website can be written with just *HTML* (Hyper Text Markup Language) code. HTML is written with tags such as `<html>, <body>, </body>, </html>` and then the code is converted into a tree format of JavaScript node objects or Document Object Model (DOM) by the web browser.The purpose of this is to provide a programmatic interface for scripting (removing, adding, replacing and modifying) this live document using *JavaScript*. Different tags can be added to an HTML element, such as the class tag, which gets used to hook extra attributes into the element. For adding attributes such as font, color, dimensions and many more, *CSS* is used. With CSS an HTML element can be assigned a class (via the class tag), which will refer it to specifc set of CSS rules for setting style and look. We used the above basics of developing web applications for designing the *3d Brain Atlas.*

The web-server is implemented using *Node.js* which is a JavaScript runtime built on Chrome's V8 javascript engine. As an asynchronous event driven JavaScript runtime, Node is designed to create scalable network applications. HTTP is a first class citizen in Node, designed with streaming and low latency in mind; this makes Node well suited for the foundation of our web framework. We installed an HTTP server in Node and started it. The web-page could be easily accessed with the either the URL or http://localhost:8080.

WebGL is a 3D graphics API for the Web and we utilized *XTK* which is a WebGL framework that provides easy-to-use APIs for scientific data visualization on the web. We used some basic APIs such as **X.renderer3D()** for creating a new 3D renderer inside a given DOM element and **X.mesh()** to create a mesh or displayable object loaded from an .STL file.

We designed a lightweight Graphical User Interface for our web framework using a JavaScript controller library called *dat.gui.* This library creates an interface to easily change variables in JavaScript. The easiest way to use dat.gui in our code is by using the built source at *build/dat.gui.min.js.* These built JavaScript files bundle all the necessary dependencies to run dat.gui.

7.3 Input Data Set

The web-based framework is a WebGL based application built using the X Toolkit. This toolkit supports many different surface models/mesh file formats such as `.STL` (Standard Tessellation), `.VTK` (Visualization Toolkit poly data), `.OBJ` (Wavefront format) and `.FSM`, `.INFLATED`, `.SMOOTHWM`, `.SPHERE`, `.PIAL`, `.ORIG` (Freesurfer meshes). It also supports volume files such as .DICOM, .DCM (Multi file DICOM). However, surface models/meshes provide better architectural rendering and define the shape of the polyhedral object in 3D computer graphics and modeling. Thus, for the detailed study and analysis of the structure of the Mouse Brain Vascular Networks, we prefer rendering surface models/meshes.

The idea is to drop the files on the web server and have them ready for rendering. Therefore, we select a file format which is supported by XTK for fast loading and can correctly regenerate the Mouse Brain Vasculature data points. An STL file describes a raw unstructured triangulated surface by the unit normal and vertices (ordered by the right-hand rule) of the triangles using a three-dimensional Cartesian coordinate system. STL coordinates must be positive numbers, there is no scale information, and the units are arbitrary. We used the STL Meshes created by the 3D Model Maker block in the image processing pipeline, as input data set to the web-based framework. The STL meshes can be directly uploaded to the web-server from the local disk space.

7.4 Graphical User Interface Design

Before we step into the Graphical User Interface Design of the *3D Brain Atlas*, let's understand the basics of 3D graphics which is important to design this framework. 3D Graphics are usually defined by a space in Cartesian coordinates in which reside three-dimensional objects, as well as a camera object through which the scene is viewed with help of a projection matrix (see Fig.**??**). 3D graphics is computationally much more expensive than 2D graphics due to its more complex mathematics. On top of that, rendering a 3D scene can mean that the image should be refreshed 60 times a second, which poses a challenge for many web browsers.

Now let's describe the implemented features of the **3D Brain Atlas** Graphical User Interface in steps:

Main Page Layout

The main view that the user is presented with shows a data display pane with a mesh loaded and control panels on the right side of the page (see Fig.**??**). At first, when we open the web URL or the http://localhost:8080, after starting the web server. The web page opens with the smallest resolution mesh data file i.e. $300 \times 375 \times 301$ sized volume, loaded at the onset of the page. The background properties of the *CSS* are modified to give it a 3D rendering background effect. The linear gradients are set to change from the top of the page, with color-stop values of `rgba(255, 255, 255, 1)`, `rgba(199, 199, 199, 1)` and `rgba(48, 43, 48, 1)`.

Control Panel

Resolution Tab

We added controls to switch resolutions and display different meshes using *dat.gui* library. We added a drop-down menu for selecting the desired resolution, where the drop-down is a list of strings with values: 300×375, 600×750, 1200×1500, 2400×3000, 4800×6000 and 9600×12000. At an instance, an STL mesh of *unit-volume* i.e. of 256^3 dimension is loaded, according to the user scroll input depending upon their region-of-interest and angle-of-perspective.

Mesh Unit Specs Tab

A user can manually select the *unit-x*, *unit-y* and *unit-z* coordinates for a particular resolution from these drop down menus. If a valid STL mesh file for such a combination exists, it will be loaded to the web framework. The *unit-x*, *unit-y* and *unit-z* coordinate entries provided by the user are taken as input for framing a file name appending the string `Vol_(unit-x)_(unit-y)_(unit-z).stl`. This file name is then searched in the respective directory for the selected resolution.

FilePath

This tab will display the complete path name of the mesh file currently loaded in the visualization pane. It would show the user if the mesh were loaded from the local disk or the cloud space. For initial development stage, we used files from the local disk space but later after we uploaded all the mesh data files in the cloud data space, which makes the loading and caching process faster.

Rotate Tab

For better visualization and understanding of the architecture of the mouse brain vascular networks, it is important that the user can interact with it. Interaction with the 3D graphic model/mesh means that the user should be able to perform Rotate, Zoom, and Pan operations on the model/mesh at least. The X Toolkit supports a list of control modes to interact with renderers. Tables 7.1 and 7.2 enlists the types of control modes defined by the X Toolkit. They are enabled by default but can be disabled on request.

The *Pan* and *Zoom* operations are same for 3D and 2D renderers. However, the *Rotate* operation using either the mouse or keyboard is mostly for 2D renderers. Thus we disabled the control modes for *Rotate* operations and built our control mode for rotation in all three

Mouse Interaction	**Type of Graphics Interaction**
LEFT CLICK + MOVE	Rotate the scene or Window/Level adjustment in 2D
SHIFT + LEFT CLICK or MIDDLE CLICK	Pan the scene
MOUSE WHEEL UP	Zoom In, fast
MOUSE WHEEL DOWN	Zoom Out, fast
RIGHT CLICK + MOVE UP	Zoom In, fine
RIGHT CLICK + MOVE DOWN	Zoom Out, fine

Table 7.1: List of mouse control modes used for interacting with the renderer. We used the mouse control modes for panning, zooming in and zooming out operations.

Keyboard Interaction	**Type of Graphics Interaction**
ARROW KEYS	Rotate the scene
SHIFT + ARROW KEYS	Pan the scene
ALT + UP	Zoom In, fast
ALT + DOWN	Zoom Out, fast
ALT + LEFT	Zoom In, fine
ALT + RIGHT	Zoom Out, fine
r	Reset the view to default based on bounding box of all visible objects or a manual configured camera position

Table 7.2: List of keyboard control modes used for interacting with the renderer. When multiple renderers exist in the document, the one under the mouse listens to the keyboard interaction

directions. The **Rotate Tab** consists of three controllable *checkboxes* which can be used for both monitoring and control. Each checked-box will overwrite the previous rotate operation.

Update

After the user makes any changes to any of the above tabs either *Resolution* or *Mesh Unit Specs*, this *Update* button will actually perform the changes. The user can switch other resolution or change the unit specs of the mesh for the selected, and then click *update*. This will upload the corresponding mesh file into the renderer. If there is no file available for the combination selected, the same data will continue to be rendered in the visualization pane.

Mesh Details Tab

This tab is a read-only tab for the user to learn more about the morphological details of the mesh loaded by the renderer. As a part of previous research work done in our lab, a research student had worked on extracting details of the *unit-volumes* of the mouse brain vasculature using a 3D Analysis Software called Vaa3D []. The details of the extracted features were saved in .Json files, one file per *unit-volume*. The *Mesh Details* tab will showcase those details such as: number of branches, number of ...

7.5 Results

The framework automatically loads the smallest resolution volume when first launched. The resolution and *unit-x*, *unit-y*, *unit-z* coordinates can then be selected to load the other higher resolution volume *Meshes* (See Fig.**??**). It is observed that without caching, the time taken to display the data is about 10-15 seconds. The URL of the Web-based visualization for a KESM vascular network data set is http://jrkwon.com/3dbrainatlas.

7.6 Challenges

Slow loading of the STL Meshes

The Whole Mouse Brain Vascular Networks are complex meshes. The STL meshes created for a single *unit-volume* have around 2,400,000 vertices and are approximately of size 42000 KB. Loading and rendering such huge mesh files from the local disk space is a time-consuming task, leading to slow loading of the STL meshes in the display pane. Therefore to avoid such latency, we upload all the STL files created by the 3D Model Maker block to a cloud based data storage space, so the web-based application directly fetches the mesh data files from the same web data storage space instead of the local computer disk. This process speeds up the loading and unloading of the large mesh data files on the web-application.

Part IV

Virtual Reality Framework : Methods and Implementation

Chapter 8

Virtual Reality for Mouse Brain Vascular Network Study

8.1 Overview

Virtual Reality is an emerging computer technology to enable a user to be physically present in a virtual environment. It requires Virtual Reality headsets in combination with physical spaces or multi-projected environments to simulate audio, visual and other sensations for a real life-like experience. With a headset and motion tracking, VR lets you look around a virtual space as if you're actually there.

Medical Image Visualization has been struggling with the need to accurately analyze and decipher information from the vast volumes of data generated by multiple imaging modalities that exist today. Virtual Reality has a variety of potential benefits for many aspects of Medical Imaging. Fusing the evolution of advanced medical imaging systems and Virtual Reality, has lead to the development of powerful computational techniques to visualize, analyze and use these images for advanced use in medical practice.

In our research work, we aim to study the Mammalian (Mouse) Brain morphology to be able to compare a diseased brain with a healthy one. The whole Mouse Brain Vasculature data acquired from the IEROM is a motley bunch of interconnected blood vessels. To properly study and analyze the architecture of such data sets, we need sophisticated visualization methods which can deduce even minor morphological details accurately. With the advancement of Virtual Reality in the field of Medical Image Visualization, it is now possible to design Virtual Reality frameworks to visualize and interact with 3D medical images. Therefore, in this chapter, we will explain our approach to create such a VR framework for the study and analysis of Mouse Brain Vascular Networks through better visualization and interaction techniques. The aim is to load the Mouse Brain Vascular volumes in a virtual reality space and be able to walk-through the structure and interact with it.

8.2 Tools and Environment Settings

To achieve the aim of visualizing and interacting with the whole mouse brain vascular data models in Virtual Reality, it is first important to get the right tools and setup the right environment.

Selecting the VR Headset

There are many VR headsets available in the market today from leading companies like HTC, Sony, Oculus, Google, Samsung, etc. However Virtual Reality can be best realized with an Oculus Rift. Oculus Rift in comparison with other headsets has a better per eye resolution of 1080×1200, refresh rate of 90 Hz and a wider field of view of 110 degrees. Developing a VR framework on a PC using an Oculus platform is also easy to integrate with most of the modern game engines and SDKs. Powerful game engines like *Unity* and *Unreal* come with out-of-the-box support for the Oculus hardware, such as Platform SDK, tutorials, sample scenes, custom utility packages, and more. Therefore, we select the **Oculus Rift** Headset for developing and realizing our VR framework.

Leap-Motion Controller

To be able to interact with 3D models in the virtual reality with our bare hands, we mount a *Leap Motion Controller* onto the Oculus Rift Headset. Leap Motion's hand tracking technology is designed to be embedded directly into VR/AR headsets. It is a hardware sensor device that supports hand and finger motions as input analogous to a mouse but requires no hand contact or touching. The device consists of two cameras and three infrared LEDs that track infrared light with a wavelength of 850 nanometers, which is outside the visible light spectrum. The Leap Motion Controller comes with an easy and flexible VR integration packages for game engines like *Unity* and *Unreal.*

Unity Game Engine

Unity is a cross-platform game engine developed by Unity Technologies. It is an all purpose game engine, and supports both 2D and 3D graphics, drag and drop functionality and scripting through C# and UnityScript. UnityScript is a proprietary scripting language which is syntactically similar to JavaScript.

Oculus VR Headset Support in Unity

Unity has built-in support for certain VR devices including Oculus Rift and Oculus Development Kit2 (DK2). Oculus Utilities Unity Package assists all VR development needs, including assets, scripts, and sample scenes (see Table 8.1). The sample scenes are helpful for developers to have a starting reference.

Unity Utilities	**Usage**
OVRManager	An interface for controlling VR camera behavior
OVRPlayerController	A VR first-person control prefab
OVRInput	A unified API for Xbox controllers, Oculus Touch, and Oculus Remote
OVRHaptics	An API for Oculus Touch haptic feedback
OVRScreenshot	A tool for taking cubemap screenshots of Unity applications
Adaptive Resolution	Automatically scales down resolution as GPU exceeds 85

Table 8.1: List of Unity Utilities to support Oculus VR devices.

Leap-Motion Controller Support in Unity

Unity provides Core Assets and Modules for Leap Motion to make it easy to design hands, user interfaces, and interactions. Leap Motion's Core Assets provide the foundation for VR applications with a minimal interface between Unity and the Leap Motion Controller. With Core, you can render a basic set of Leap hands or attach arbitrary objects to hand joints. Unity Modules are extensions built on top of our Unity Core Assets to provide additional features and capabilities.

Unity Editor Interface Basics

When you open a project in Unity, the main editor window opens which is made up of tabbed windows which can be rearranged, grouped, detached and docked. The default arrangement of windows gives you practical access to the the most common windows (see Fig.**??**).

- *Project Window*: Displays library of assets that are available to use in the project. The imported assets of the project appear here.
- *Scene View*: Allows to visually navigate and edit scenes. The scene view can show a 3D or 2D perspective, depending on the type of working project.
- *Hierarchy Window*: A hierarchical text representation of every object in the scene. Each item in the scene has an entry in the hierarchy, so the two windows are inherently linked. The hierarchy reveals the structure of how objects are attached to one another.
- *Inspector Window*: Allows to view and edit all the properties of the currently selected object. Because different types of objects have different sets of properties, the layout and contents of the inspector window will vary.

- *Toolbar*: Provides access to the most essential working features. On the left it contains the basic tools for manipulating the scene view and the objects within it. In the centre are the play, pause and step controls. The buttons to the right give access to Unity Cloud Services and Unity Account, followed by a layer visibility menu, and finally the editor layout menu (which provides some alternate layouts for the editor windows, and allows to save our own custom layouts). The toolbar is not a window, and is the only part of the Unity interface that you can't rearrange.

Asset

An *Asset* is a representation of any item that can be used in the game or project. An asset may come from a file created outside of Unity, such as a 3D model, an audio file, an image, a mesh, or any of the other types of file that Unity supports.

Scenes

Scenes contain the environments and menus of a game. Think of each unique Scene file as a unique level. In each Scene, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces. When you create a new Unity project, your scene view displays a new Scene. This Scene is untitled and unsaved. The Scene is empty except for a Camera (called **Main Camera**) and a Light (called **Directional Light**) (see Fig.**??**).

GameObjects

The *GameObject* is the most important concept in the Unity Editor. Every fundamental object in a game is a GameObject, from characters, props, and collectible items to lights, cameras, special effects and scenery. They do not accomplish much in themselves but they act as containers for **Components**, which implement the real functionality.

A GameObject always has a *Transform* component attached (to represent position and orientation) and it is not possible to remove it. The other components that give the object its functionality can be added from the editor's Component menu or from a script (see Example Fig.**??**).

Transforms

The Transform is used to store a GameObject's position, rotation, scale and parenting state and is thus very important. A GameObject will always have a Transform component attached, it is not possible to remove a Transform or to create a GameObject without one. A Transform can be edited in the Scene View or by changing its properties in the Inspector.

Lights

Lights are an essential part of every scene. While meshes and textures define the shape and look of a scene, lights define the color and mood of the 3D environment.

Cameras

Cameras in Unity are used to display the game world to the player. You will always have at least one camera in a scene, but you can have more than one. We used the Leap Motion Head Mounted camera in our scene enabled with VR suport.

Required Hardware Setup

System Prerequisites

To power up an Oculus Rift and design VR frameworks with Rift and Leap Motion Controller, the host computer needs to meet or exceed some system specifications as mentioned in Table 8.2. We used an Oculus recommended desktop; *Alienware X51 R3 i5 Desktop.* It has Intel core i5 6400 2.7 GHz, 16GB RAM, 256GB Solid-State Hard Drive, and NVIDIA GeForce GTX970 graphics card.

Oculus Rift Hardware Setup

- Unbox the Oculus Rift kit that contains: Oculus Rift Headset with Quick Start Guide, Oculus Camera Sensor with Built-in Stand, Small Plastic Tool for Integrated Headphone Removal, Oculus Remote with Integrated Battery, Xbox One Wireless Gamepad controller, Xbox One USB Wireless Receiver for Gamepad and Instructions, 2 x AA Batteries for Gamepad, 2 x Oculus Logo Stickers, and Oculus Lens Wipe Cloth.

- Remove the protective films from the headset lenses and from the sensor lens (shiny side of the sensor body).

- Connect the HDMI end of the headset cable to the HDMI port on your graphics card. Note: Don't use the HDMI port on your motherboard, if you have one. If you're not sure which HDMI port to use, try the one on the narrower and simpler panel on the back of your computer.

- Connect the USB end of the headset cable to a USB 3.0 (blue) port on your computer.

- Connect the sensor cable to another USB 3.0 (blue) port on your computer.

- You'll see three green icons in the lower left of the Oculus screen indicating that the headset and sensor have both connected successfully. Note: If you see red or yellow warning icons or have any other issues try Rift Hardware Troubleshooting.

- Follow the link: https://www3.oculus.com/en-us/setup/. Download and run the Rift's setup tool, which will automatically install all the software required to use the Rift. It will further guide the user to setup and configure the Rift headset, sensor and other hardware.

- Gently pull the clear plastic tab out from the battery door on the back of the Oculus Remote. This tab keeps the batteries from running down during shipping. Press and hold the select button, which is in the center of the navigation disk, to pair the remote with your headset. See Table 8.3 to know how to use the Oculus Remote.

- The Oculus sensor makes sure what the user is seeing in Rift tracks their position and movement. While setting up the Oculus sensor, we need to enter the correct height of placement when asked. This helps make sure that the VR environment looks right towards the user.

- It is required to find a good place for placing the Oculus sensor. The ideal position is:
  - Between 3 feet (1 m) and 6 feet (2 m) away from your head.
  - Where nothing will get in the way. Try crouching and stepping sideways to make sure the edges of the desk or shelf you're using won't block your view of the sensor. Don't use an area where people will be walking between you and the sensor.
  - Where you'll normally be facing. Keep the sensor inside your starting field of view.
  - On a stable surface. Don't put the sensor on top of your monitor or computer, or anywhere else it will vibrate or wobble.
  - Slightly above your headset. If that's not possible, it's fine to have it below your head instead.

- Take the Rift headset to the spot where you plan to use it. Make sure the sensor lens (the shiny side) is pointing at your head.

- Gently adjust the angle of the sensor body on the sensor stand if necessary.

- Hold the headset just in front of you and move it slowly side to side. You may also need to swing it gently down toward the floor and back up in front of your head. You'll be notified when the Oculus sensor has found your headset.

- After your Rift headset fits you properly, it will be quick and easy to put it on from now on.
  - Open the side tabs on the main strap. Fasten the tabs to the middle of the strap arms as a starting point.
  - Open the top tab, loosen the top strap all the way, and leave it loose. Angle the On-Ear Headphones outward.

  - Hook the tracking triangle on the back of your head. Tighten the side tabs slightly and tighten the top strap until you feel the weight balanced around your head.
  - Rotate the headphones into position and push them onto your ears.
  - Find the lens slider on the underside of the headset. It controls lens spacing inside the headset. Push and hold it into the headset, then slide it to get the sharpest possible image

- Clear the surrounding area where you stand wearing the VR headset, at least a few feet in all directions. Move anything that might get in your way, like furniture or other objects (see Fig. **??**.) Always be aware of your surroundings while using the Rift.
- Once the hardware setup for the Oculus Rift is completed:
  - Stand in the spot where you'd like to use Rift and face the Oculus sensor
  - Slide your wrist through the Oculus remote's lanyard
  - Put on the Rift headset
  - Move the headset very slightly up and down on your face until the image is sharpest
  - Then push in the lens slider on the bottom of the headset and slowly slide it from side to side until the image is sharpest
  - Press the select button on the Oculus remote. You'll see a few short experiences to get you started in VR.

Leap Motion Controller Hardware Setup

- Peel Off Sticker: Remove the Sticker from the top of the controller (see Fig. **??**).
- Plug into the Computer: Use the USB cable included in the box. Shiny side of the controller faces up and the green light faces towards you (see Fig. **??**).
- Be Comfortable and Calibrate the Controller: Place controller in front of you. Make room to rest your elbow (see Fig. **??**).
- Clean the Headset: Make sure that the surface on the Oculus VR headset is fully cleaned. It is recommended using rubbing alcohol and a clean cloth.
- Align the mount: align the angled sides of the adhesive mount with the angled details on the Rift. Make sure the orientation of the mount is as shown in Fig. **??**.
- Fix the mount: Attach the VR Developer Mount to your headset. Firmly press the adhesive mount into place pressuring full contact over the entire surface. Allow at least 1 hour for the mount to adhere to the surface. Then use the free cable extender bundled with the mount to connect the controller directly to the computer.

System Specs	**Recommended**	**Minimum**
Graphics Card	NVIDIA GTX 1060 / AMD Radeon RX 480 or greater	NVIDIA GTX 1050 Ti / AMD Radeon RX 470 or greater
Alternative Graphics Card	NVIDIA GTX 970 / AMD Radeon R9 290 or greater	NVIDIA GTX 960 4GB / AMD Radeon R9 290 or greater
CPU	Intel i5-4590 equivalent or greater	Intel i3-6100 / AMD FX4350 or greater
Memory	8GB+ RAM	8GB+ RAM
Video Output	Compatible HDMI 1.3 video output	Compatible HDMI 1.3 video output
USB Ports	3×USB 3.0 ports, plus 1×USB 2.0 port	1×USB 3.0 port, plus 2×USB 2.0 ports
OS	Windows 10	Windows 8.1 or newer

Table 8.2: System Specifications of a computer required to power-up the connected Oculus VR device.

Oculus Remote Buttons	**Usage**
Navigation disk	Move up, down, back, or forward through menu options
Select button	Select a menu option, or to select an item in a game or app
Back button	Cancel an option, or move back a screen
Volume Down and Volume Up	Control the volume in the On-Ear Headphones
Oculus button	Press to access the Universal Menu from almost anywhere in Rift

Table 8.3: List of Oculus Remote button's functionalities

Software Setup

Unity Game Engine Installation

Game Engines can significantly reduce the time and effort it takes to build Virtual Reality experiences with Oculus devices like Rift. *Unity* well supports Oculus devices. We downloaded the latest version of Unity Installer. The installer uses a *Download Assistant* which will direct a user to install correctly. It provides options to install specific components of the Unity Editor, according to user's requirements.

We need to import Unity Packages for specific applications we plan to use in our project, like Oculus and Leap Motion. To import any package, it is required that we delete any previous version of the same package, then either create a new project or save the current scene. For importing any package: Select *Assets → CustomPackage → and select the Utilities Unity Package* to import it. Alternately, you can locate the *.unityPackage* file in your file system and double-click it to launch.

Oculus Software Setup

- Unlike Oculus DK2, Oculus Rift comes with an easy software installation process. After the user has successfully connected and completed the hardware setup, *Rift's Setup Tool* will automatically install all the software required to use it.
- The list of software's that the tool will install includes the *Oculus Runtime* and *Oculus App.* We need to have this App and calibrate the Rift. A user needs to create an Oculus Account to use the Rift. After all the software is installed correctly. Test the Rift by opening any example scene or movie from the Oculus App.
- Each VR device requires appropriate runtime installed on the machine. To develop and run Oculus within Unity, we need to have the *Oculus runtime.*
- For any project in Unity, Oculus support is enabled by checking *Virtual Reality Supported* in the *Edit → ProjectSettings → Player → OtherSettings Configuration tab* (see Fig.**??**). Unity automatically applies position and orientation tracking, stereoscopic rendering, and distortion correction to your main camera when VR support is enabled.
- The *Unity Utilities Package* is easy to download and contain useful prefabs, C# scripts, and other resources to support VR projects in Unity. The package includes an interface for controlling VR camera behavior, a first-person control prefab, a unified input API for controllers, advanced rendering features, object-grabbing and haptics scripts for Touch, debugging tools, and more.
- The *Oculus Unity Sample* Framework includes sample scenes and scripts illustrating common VR features such as locomotion, in-app media players, crosshairs, UI, interaction with Game Objects with Oculus Touch, and more.

When Unity virtual reality support is enabled, any camera with no render texture is automatically rendered in stereo to the device. Positional and head tracking are automatically applied to the camera, overriding the camera's transform. Unity uses head tracking to the VR camera within the reference frame of the camera's local pose when the application starts. If one is using *OVRCameraRig*, that reference frame is defined by the Tracking Space GameObject, which is the parent of the *CenterEyeAnchor GameObject* that has the Camera component. The Unity Game View does not apply lens distortion. The image corresponds to the left eye buffer and uses simple pan-and-scan logic to correct the aspect ratio.

Leap Motion Software Setup

- Download the Orion software and run the installer.
- Right-click on the new *Leap Motion* system tray icon and click Settings. Go to the Troubleshooting tab and select *Recalibrate Device*. We recommend a calibration of 90%.
- On the General tab, check Allow Images. This allows apps to access the infrared video pass through.
- Download the *Unity Core Assets for Leap Motion* to program the Leap Motion in Unity. In Unity, go to File and click New Project. Name your project and click Create Project. Right-click in the Assets window, go to Import Package and left-click Custom Package. Find the *Core Unity package* and import it.
- Core Assets import three folders in the Assets window – the *Plugins* folder and *LeapC* folder which contain all of our API bindings, and the *LeapMotion* folder, which contain all of our Prefabs, Scripts, and Scenes (see Table 8.4).
- Once the Core Assets are set, go to LeapMotion/Core/Prefabs in the Asset window. From there, drag a **LMHeadMountedRig** into the scene. In the hierachy for the LMHeadMountedRig, we have a **LeapHandController**.
- From the LeapMotion/Prefabs/HandModelsNonhuman folder, drag a *CapsuleHand_L* and a *CapsuleHand_R* to the scene's hierarchy window and make them children of the LeapHandController.
- From the LeapMotion/Prefabs/HandModelsPhysical folder, drag a *RigidRoundHand_L* and a *RigidRoundHand_R* to the scene's hierarchy window and make them children of the LeapHandController.
- Locate the HandPool component attached to the LeapHandController. Set the Model Collection value to 4. Then move your two graphics hands and two physics hands from the Hierarchy view to the four empty slots.

Prefab/ Components	**Use Case**
LMHeadMountedRig Prefab	A full VR rig that combines cameras and hand tracking. This prefab is designed to work with Unity's built-in VR Support. To use the LMHeadMountedRig to a scene, we need to remove any existing camera or camera rigs from the scene
LeapHandController Prefab	Queries the Leap Motion service for tracking data and uses it to place hands in the scene. The tracking data from the service is transformed relative to the prefab's position and orientation in the scene. The scripts in the controller manage the hand objects that represent the physical hands detected by the Leap Motion device.
Leap Service Provider Component	The connection point between the Leap Motion service and the rest of the Unity assets. The service provider gets frames and images from the service and provides them other parts of your application.
Hand Pool Component	Manages the representation of the hands in a scene. A single tracked hand can have any number of task-specific Unity game objects associated with it.

Table 8.4: Leap Motion Core Asset Prefabs and their Use Cases.

- On the LeapHandController GameObject you'll see a LeapProvider component. For VR, make sure that "*Is Head Mounted*" is enabled.

8.3 Input Data Preparation

The Mouse Brain Vascular Network's 3D Models created by the IEROM Image Processing Pipeline is loaded into the Unity Editor as an Asset. Unity can read .fbx, .dae (Collada), .3ds, .dxf, .obj, and .skp model files. We decided to use .OBJ file format for this application. OBJ is a geometry definition file format developed by Wavefront Technologies. The OBJ file format is a simple data-format that represents 3D geometry alone — namely, the position of each vertex, the UV position of each texture coordinate vertex, vertex normals, and the faces that make each polygon defined as a list of vertices, and texture vertices. Vertices are stored in a counter-clockwise order by default, making explicit declaration of face normals unnecessary. OBJ coordinates have no units, but OBJ files can contain scale information in a human readable comment line.

User's Movement in Virtual Reality	**3D Space Translation**
Right	Positive X direction
Left	Negative X direction
Up	Positive Y direction
Down	Negative Y direction
Forward	Positive Z direction
Backward	Negative Z direction

We converted the .STL files created by the Image Processing Pipeline into .OBJ files, using a converter tool called *3D Tool.* The converter tool maintains the STL structure and order of the faces, vertices and normals in the model.

8.4 Mouse Brain as a GameObject in Unity

At first, a new 3D project is created in Unity which will open the basic settings for a scene. The basic settings of a scene include a *Main Camera* and a *Directional Light.* The Oculus support is enabled by checking the *Virtual Reality Support* option in project settings (as described in the *Oculus Software Setup* section above.

Camera in the Scene

We replaced the *Main Camera* in the scene hierarchy with a *Leap Motion Prefab* called *LMHeadMountedRig.* It is a combination of camera and hand-tracking, and works well with Unity's built-in VR support. The *LMHeadMountedRig* has three parts namely: *CenterEyeAnchor, LeapSpace,* and *LeapHandController.* The scripts in the LMHeadMountedRig automatically adjust the stereo camera positions to the correct interpupillary distance and automatically compensate for video lag in VR scenes (see Fig.**??**). The default settings are typically correct for the default LMHeadMountedRig as used in a VR scene. We didn't have to change the position or orientation of the Rig for our scene, the Rig is set to have the hands in the field of view. The default *Transform* position is set to **(x=0, y=0, z=0)**, which is the origin of the world space coordinate system (see Fig.**??** and Table 8.4). This means the user's vision is centered at the origin of the 3D space and is looking along the positive *z direction.*

The *CenterEyeAnchor* has the camera component and the *Leap VR Camera Control* component. The camera component has the *Player Settings* of the project included in the *Rendering Path.* So, the camera, like the Leap Motion Controller, also follows the orientation and positional tracking of the Oculus SDK. The *LeapSpace* has the *Leap VR Temporal Warping* component which interpolates the position of the cameras to compensate for differences between the captured Leap Motion frame time and the current Unity update time. The *LeapHandController* has the *Leap Service Provider* component, which is the connection point between the Leap Motion service and the rest of the Unity assets. The service provider gets frames and images from the service and provides them to the other parts of the application. The Leap Service Provider is an important component and the following options are enabled:

- *Is Head Mounted* is enabled for proper hand tracking when the Leap Motion hardware is mounted on an HMD (Oculus Rift).
- *Override Device Type* is checked to enable the use of a specific Leap Motion hardware profile (i.e. Leap Motion Controller)

Directional Light

Directional lights are very useful for creating effects such as sunlight in the scenes. Behaving in many ways like the sun, directional lights can be thought of as distant light sources which exist infinitely far away. A directional light does not have any identifiable source position and so the light object can be placed anywhere in the scene. All objects in the scene are illuminated as if the light is always from the same direction. The distance of the light from the target object is not defined and so the light does not diminish. The *Transform* component of this Gameobject is modified such as the position is set to face the positive *y-axis* (see Fig.**??**).

Mouse Brain as the Target Object

Unity allows importing 3D models by either dragging the model file from the file browser straight into the Unity project window or by copying the 3D model file into the Project's Assets folder. We imported the Unit Mouse Brain Vasculature Model as an .OBJ file into the Assets Folder of the project. The import settings of the model file is selected from the importer's inspector window (see Fig.**??**). Some of the import settings enabled are as follows:

- Default *Scale Factor* of the imported .OBJ model is 1 unit. Unity's physics system expects 1 meter in the game world to be 1 unit in the imported file.
- *Read/Write Enabled* option is checked so the Mesh data is kept in memory due to which a custom script can read and change it.
- *Optimize Mesh* option is enabled, so that Unity determines the order in which triangles are listed in the Mesh.
- *Import Blendshapes* checked, allows Unity to import BlendShapes with the Mesh.
- *Normals* are imported and *Tangents* are calculated.

- *Materials* are imported by default. Material Naming is *by Base Texture Name* which means the name of the diffuse Texture of the imported Material is used to name the Material in Unity. Material is searched *recursively*, which means Unity tries to find existing Materials in all Materials subfolders in all parent folders up to the Assets folder.

Model files placed in the Assets folder of the Unity project are automatically imported and stored as Unity Assets. A model file containing a 3D model, such as a Mouse Brain Vasculature Mesh is imported in the project as a .OBJ file. In the Project window, the primary imported object appears as a model Prefab. Usually, several Mesh objects are referenced by the model Prefab. In our case, the model prefab was split into 8 Mesh Objects, each representing a part of the original model. While importing, Unity breaks up any high-poly model prefabs into sub-models based on the maximum vertex limit of each mesh. The maximum vertex limit for one imported model is 65535.

This imported model is added to the scene as a Gameobject and also the only target object in the scene. The *Transform* parameters of the Gameobject was modified to fit it into the VR camera's/ user's field of view:

- *Scale Factor* is reduced to 0.1 unit as the model is too big to visualize.
- *Rotation* value is unchanged and is by default as **x=0°, y=0°, z=0°.**
- *Position* is changed to **x=12.8, y=-12.8, z=2** from the default values i.e. **x=0, y=0, z=0**. Originally when the model is loaded into the scene, only one corner end of the mesh is visible (see Fig.**??**). This is because the of the pivot point of the Gameobject is at that vertex. A pivot point of the any model/mesh in its local space is the first vertex point (x=0.0, y=0.0, z=0.0). In the world space, the pivot point acts as the center of the object and is placed at the origin of the world space coordinate system (see Fig. **??**).

  The imported OBJ models are created originally from the STL meshes, which sets a starting point or vertex for the meshes. This starting point is retrieved by the OBJ file and is transferred as the pivot point of the imported model. The volumetric dimension of the original model loaded into the scene is $256 \times 256 \times 256$ units. After the reduction of the *Scale Factor* to 0.1 unit, the reference dimensions of the model becomes $25.6 \times 25.6 \times 25.6$. To set the center of the model as the visualization starting point, we shifted the position transform of the model along the x and y directions to get the center of x-y plane first; **(x=12.8, y=-12.8)**. This implies we moved the pivot point 12.8 units to the right and 12.8 units down, so the user sees the center of the plane. The position of the camera is the position of the user in Virtual Reality. At start, the aim is to visualize the whole Mouse Brain model in front of the user at a distance that corresponds to distinct vision. We pushed the model 2 units away from the user to visualize it clearly from the outside (see Fig.**??**).

8.5 Challenges

OBJ File Writer in ITK

Chapter 9

Gesture Controlled Navigation Method

9.1 Overview

In the previous chapter, we studied how to setup the Virtual Reality framework and how to load the Mouse Brain Vasculature meshes in the framework as mere GameObjects. In this chapter, we will design a primary use case for this VR framework implementation. Zooming in and Zooming out in 3D space provides a good way to visualize and analyze a biological mesh. However, virtual reality provides an upper hand advantage of being able to move around in the structure providing a better-detailed understanding of the different parts of the meshes. While studying the architecture of the mouse brain vasculature using different 3D visualization methods, it is always observed that we can't walk into the structure and see the inner parts of the blood vessels. With this Virtual Reality framework, we can easily walk into the structure and visualize any part of the model. Thus, to better study and analyze the morphology of the mouse brain vasculature in Virtual Reality, we designed this *Gesture Controlled Navigation* method to navigate inside out the mouse brain volume. It allows the user to visualize from every angle and distance. This method is developed to navigate inside a single volume model loaded as a GameObject in Unity.

9.2 Required Input

Gesture Controlled Navigation method uses our hand gestures as controllers for navigation. We captured the right-hand gestures as input and transformed those gestures into events. These events, in turn, trigger the control algorithm implemented for camera movements. The Leap Motion Controller tracks the hand gestures and provides input to the Leap Service Provider, which is fed in a script for further implementation. The *Leap Service Provider* is a major component of the *Leap Hand Controller* in the *LMHeadMountedRig* prefab. The Leap Service Provider is the connection point between the Leap Motion service and the rest of the Unity assets. The service provider gets frames and images from the service and provides them to the other parts of the application (see Fig.**??**.)

9.3 Basics of Scripting in Unity

Scripting is an essential ingredient in Unity; it can control objects created in Unity Editor. Although Unity uses an implementation of the standard Mono runtime for scripting, it still has its practices and techniques for accessing the engine from scripts. The behavior of Game Objects is controlled by the Components that are attached to them. However, Unity allows creating Components using scripts extrapolating the provided features. These allow to trigger game events, modify Component properties over time and respond to user inputs. Unity supports two programming languages: C# (pronounced C-sharp), an industry-standard language similar to Java or C++; and UnityScript, a language designed specifically for use with Unity and modeled after JavaScript. A user can create a new script from the Create menu at the top left of the Project panel or by selecting *Assets → Create → C# Script* (or JavaScript) from the main menu. Unity uses *MonoDevelop* script editor, but any editor can be selected from the External Tools panel in *Unity's Preferences*.

A script can be attached by dragging it from the asset window to a GameObject in the hierarchy panel or to the inspector of the GameObject that is currently selected. There is also a Scripts sub menu on the Component menu which will contain all the scripts available in the project, including the one created by the user.

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called *MonoBehaviour*. A class is a kind of blueprint for creating a new Component type that can be attached to GameObjects. Each time a script component is attached to a GameObject, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name you supplied when the file is created. The class name and file name must be the same to enable the script component to be attached to a GameObject.

MonoBehaviour is the base class for all new Unity scripts, the MonoBehaviour reference provides a list of all the functions and events that are available to standard scripts attached to Game Objects. It is the starting reference for any kind of interaction or control over individual objects in the game.

In Unity scripting, there are a number of event functions that get executed in a predetermined order as a script executes (see Table 9.1). Two basic functions exist in any script by default when it is created namely: *Start()* and *Update()*.

- The *Start()* function will be called by Unity before gameplay begins (i.e., before the Update function is called for the first time) and is an ideal place to do any initialization. It is called before the first frame updates only if the script instance is enabled.

- The *Update()* function is the place to put code that will handle the frame update for the GameObject. This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay. To enable the Update function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other GameObjects before any game action takes place.

9.4 Gesture Controller

We created a C# script to detect hand gestures and control the physical movement of the camera, named *Gesture Controller.* This script will act as a new component for the Mouse Brain GameObject in the scene. We added some public variables to examine better the features of the component and some private variables for taking inputs from other Unity Assets for internal implementation. The public variables can be viewed under the *Gesture Controller* component in the inspector window and are useful for debugging while the scene is running (see Fig.**??**):

- *Joystick 3D Activated:* This is a flag controlled by the right-hand gestures. The status of this flag activates or deactivates the camera movement in the 3D space. This flag is unchecked by default and is checked only when the right-hand in front of the Leap Tracker makes a transition from an *Open Palm* to a *Closed Fist.* If the hand remains open, this flag remains unchecked leading to deactivated movement of the camera.

- *LM Right Hand Pos Curr:* This is a 3D vector value indicating the current position of the right-hand in the 3D space relative to the Leap Motion Camera. This value keeps changing with the hand movements and is tracked every frame.

- *LM Right Hand Pos Pivot:* This is a 3D vector value indicating the position of the right-hand at the beginning of the control implementation, relative to the Leap Motion Camera. This value is updated only when the camera movement is active and stores the reference current position of the right-hand after each frame as the pivot.

- *Speed 3D:* This is a 3D vector which stores the difference between the right-hand's current position and pivot position and is tracked once per frame in the *LateUpdate()* function. This value is used to move the camera after it has tracked the position of the hand in the *Update()* function.

- *Speed 3D Scale:* This is an editable floating value which is used to decide the speed of camera's movement. We value is calibrated between 1.0 to 1.5 units to experience a firm, steady camera movement, comforting the user.

The private variables are used to provide input from other GameObjects in the scene, for the computation of gesture control algorithm:

Event Functions	**Execution Order**
Awake()	This function is always called before any Start functions and also just after a prefab is instantiated. (If a GameObject is inactive during start up Awake is not called until it is made active.)
OnEnable()	This function is called just after the object is enabled. This happens when a MonoBehaviour instance is created, such as when a level is loaded or a GameObject with the script component is instantiated.
OnApplicationPause()	This is called at the end of the frame where the pause is detected, effectively between the normal frame updates. One extra frame will be issued after OnApplicationPause is called to allow the game to show graphics that indicate the paused state.
FixedUpdate()	It is often called more frequently than Update. It can be called multiple times per frame, if the frame rate is low and it may not be called between frames at all if the frame rate is high. All physics calculations and updates occur immediately after FixedUpdate. When applying movement calculations inside FixedUpdate, you do not need to multiply your values by Time.deltaTime. This is because FixedUpdate is called on a reliable timer, independent of the frame rate.
LateUpdate()	LateUpdate is called once per frame, after Update has finished. Any calculations that are performed in Update will have completed when LateUpdate begins. A common use for LateUpdate would be a following third-person camera. If you make your character move and turn inside Update, you can perform all camera movement and rotation calculations in LateUpdate. This will ensure that the character has moved completely before the camera tracks its position.
OnDestroy()	This function is called after all frame updates for the last frame of the object's existence (the object might be destroyed in response to Object. Destroy or at the closure of a scene).

Table 9.1: Execution order of some editable Event Functions while scripting in Unity.

- *LM Service Provider:* It is a Leap Service Provider instance for creating a controller in the scene. It provides leap hands and images in the scene.
- *LM Camera:* It is the Leap Motion Camera GameObject in the scene i.e., *LMHeadMountedRig.* The user looks through this camera in the scene and the movement of this camera is controlled through the *Gesture Controller.* In the script, we access the *LMHeadMountedRig* GameObject via a Tag called *LMCamera.*
- *LM Right Hand Detected:* It is a flag to detect if the Leap Motion Controller is tracking the right-hand in the scene in the field of view of the user.
- *LM Right Hand Closed:* It is a flag to detect if the Leap Motion Controller is tracking the right-hand as a fist or a closed-palm.

With the above defined variables we first track the right-hand gestures and then set flags to perform camera movements accordingly. The following steps are performed to compute the gesture change of the right-hand in the scene:

Step 1: In the *Update()* function which is called once per frame, we first read the current frame from the Leap Service Provider.

Step 2: Detect all hands in the current Leap Motion frame and then get the current hand instance.

Step 3: Check performed to see if the current hand is a right-hand.

Step 4: Get the current position of the right-hand in the scene relative to the Leap Motion Camera position, and store it in the 3D vector *LM Right Hand Pos Curr* variable.

Step 5: Store the previous right-hand status which includes the flags for right-hand detection in the scene and right-hand closed.

Step 6: Update the right-hand status with current values. To find out if the right-hand detected in the scene is an open palm or a closed fist, the *Grab Angle* of the hand is measured. If the *Grab Angle* is more than $90°$, the hand is considered to be closed.

Step 7: If previously right-hand was detected with an open palm status and the current right-hand is detected with a closed fist. Then, the camera movement is activated which can be reviewed through the *Joystick 3D Activated* variable in the Gesture Controller Component. The current right-hand position is stored as the pivot in the *LM Right Hand Pos Pivot* variable.

Step 8: The camera movement remains deactivated if the right-hand is detected as an open palm.

Further, after the gesture change of the right-hand is captured for a frame, we perform the camera movement accordingly in the *LateUpdate()* function. In this function, we change the transform position of the Leap Motion Camera only if the *Joystick 3D Activated* flag is checked. The distance to move is retrieved from the *Speed 3D* value. This value is multiplied with the *Time.deltaTime* and *Speed 3D Scale* values. The *Speed 3D Scale* will maintain the steady camera movement and the *Time.deltaTime* will make the movement frame rate independent. The camera moves *Speed 3D* distance per second and not per frame, at a speed defined by *Speed 3D Scale* value. Thus, the camera movement is smooth and steady and is not delayed due to frame loses. In this way, the user can activate the camera movement by the gesture transition and then quickly move inside out of the Mouse Brain Vasculature Model.

Chapter 10

User-Interface for the Virtual Reality Framework

10.1 Overview

The previous chapter serves a major use case of the Virtual Reality framework but navigates within a single Mouse Brain model loaded in the scene. In this section, we extend the capability of this framework by implementing a way to change the model loaded into the scene and be able to visualize and study the Mouse Brain Vasculature in different resolutions. We created a user-interface for performing this switching of data set. This user-interface will allow the user to visualize a section of the Mouse Brain in multiple resolutions, helping to study the biological organ in detail.

10.2 Required Input

The User-Interface for this framework appears as an overlay screen space with UI controls. The screen space is not visible to the user by default or at the start of the scene. It appears only when the left-hand is detected in the scene, and it makes a gesture transition from a *closed fist* to an *open palm*.

To switch to different resolution models, it is required to have the 3D models loaded into the scene as and when called by the user. The *OBJReader* asset will allow loading of 3D models during run-time. We load these models directly from the local computer disk. We converted all the STL meshes created by the Image Processing Pipeline to OBJ models. Thus, we end up having similar directory structure for different resolution OBJ models (see Table**??**.)

10.3 Basics of UI Designing in Unity

The user-interface appears as a overlaid screen space or *canvas* in Unity's language. All UI elements reside inside the Canvas. The Canvas is a Game Object with a Canvas component on it, and all UI elements are children of the Canvas. The Canvas area is shown as a rectangle in the Scene View and uses the EventSystem object to help the Messaging System. UI elements in the Canvas are drawn in the same order they appear in the Hierarchy.

The Canvas has a Render Mode setting which can be used to make it render in screen space or world space.

- *Screen Space - Overlay* render mode places UI elements on the screen rendered on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change size to match this.
- *Screen Space - Camera* is similar to Screen Space - Overlay, but in this render mode the Canvas is placed a given distance in front of a specified Camera. The UI elements are rendered by this camera, which means that the Camera settings affect the appearance of the UI.
- In *World Space* render mode, the Canvas will behave as any other object in the scene. The size of the Canvas can be set manually using its Rect Transform, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be a part of the world. This is also known as a *diegetic interface.*

The *Rect Transform* is a new transform component that is used for all UI elements instead of the regular Transform component. Rect Transforms have position, rotation, and scale just like regular Transforms, but it also has a width and height, used to specify the dimensions of the rectangle.

10.4 Walk-Through the User-Interface Design

We created a canvas with *screen space overlay* rendering mode. We added a set of interactive UI elements such as buttons to the canvas:

- *Resolution Buttons:* One button for each resolution is added to the canvas for switching to that resolution 3D models during run-time.
- *Reset:* This button is used to reset the camera's position at the starting point. While navigating inside a structure, we might get to random locations in the 3D space. So, in any situation, if a user wants to come back to the point from where they started navigating, they can click this button to do so.
- *Exit:* This button is used to exit the user-interface screen space overlay.

Enabling/Disabling the UI Control Panel

We created a component called *UIController* through a C# script for this implementation (see Fig.**??**). This component is attached to the Canvas and has following variables:

- *UI Controller Visible:* A public variable flag that is controlled by the left-hand gestures and is used to show or hide the user-interface Control Panel.
- *LM Service Provider:* A private variable and Leap Service Provider instance to create a controller and supply leap hands as inputs.
- *Control Panel:* Private variable to hold the *canvas* instance. This instance is disabled by default and is enabled only when the *UI Controller Visible* flag is checked. When the *Exit* button is clicked, the canvas is disabled again.

The left-hand gestures are tracked for each frame in the same way as done in the *Gesture Controller* component. However, the interpretation is different. In this case, if the left-hand previously detected was in a *closed fist* situation and current left-hand gesture is detected as an *open palm,* the *UI Controller Visible* flag is checked and the *Control Panel* is enabled.

Loading the 3D Models

We created another component called *Object Manager* for switching and loading different resolution data sets from the local disk space during run-time. This script exploits a unique *Data Mapping* method to select the 3D model to be loaded (explained below). It then implements the *OBJReader* method to load the selected 3D model in the scene during run-time. This component contains variables to display the loaded .OBJ file name and file path. It also shows the material and texture details used in the .mtl file (see Fig.**??**).

Data Mapping

Let's explain this with an example: If a user is visualizing a part of a model in a lower resolution i.e., $300 \times 375 \times 301$ and wants to switch to visualize that section of the model in the next higher resolution i.e., $600 \times 750 \times 602$. First, the user needs to navigate to that section of the model and then open the *UI Control Panel* and select the desired higher resolution. The *Object Manager* component will perform the following steps to select the appropriate 3D Model to be loaded:

Step 1: It will first capture the coordinates of the current position of the user (or LM Camera) inside the lower resolution model, and store this coordinate in a temporary vector3 variable. Let's say the user is at position **x=3.5012, y=21.4562, z=8.9275.**

Step 2: A 3D model in its lower resolution is internally split into 8 parts to map to 8 higher resolution data sets (see Fig.**??**). The 3D Models created by the Image Processing Pipeline are unit volumes labelled with their *unit-x, unit-y, unit-z* coordinates.

Sections of the Lower Resolution	**Corresponding Range of Coordinates in the 3D Model**	**Mapped Higher Resolution File**
`Part_000`	x:$(0 \rightarrow 15.0)$, y:$(0 \rightarrow 18.75)$, z:$(0 \rightarrow 15.5)$	`Vol_0_0_0.Obj`
`Part_001`	x:$(0 \rightarrow 15.0)$, y:$(0 \rightarrow 18.75)$, z:$(15.5 \rightarrow 30.1)$	`Vol_0_0_1.Obj`
`Part_010`	x:$(0 \rightarrow 15.0)$, y:$(18.75 \rightarrow 37.5)$, z:$(0 \rightarrow 15.5)$	`Vol_0_1_0.Obj`
`Part_011`	x:$(0 \rightarrow 15.0)$, y:$(18.75 \rightarrow 37.5)$, z:$(15.5 \rightarrow 30.1)$	`Vol_0_1_1.Obj`
`Part_100`	x:$(15.0 \rightarrow 30.0)$, y:$(0 \rightarrow 18.75)$, z:$(0 \rightarrow 15.5)$	`Vol_1_0_0.Obj`
`Part_101`	x:$(15.0 \rightarrow 30.0)$, y:$(0 \rightarrow 18.75)$, z:$(15.5 \rightarrow 30.1)$	`Vol_1_0_1.Obj`
`Part_110`	x:$(15.0 \rightarrow 30.0)$, y:$(18.75 \rightarrow 37.5)$, z:$(0 \rightarrow 15.5)$	`Vol_1_1_0.Obj`
`Part_111`	x:$(15.0 \rightarrow 30.0)$, y:$(18.75 \rightarrow 37.5)$, z:$(15.5 \rightarrow 30.1)$	`Vol_1_1_1.Obj`

Table 10.1: The 8 split parts of the lower resolution mapped to the 8 unit volume models in the higher resolution.

The lowest resolution i.e., $300 \times 375 \times 301$ has only one 3D model volume i.e., `Vol_0_0_0.Obj`. Split this model into 8 parts means bi-sectioning the model in each direction (see Table 10.1). So the 8 parts created are: `part_000`, `part_001`, `part_010`,..., `part_111`. These parts are mapped to the 8 unit volume models in the next higher resolution i.e., $600 \times 750 \times 602$.

Step 3: As per the Table 10.1, the current position of the LM Camera inside the model lies in the range x:$(0 \rightarrow 15.0)$, y:$(18.75 \rightarrow 37.5)$, z:$(0 \rightarrow 15.5)$, corresponding to `Part_010`. Thus, the higher resolution model to be loaded is `Vol_0_1_0.Obj`.

OBJReader

OBJReader can load 3D models at run-time from local files or web. It requires only a file name or a string for the .Obj file, and automatically loads the model as a GameObjects. It works with most .obj files from a variety of sources, and has .mtl file support for materials and textures. Includes various options such as generate tangents, position/rotate/scale, and combine groups into a single mesh or generate one mesh per group, using submeshes or not. Plus it's fast...a 4MB .obj file will typically generate a mesh in less than 0.5 second, depending on CPU speed.

When we import the OBJReader asset to use in our project, a GameObject named *ObjManager* is added to the hierarchy window. This GameObject has *Obj Reader* component which has the following public variables (see Fig.**??**) :

- *MaxPoints:* Maximum number of vertices that the ObjReader will accept per group. The .obj file can exceed this limit by using multiple groups, as long as each group is under the vertex limit. The MaxPoints is clamped to 65,534 as the maximum regardless of what is set by the user. Since, that is the most Unity will accept for one mesh.

- *CombineMultipleGroups:* If checked, will combine all groups found per .obj file into one mesh as a single GameObject. Otherwise each group will result in a separate GameObject. Combining multiple groups into one object will fail if it causes the number of vertices to exceed MaxPoints.

- *UseSubmeshesWhenCombining:* It is only used if *CombineMultipleGroups* is checked. It makes each group into a submesh on a single GameObject. If it's not checked, then using CombineMultipleGroups will result in a single mesh with a single material, regardless of what an associated .mtl file might contain.

- *UseFileNameAsObjectName:* It will cause any generated GameObjects to be named with the actual file name of the .obj file (without extension). If unchecked, the GameObjects will be named by group names that are supplied in the .obj file.

- *ComputeTangents:* If checked, will cause tangents to be calculated for each object.

- *UseSuppliedNormals:* It will cause ObjReader to use any normals that may be supplied by the .obj file. Otherwise, normals are calculated instead (using Unity's RecalculateNormals function).

- *OverrideDiffuse:* If checked, will discard any diffuse color supplied by MTL files in favor of the Main Color used on any materials that you supply.

- *OverrideSpecular:* If checked, will discard any specular color supplied by MTL files in favor of the Specular Color used on any materials that you supply.

- *OverrideAmbient:* If checked, will discard any ambient color supplied by MTL files in favor of the Emissive Color used on any materials that you supply.

- *SuppressWarnings:* It will prevent any standard ObjReader warnings from being printed.

- *UseMTLFallback:* If checked, will use the standard material in such case as an MTL file is specified in the OBJ file, but is missing. If useMTLFallback is not checked, then missing MTL files will generate an error.

- *AutoCenterOnOrigin:* If checked, will physically move the vertices so the object is centered on (0, 0, 0).

- *ScaleFactor:* It is a Vector3 that scales the resulting object meshes by the specified amount on the x, y, and z axes.

- *ObjRotation:* It will rotate the resulting object meshes around the specified x, y, and z axes.

- *ObjPosition:* It does the same thing for the position of object meshes. If the .obj file is off-center, you can compensate by using ObjPosition to move the mesh to the desired location.

The Object Manager uses the *OBJReader* method to load the file `Vol_0_1_0.Obj` into the scene. It will first remove the 3D model loaded in the scene and then simply pass this file name to the *OBJReader.*

Reset Camera

This is the simplest component created to Reset the Leap Motion Camera Position to the starting point from where the user started navigating. This component is attached to the *Reset* button in the *UI Control Panel.* It implements a function which changes the position transform of the Leap Motion Camera i.e., *LMHeadMountedRig* back to origin of the 3D space i.e., $x=0.0$, $y=0.0$, $z=0.0$.

10.5 Challenges

Importing all the 3D models in the Project Window

To switch to different resolution models, it is important that we have those models loaded in the Project window like imported Assets. However, the 3D models created by the Image Processing Pipeline for the whole mouse brain structure is a huge amount of data. Unity cannot accommodate such a huge number of models. Thus, we loaded the models into the scene directly without importing them. We loaded the models from the local disk of the computer where all the 3D models of all the resolutions are stored. The loading happens during run-time using an imported *OBJReader* Asset. *OBJReader* allows to load 3D models in a scene at run-time from local files or the web.

Loading the huge Mouse Brain models into the scene

The *OBJReader* can load the 3D models into the scene during run-time. However, it has a limitation to the number of vertices to be loaded in one model, i.e., 65000. All the 3D models created for the Mouse Brain Vasculature has a lot more than 65000 vertices. This poses a big problem of improper data input for the *OBJReader.* This issue is resolved by splitting the 3D models into separate sections, each section having a maximum of 65000 vertices. Usually, Unity performs this splitting when a 3D model is imported in the project window. But since we are directly loading the models from the local disk space at run-time, Unity's mesh splitting is overridden. We performed the splitting of OBJ models using a tool called *Blender.* It is a time-taking, complicated and manual process of splitting in *Blender*, but this is the only solution feasible in the available time.

Part V

Conclusion and Future Work

Chapter 11

Conclusion and Future Work

11.1 Image Processing Pipeline Additions

11.2 Web-Based Framework Improvements

11.3 Virtual Reality Framework Improvements

Dataset Creation

Data Mapping for Higher Resolutions

11.4 Conclusion

Part VI

Appendices

Appendix A

Bibliography

[1] James Ahrens et al. *36-ParaView: An End-User Tool for Large-Data Visualization.* 2005.

[2] Alessandro Bria, Giulio Iannello, and Hanchuan Peng. “An open-source VAA3D plugin for real-time 3D visualization of terabyte-sized volumetric images”. In: *Biomedical Imaging (ISBI), 2015 IEEE 12th International Symposium on*. IEEE. 2015, pp. 520–523.

[3] Yoonsuck Choe et al. “Complete submicrometer scans of mouse brain microstructure: neurons and vasculatures”. In: *Neuroscience Meeting Planner, Chicago, IL: Society for Neuroscience*. 2009.

[4] Yoonsuck Choe et al. “Multiscale imaging, analysis, and integration of mouse brain networks”. In: *Neuroscience Meeting Planner*. Society for Neuroscience San Diego, CA. 2010.

[5] Yoonsuck Choe et al. “Specimen preparation, imaging, and analysis protocols for knife-edge scanning microscopy”. In: *Journal of visualized experiments: JoVE* 58 (2011).

[6] Ji Ryang Chung et al. “Multiscale exploration of mouse brain microstructures using the knife-edge scanning microscope brain atlas”. In: *Frontiers in neuroinformatics* 5 (2011).

[7] *dat.GUI A lightweight graphical user interface for changing variables in JavaScript.* http://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage. Accessed: 2016-06-11.

[8] Ciro Donalek et al. “Immersive and collaborative data visualization using virtual reality platforms”. In: *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE. 2014, pp. 609–614.

[9] David A Gutman et al. “Web based tools for visualizing imaging data and development of XNATView, a zero footprint image viewer.” In: *Frontiers in neuroinformatics* 8 (2013), pp. 53–53.

[10] Daniel Haehn et al. “Neuroimaging in the browser using the x toolkit”. In: *Frontiers in Neuroinformatics* 101 (2014).

[11] G Allan Johnson et al. “Waxholm space: an image-based reference for coordinating mouse brain research”. In: *Neuroimage* 53.2 (2010), pp. 365–372.

[12] Jaerock Kwon et al. “Automated lateral sectioning for knife-edge scanning microscopy”. In: *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*. IEEE. 2008, pp. 1371–1374.

[13] Ed S Lein et al. “Genome-wide atlas of gene expression in the adult mouse brain”. In: *Nature* 445.7124 (2007), pp. 168–176.

[14] David Lesage et al. “A review of 3D vessel lumen segmentation techniques: Models, features and extraction schemes”. In: *Medical image analysis* 13.6 (2009), pp. 819–845.

[15] Fuhui Long, Jianlong Zhou, and Hanchuan Peng. “Visualization and Analysis of 3D Microscopic Images”. In: *PLOS Computational Biology* 8 (6 2012). DOI: 10.1371/journal.pcbi.1002519.

[16] Allan MacKenzie-Graham et al. “The informatics of a C57BL/6J mouse brain atlas”. In: *Neuroinformatics* 1.4 (2003), pp. 397–410.

[17] David Mayerich, L Abbott, and B McCormick. “Knife-edge scanning microscopy for imaging and reconstruction of three-dimensional anatomical structures of the mouse brain”. In: *Journal of microscopy* 231.1 (2008), pp. 134–143.

[18] David Mayerich, Bruce H McCormick, and John Keyser. “Noise and artifact removal in knife-edge scanning microscopy”. In: *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on*. IEEE. 2007, pp. 556–559.

[19] Bruce H McCormick et al. “Construction of anatomically correct models of mouse brain networks”. In: *Neurocomputing* 58 (2004), pp. 379–386.

[20] Shawn Mikula et al. “Internet-enabled high-resolution brain mapping and virtual microscopy”. In: *Neuroimage* 35.1 (2007), pp. 9–15.

[21] Xing Ming et al. “Rapid reconstruction of 3D neuronal morphology from light microscopy images with augmented rayburst sampling”. In: *PloS one* 8.12 (2013), e84557.

[22] *Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine.* http://nodejs.org. Accessed: 2016-06-11.

[23] Hanchuan Peng et al. “Extensible visualization and analysis for multidimensional images using Vaa3D”. In: *Nature protocols* 9.1 (2014), pp. 193–208.

[24] Shruthi Raghavan and Jaerock Kwon. “Fully Automated Image Preprocessing for Feature Extraction from Knife-Edge Scanning Microscopy Image Stacks”. In: ().

[25] F. Ritter et al. “Medical Image Analysis”. In: *IEEE Pulse* 2.6 (Nov. 2011), pp. 60–70. ISSN: 2154-2287. DOI: 10.1109/MPUL.2011.942929.

[26] Glenn D Rosen et al. “The mouse brain library@ www. mbl. org”. In: *Int Mouse Genome Conference*. Vol. 14. 2000, p. 166.

[27] Tarek Sherif et al. “BrainBrowser: distributed, web-based neurological data visualization.” In: *Frontiers in neuroinformatics* 8 (2013).

[28] *XTK The X Toolkit: WebGL^{TM} for Scientific Visualization.* http://github.com/xtk/X. Accessed: 2016-06-11.