A System for Scalable 3D Visualization and Editing of Connectomic Data

by

Brett M. Warne

B.S. Computer Science and Engineering
Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

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

Certified by .....
H. Sebastian Seung
Associate Professor
Thesis Supervisor

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

A System for Scalable 3D Visualization and Editing of Connectomic Data

by

Brett M. Warne

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

Abstract

The new field of connectomics is using technological advances in microscopy and neural computation to form a detailed understanding of structure and connectivity of neurons. Using the vast amounts of imagery generated by light and electron microscopes, connectomic analysis segments the image data to define 3D regions, forming neural-networks called connectomes. Yet as the dimensions of these volumes grow from hundreds to thousands of pixels or more, connectomics is pushing the computational limits of what can be interactively displayed and manipulated in a 3D environment. The computational cost of rendering in 3D is compounded by the vast size and number of segmented regions that can be formed from segmentation analysis. As a result, most neural data sets are too large and complex to be handled by conventional hardware using standard rendering techniques. This thesis describes a scalable system for visualizing large connectomic data using multiple resolution meshes for performance while providing focused voxel rendering when editing for precision. After pre-processing a given set of data, users of the system are able to visualize neural data in real-time while having the ability to make detailed adjustments at the single voxel scale. The design and implementation of the system are discussed and evaluated.

Thesis Supervisor: H. Sebastian Seung
Title: Associate Professor

Acknowledgments

This paper marks the end to over a year of hard work. A year of arguing with cross-platform build systems, late night multi-threaded debugging, and mind-numbing questions about how to code the impossible.

Of course, I must credit Sebastian Seung for giving me the opportunity to work on this project. As overwhelming and demanding as the goals for this project felt, it pushed me to do the best work I can do. Credit is also due to Srinivas Turga, Viren Jain, and Daniel Berger for invaluable feedback as well as a laundry list of feature requests that always pushed me to do more. And while sometimes I wondered what I was doing in a lab full of computational neuroscientists, I gained a newfound appreciation for connectomics, and I was honored that I could bring my talents of software design and development to the group.

But if I ever talk about software design, I need to give credit to Ricarose Roque and Daniel Wendel. I learned more about quality software design working with them in Eric Klopfer's lab developing StarLogoTNG than in any class I could ever take. Their passion for clean, modular, and functional design has since been an inspiration for all of my work. And yet the remind me that no matter how daunting code may be, never let it get in the way of your humor and spirit. I wish them all the best.

Even more so than just a year of work, this paper marks the end of five years at MIT. Survival was only possible with the support of so many caring friends. The late nights avoiding work or rare adventures off campus when we could get a car were much needed breaks from tooling. So while we may call each other by our Athena handles (with the exception of "trash-face"), we never let the institute take us over completely.

Finally I have to thank the parents and sister whose love and support never dwindled. I'm sure my over-exaggerated phone conversations have lead them to believe that MIT is the most unhappy place on earth. And while that's not exactly correct, it was made a lot better knowing I always had a family that never stopped believing in me.

Contents

List of Figures

Chapter 1

Introduction

Recent advances in neural imaging have enabled neuroscientists to gather data at previously unobtainable dimensions. Smaller resolutions and large extents of captured data facilitate a more global perspective into neural structure. Rather than focusing on specific interaction between neuron pairs, neuroscience now has the ability to consider the neuron activity as more generalized network of interaction. Mapping this activity into a connectivity network, called a connectome, could lead to an understanding of brain functionality that is currently unexplainable by local neuron interaction. The relatively new field of connectomic research has established itself to formulate these connectivity networks through mass neural connectivity data generation and analysis.

A combination of new technology being developed at the Massachusetts Institute of Technology, Harvard, and the Max Plank Institute for Medical Research in Heidelberg, Germany has been critical to the significant growth that connectomics has experienced recently. Heidelberg and Harvard are leading advances in microscopy to allow for a new level of detail in cellular imaging, resulting in dramatic increases in the amount of image data that can be gathered. Furthermore, development of computation ability at MIT has greatly aided in the analysis of this image data resulting in the generation of vast amounts of corresponding connectomic data.

1.1 Motivations for Scalability

The increased generation of connectomic data, as facilitated by technological advances in the capture of raw image data also results in significant performance scalability problems. The relationship between dimension and volume dictates that a linear increase in image capture resolution results in a cubic increase in volume data. As a result, even modest

improvements in microscopy can dramatically increase the number of voxels in volume data to be visualized. Currently, standard image volumes have dimensions of thousands of pixels. If this is increased just a single order of magnitude, the volume data will grow from billions to trillions of voxels. A system that aims to meet the challenge of interactively displaying such data sets must be built on principles of scalability if it intends on maintaining usefulness.

1.2 Motivations for 3D Editing

Visualizing connectomic data in 3D is a common technique used during the analysis process. After the initial raw image data is acquired, the data is segmented into regions that correspond to objects in the volume. This segmentation data is used to construct 3D models that represent these objects. Drawing these objects in a 3D space provides a useful relative spacial context and intuitive navigation means into the connectomic data. Orbiting around an object in space offers numerous perspectives into the inherently 3D structure of segmented volume data. Many of these interactions that are immediately apparent as a 3D model are quite difficult to visualize when only using layers of 2D slices.

In addition to simply visualizing the results of segmented connectomic data, users require the ability to create and modify segmentations as well. While ideally all raw image data would be segmented into various objects perfectly by a segmentation algorithm, current analysis techniques occasionally make mistakes when labeling image data. Traditional segmentation editing uses an interface similar to many drawing programs to facilitate changes in values that make up a segmentation within the volume. This can be done using a brush tool to paint in specific pixels or the use of splines to surround a group of pixels. Yet either interface is inherently limited to a requirement of working on a 2D plane. Most significantly, this means the user is forced to flip between slices to gain an understanding of the true 3D structure of the segment. This is less than ideal when trying to understand the connectivity of these generally topographically complex structures.

Editing in 3D offers the spacial context benefits of 3D visualization to quickly understand how subtle segmentation modifications are changing the connectivity of the data. Direct user interaction with a 3D object allows the user to get instant feedback on how segmentation edits affect the topography of segmented regions. This allows a user to quickly and easily correct data that have been miss-labeled during the segmentation process.

1.3 Thesis Outline

This thesis proposes a new scalable system for working with connectomic data in 3D. This includes an architecture for handling vast quantities of data to facilitate interactive visualization of objects produced from segmentation as well as a novel approach for editing segmentation data in 3D. Chapter 2 describes the background of connectomic data and discusses work related to 3D connectomic visualization and traditional segmentation editing. The general design of the new system proposed by this thesis is outlined in Chapter 3. Chapter 4 discusses the system components and their unique features that contribute to the performance and functionality of the system. The application of this thesis is discussed in Chapter 5, which describes the use of the system in a full application. Chapter 6 concludes by describing improvements that could be considered for the future and reviews the major contributions of the system.

Chapter 2

Background

This chapter provides a background into current techniques for connectomic visualization and editing. Section 2.1 details the generation and analysis of segmentation data and section 2.2 outlines software related to the proposed system.

2.1 Data Generation and Analysis

In order to construct connectomic networks, data must first be generated from a tissue source and then analyzed to form a segmentation. A data generation starts with a section of brain tissue that is carefully sliced to a few dozen nanometers in thickness. These cross-section slices are scanned via electron microscopy to form layered sets of neural data. Various stains or fluorescent markers may be added to the tissue, causing different features to become pigmented and making them easier to differentiate. These unique versions of image data are called 'channels'. Multiple channels allow for greater clarity and perspective into the physical makeup of the image slice. A significant challenge, however, is the necessary care needed to achieve high-quality image data through careful slicing and scanning so as to avoid creating imaging artifacts. Poor cuts can distort the tissue from its original shape and bad scanning can result in noise in the data in the form of blurry or obscured data. Any of these issues can lead to significant segmentation errors during image analysis.

In the analysis stage, the data that was gathered from tissue slices are grouped into regions based on evidence of boundaries in the raw image data. These boundaries are segmented apart either by hand or complex machine-learning algorithms to construct 3D objects that are contained within the initial tissue sample. However, the recent exponential growth of volume data is resulting in a near total reliance on software to perform the task of segmentation. Yet the limitations of electron microscopy resolution and the possible

addition of artifacts, as described previously, can result in distorted and ambiguous data. This data is more difficult to analyze and is more likely to cause mistakes during the automated segmentation process. The three major segmentation errors produced during segmentation are:

- **Splits.** A single object in the channel data is incorrectly identified as multiple objects. This may occur when noise in the channel data on a single object is incorrectly interpreted as a boundary between multiple objects.

- **Merges.** Multiple segment objects are mislabeled as a single object. This commonly occurs when image data containing the boundaries of objects is either noisy or of low quality, making it difficult to determine where one object ends and another begins.

- **Misses.** An object is entirely unrecognized during segmentation. Unlike a merge, where one object is assimilated into another, noise in the original image data causes an object to be ignored completely and it is left as an empty space in the final segmentation.

Until segmentation algorithms improve, the output will need human intervention to recognize these errors and perform corrections. This is the primary motivation for segmentation editing as discussed in section 1.2.

2.2 Related Work

The use of software systems to visualize and edit data is ubiquitous to the connectomic analysis process. This is a result of the exponential growth in data that would otherwise be impossible to manage without computational means. This section outlines the principle features and drawback of related connectomic visualization and editing software systems.

2.2.1 Validate

Validate is a tool developed at the Massachusetts Institute of Technology to allow previously traced 2D segment object contours to be constructed into 3D models. This tool allows for simple merging of previously of traced object contours to form unified structures. Validate is primarily used as a merging tool to stitch together pieces of segment results from image

data that may have been segmented multiple times with varying parameters. The resulting segmented regions are individually colored to allow for easy 3D visualization of their connectivity.

Yet there are many feature and performance limitations to this tool. Contours can only be merged into larger structures and cannot otherwise be split or modified without having to refer to other software. Validate also suffers numerous performance issues. The primary causes for this performance limitation is the speed of the language the tool is built on and the limited use of pre-processing. Validate uses the MATLAB scripting language which, like any interpreted language, has a significant performance overhead cost in comparison to a complied language. Furthermore, Validate does not make effective use of preprocessing to provide interactivity. The 3D object representations of segmented regions are generated as needed, but not retained between viewing other regions. This means each time a different object is rendered, the 3D representation must be reconstructed. This causing significant performance reductions when viewing large structures.

In contrast, the proposed system contains editing ability to allow users to split, merge, and otherwise modify segmented regions. Significantly, this functionality is enough to allow a user to modify a segmentation to any shape. Also, the system makes extensive use of pre-processing to allow segmented regions to be quickly loaded and displayed for interactive 3D visualization.

2.2.2 Reconstruct

Developed at Boston University, Reconstruct is a popular connectomic analysis tool because it is freely available and contains much of the basic functionality needed to generate 3D models of layered cross-sections [4]. Reconstruct uses a point series representation of object segmentation rather than the traditional segmentation image data. This means that a segment is defined by a collection of ordered points. A group of these tracings can be converted to a 3D representation which can be viewed interactively or exported to allow for high-resolution rendering in other programs.

Yet the 3D functionality of Reconstruct is focused on exporting generated objects more than a tool for interactive analysis. To view objects in 3D, segment objects are added to the "scene" collection to cause a meshed version of the object to become visible. The meshes that make up the scene may be viewed in Reconstruct or exported to common formats, but

may not be reused in Reconstruct again [5]. This means that every use of the Reconstruct system requires that the meshes be regenerated if they are to be viewed in the 3D component of the system. Furthermore, the reliance on point series data to generate meshes causes the meshing scheme to be highly dependent on point order. This means small irregularities in tracings can lead to severely unexpected results. Section 3.2.2 further describes the use of point series data for mesh generation.

The system proposed, however, uses a robust voxel based volume data representation. Similarly to pixels in many image editing programs, voxels in the 3D space can be set to various values in any order to define a segmented region. Furthermore, meshes generated by the system are stored to disk in a format that can be reused. This allows users to quickly visualize their data each time the system is used after taking the time to build all of the meshes once.

2.2.3 Amira

In contract to Validate and Reconstruct, the Amira software system developed by Visage Imaging is specifically advertised as a scalable visualization system developed for use with large, complex data sets. In addition to rendering large network structures it also provides simple controls for interactively modifying orientation and visualization properties of the data. Amira can generate large meshes constructed from many segment objects and offers a wide variety of viewing options to creatively display 3D data. Unfortunately, Amira is only limited to visualization and lacks any segmentation editing capabilities. Furthermore, while it is able render large, complex, segmentation data, user feedback has indicated this comes at a severe performance cost. This corresponds to significant load times as well as poor frame rates when viewing large data sets. These penalties on interactivity severely limit the usability of the system.

The system described in this paper aims to achieve such scalable rendering without the noticeable degradation in interactivity that Amira suffers. Through sophisticated hierarchical culling methods, the system bounds the data that is loaded and rendered to only visible or partially visible regions. This cuts down on unnecessary processing and dramatically improves interactive performance.

Chapter 3

System Design

This section discusses the design of the proposed system. Section 3.1 sets forth the major system goals, section 3.2 discusses primary considerations of how these goals can be accomplished, and section 3.3 outlines the structure of the final proposed system.

3.1 System Goals

The use of 3D visualization with connectomic data is a common tool for reviewing the results of the segmentation process. Section 3.1.1 describes the 3D visualization aspects of the related system described in section 2.2 that the system aims to incorporate. The novel scalability and 3D editing goals are detailed in section 3.1.2.

3.1.1 Hybrid 3D Visualization

Many of the desired 3D visualization capabilities of the system can be currently found individually in related software systems. But as described in section 2.2, many systems are unable to deal with the interactive performance demands of 3D visualization with the growing segmentation data sets. The proposed system aims to hybrid much of the 3D visualization functionality of related systems within a scalable architecture. The key 3D visualization goals of the system include:

- **3D object generation.** The internal format that the system maintains segmentation data in must facilitate easy extraction of 3D object representations of segmented regions. This process must be fast and efficient to support extracting 3D objects from all segmented regions in all segmentation data.

- **Fast 3D object loading.** During the 3D visualization process, 3D objects must be able to be loaded quickly without significant overhead. As the size of segmentations grows, so does the number of unique segments found and it is critical that the 3D objects associated with each of these segments can be quickly loaded into the 3D space.

- **Interactive 3D visualization performance.** Rendering each unique 3D segment object quickly to achieve interactive performance is critical to the user experience during 3D visualization. The system must be able to manage displaying hundreds to thousands of 3D segment objects simultaneously, without significant degradation to rendering frame rates.

Each of these goals requires processing and managing substantial amounts of data. To integrate all three goals requires a specialized scheme that can scale as the segmentation data grows.

3.1.2 3D Editing

As described in section 1.2, there is significant motivation for 3D segmentation editing. The proposed system aims to allow users to interactively manipulate 3D segment objects in a manner that is integrated with the 3D visualization and provides a unified user experience. The capability to perform editing actions in a 3D space is a feature not found in any other related system. Breaking away from the standard 2D editing interface allows a more natural and intuitive means of interaction when working with segment objects that are inherently 3D themselves. Specifically this includes the ability to perform the following:

- **Precise segmentation data editing.** Given a 3D segment object, a user must have the ability to carefully augment or remove voxels in the segmentation object that are identified with the object. These topographical modifications allow the user to sculpt the object to achieve segmentation corrections.

- **Split and merge functionality.** It is also important to give users the ability to split and merge segments in a 3D environment as well. Splitting depends on the previously described editing functionality to allow the user to topographically disconnect two or more regions of segmentation data labeled as belonging to a single segment and

finally allow the user to relabel one of those regions as a different segment. The merging process involves selecting two or more unique 3D segment objects and easily relabeling them such that all belong to a single segment.

- **Export edited 3D segmentations.** Modifications performed in 3D must be exportable to the common volume data format used in other segmentation systems. This enables a pipeline of use such that the proposed system can be used for viewing and editing data that can be processed by another system.

3.2 Considerations

This section discusses a number of design schemes that could be used to accomplish the previously defined system goals. The major considerations include how segmentation data are to be represented, the representation of 3D segment objects, and the update propagation model between the two.

3.2.1 3D Object Representation

In order to meet 3D visualization performance and 3D editing goals, considerable care must be taken when designing the representation of 3D segment objects. The two common methods for effectively rendering 3D objects are the use of voxel and mesh representations.

- **Voxel.** Using the inherent structure of volume image data, voxel rendering represents each voxel as a rectangular box in 3D space. This allows a single segmented object to be displayed simply by drawing all the voxels that correspond to a specific segmented region in the volume. Furthermore, displaying voxels gives an intuitive grasp into the original segmentation volume data as pixels that are familiar from image slices correspond directly to a voxel in space. This correlation can also be followed to the editing processes. Just as pixels are added and removed in common segmentation editing programs in 2D so are voxels in a 3D space.

  The primary drawbacks of a voxel representation, however, is the cost of rendering. Drawing every voxel in the region wastes considerable resources when generally the only relevant portions visually are on the surface of the object. The problem can be alleviated by filtering for the subset of surface voxels that are visible before rendering, although this degrades loading performance. Yet even when only rendering

boundary voxels, segment objects can have extremely large surfaces and the number of voxels that must be drawn every frame to cover this surface reduces performance considerably.

- **Mesh.** A mesh representation of a 3D object offers considerable performance benefits over a voxel representation. As a collection of triangles that forms a surface, a mesh naturally avoids time wasted on rendering the internal regions of a 3D object. Furthermore, unlike the regular structure of voxels, the set of triangles that form a mesh can be irregularly sized. This means that large uniform portions of the surface area of a mesh can be approximated by a few large triangles which load and render quickly, while detailed portions can be preserved by many smaller triangles. The level of detail preserved is a parameter of the mesh generation process, and allows for user defined selection of performance or detail preservation.

  But unlike a voxel representation, the approximated and simplified mesh representation abstracts the detailed structure of the original segmentation volume data. Unless a mesh is dense enough to detail every voxel, causing it to lose much of its performance advantage over a voxel representation, it is very difficult to perform precise 3D editing. Furthermore, the mesh structure itself is also difficult to interactively change. Since a mesh is a surface of connected triangles, local modifications that insert or remove triangles from the mesh require care in ensuring connectivity of the resulting surface.

The system proposed in this paper makes use of both voxel and mesh rendering depending on the intentions of the user to take advantage of the benefits of each representation. During general visualization, the system displays all data using meshes to quickly load and display 3D segment objects. When a specific segment has been specified to be edited, the system takes the time to generate and render a voxel representation to make use of the fine editing granularity of voxels. Unlike meshes that are build during pre-processing, voxel representations are constructed as need, due to the large memory requirements of storing such a detailed representation for all segment objects. The fast performance of meshes for general 3D visualization and the detail of voxels for editing optimizes the system functionality depending on visualization or editing intentions of the user.

3.2.2 Segmentation Representation

To meet the visualization and editing goals of the system, the internal representation of segmentation data must support fast extraction of both voxel and mesh based 3D objects and allow segmentation data to be easily modified. There are three common methods of defining segmented regions: regularly structured volume, point series, and point set.

- **Structured voxels.** Representing a segmentation as a regularly structured volume of voxels is the easiest and most natural extension to 2D image slices. Like pixels, data is stored at discrete coordinates called voxels in a 3D space.

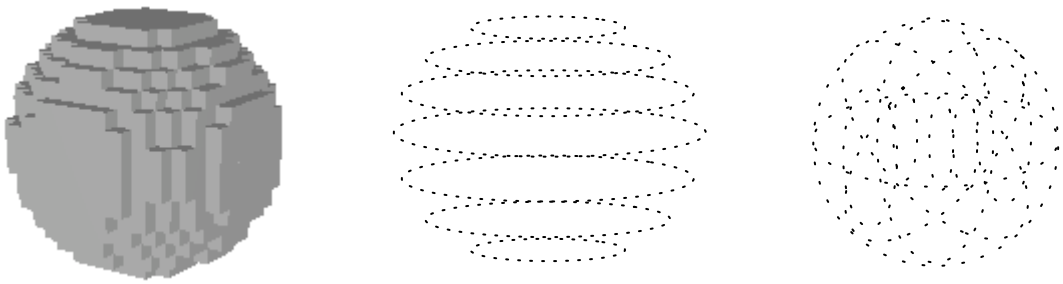
  This simple format allows for voxel objects to be extracted by loading voxels that are associated to a specific value. The standard technique for mesh generation from volume data is commonly performed using the "Marching Cubes" algorithm. This algorithm process the data in a scan-line manner to linearly-interpolate triangle vertices that are formed along a region in the volume that share similar values [10]. The resulting surface tessellation is a usable mesh approximation to a segmented region.

- **Point series.** A point series representation uses a list of ordered points on a 2D plane to create a connected outline of an object. Each connected series defines a 2D region of space, that when used with multiple sets on parallel spatial planes result in a 3D volume. This allows the point series to be extremely efficient in requiring only a few points to coarsely label very large regions.

  The drawback of this efficiency is the challenge of extracting contained volume data needed for voxel representations of 3D objects. The traditional means of rasterizing polygons to pixels is to use a scan line processes to fill in pixels along a line contained within an enclosed region. This highly repetitive process requires great care in ensuring irregularly shaped regions are properly filled. To extract meshes from point series, these points can be used to generate Boissonnat surfaces by using a 3D Delaunay triangulation to tesselate the points [3]. These surfaces depend on an ordered series of data, and can have significant irregularities if the point data is not uniform. Trace data that loops, self-intersects, has irregular ordering, or many other subtle errors can drastically affect the meshing if not causing it to fail entirely.

- **Point set.** A final representation is a set of points used to define the outer surface of a 3D segmented region. While this gives a similar appearance to point series data, the point set is unordered and points are not required to lie on a similar plane. This gives great liberty when defining data, as points can be indiscriminately added in a space to define the boundaries of a 3D region.

  Yet this also makes voxel extraction extremely difficult as there are no definite boundary points to indicate how a scanline through the region could fill in points. A complex way of gaining boundary points is to first generate a mesh and then perform the scanline process using lines that intersect with the tessellated surface. A popular technique to extract meshes from point set data is the Power Crust algorithm. The Power Crust is a mesh generation algorithm for surface reconstruction that can produce such a polyhedral solid [1]. Yet this is an expensive algorithm that requires sufficient points so as to define separated regions to avoid undesired surface connections. As a result, both voxel and mesh extraction are complex and costly processes when using a point set representation.



(a) Structured voxels (b) Point series (c) Point set

Figure 3-1: Example of a spherical segmented region defined by the various segmentation representations.

Due to the simplicity and effectiveness of a volume representation, the proposed system maintains all segmentation data as a structured volume of voxels. Both voxel and mesh representation of 3D objects can quickly and easily be formed to satisfy performance goals while a voxel representation also allows for easy import of segmentation volume data and export of segmentation edits as they all have common regular structure.

3.2.3 Update Propagation Model

Maintaining editable segmentation volume data with the intention of forming both voxel and mesh based 3D segment objects creates a challenge for ensuring that modifications are consistently reflected in both representations. Rather than dealing with the complexities of converting between representations, the system uses the segmentation volume as the source reference data for all updates and modifications. Any changes to the source data are then reflected in the derived voxel and mesh representations through appropriate updates. With the simple voxel correlation between segmentation volume data and 3D segment voxel objects, local modfications can be quickly performed to reflect changes in the source data. Due to the complexities of meshes as described in section 3.2.1, local mesh updates are not an efficient option and entire regions are meshed even if only a few voxels in the segmentation data are changed. As a result, it is more effective to update meshes less frequently, specifically after all changes are made.

The system implements these ideas as defined separate user modes. The 'Navigation Mode' prevents any modification to the segmentation data, ensuring that the render effective meshes can be displayed and are up-to-date. While in 'Editing Mode', 3D segment objects that have been selected for editing are displayed in the more rendering intensive voxel representation, but are able to be quickly updated to reflect local modifications to the segmentation volume data. The system notes which values were modified in what regions of the data volume so that when returning from editing, the system can bound the costly meshing process to only specific regions that have been modified. After these meshes have been updated, the system is ready to use the the newly formed meshes to quickly display 3D segment objects.

3.3 System Outline

To achieve the specific design goals proposed using the 3D object and segmentation representations described under the discussed update propagation model, the system is architected into five major components.

- **System.** The *system* component facilitates communication and state managing to ensure synchronicity between components.

- **Segment.** While the shape of a 3D segment object is defined by data in the segmentation volume, the *segment* component serves to manage the abstract representations of the data. This abstraction allows the segmentation data to be associated in more complex ways, allowing for greater flexibility and performance during visualization and editing.

- **Mesh.** The *mesh* component of the system manages the construction and rendering of the mesh representation of 3D segment objects. It uses the *volume* component to extract data to reconstruct into a surface and the *segment* component for access to the segment data properties needed for rendering.

- **Volume.** The *volume* component manages the segmentation representation. It serves as the interface for the other components to access, manipulate, and render segmentation volume data. The textitvolume component depends on the *mesh* component to render mesh versions segment data, the *segment* component to for access to segment data properties, and the *system* component to perform updates.

- **View.** The *view* component controls the user interface and rendering calls during visualization and editing. The *view* uses the structure of the *volume* component to efficiently render segmentations and the *system* component to synchronize screen redraws so as to reflect changes that are relevant to the 3D visualization.

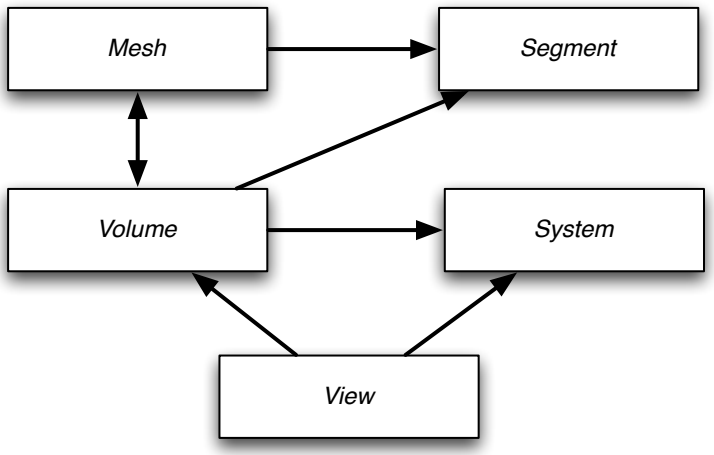


Figure 3-2: Component dependency within the proposed system.

Chapter 4

System Components

This chapter provides a detailed description of the function and design of the major components of the system. Section 4.1 details the *volume* component used to manage access to segmentation data which provides the structure to perform efficient hierarchical rendering of 3D objects. Section 4.2 details how regions of segmentation data are abstracted into Segment objects. The mesh generation and management is discussed in the *mesh* component in section 4.3. The *view* component that manages the 3D user interface and rendering model is detailed in section 4.4. Finally section 4.5 details the management performed in the *system* component to communicate events and state throughout the system.

4.1 Volume

The *volume* component of the system provides the organization needed to efficiently access, manipulate, and view data. This is accomplished using a hierarchy of classes to encapsulate functionality.

4.1.1 Volume

The root of this interface hierarchy is a single Volume class that enables access to all data managed by the system. A Volume contains no direct access to data, but manages references to multiple sets of segmentation data. In addition to managing access, a Volume also performs the conversion between the multiple coordinate frames the system uses.

- **Data.** The data coordinate frame is used to address discrete voxels. It is used primarily to specify axis aligned bounding boxes of volume data for access or modification.

- **Normalized.** The normalized coordinate frame is a internal continuous representation of the extent of the volume. The normalized frame scales the data coordinate frame such that the entire volume is accessed between zero and one on all dimensions. This means any coordinate outside the range of zero to one lies outside of the volume. It is used for the geometry of meshes as well as the geometric culling (section 4.1.4) which require a consistent and accurate coordinate frame.

- **Spacial.** The spacial coordinate frame is a dynamic, user defined frame that gives the user more control over the representation of their data. It is a scaled version of the normalized frame that is also on a continuous scale. This scale factor is user defined, so that although the data may have been scanned at a particular resolution on each dimension, it can be transformed to the appropriate physical dimensions. This allows the user to view 3D objects that have been scaled to represent the proportions of the correlated physical tissue volume, rather than being forced to view only at the discrete proportions that the data was scanned at.

As a single volume system, all contained sets of segmentation data must share the same extent. This is a deliberate limitation in the system so as to simplify interaction between data sets, avoiding the complexity of interpolating voxel data when trying to associate data between differing data sets if there is not a one to one correlation between voxels. A Volume is also used to correlate data to a user defined spacial extent.

4.1.2 MipVolume

To achieve the scalable performance, the data of a segmentation is abstracted further into a MipVolume. This class applies the mip-mapping technique of generating pyramidal data structures in an oct-tree hierarchy to efficiently compress multiple scaled versions of the data[12]. The leaf level data set of the MipVolume directly correlates to a source segmentation data volume. Successive levels are subsampled such that adjacent octals of voxels in a child level are reduced to a single voxel in the parent level. [1] With each parent reducing the dimensions of its child by a factor of two, and thus the data in the volume by eight, compact representations of large data extents are compressed into a small amount of data.

[1] Although segmentation data is directly subsampled so as to preserve sampled segmentation data values, and thereby the value a segmented region is associated with, the subsampling process can involve linear interpolation or other filters to better preserve detail when mip-mapping image data as is the case of Channel data as described in section 5.3.4.

This reduction occurs until the level is sufficiently small, at which point that particular set of data is considered the root volume level.

4.1.3 MipChunk

The MipChunk class is the final abstraction in the data hierarchy of the system. A MipChunk augments the compression of a MipVolume with a hierarchical coordinate system so as to allow a MipVolume to define and manage regions of data within mip-level volumes. A MipChunk contains directly accessible data addressed as cube of uniform dimensions containing data extracted from a mip-level volume. Each MipChunk can be uniquely addressed using a mip-level, and orthogonal x, y, z, values in a four dimensional coordinate system of MipChunkCoordinates. Based on the octal subsampling of the MipVolume, the coordinate system of the MipChunks form an octal tree such that each MipChunk of a parent mip-level volume is related to eight children MipChunks. The parent MipChunk and octal children MipChunks share the same extent in space, although the parent is a subsampled version of the region covered by the children. The MipChunk that entirely contains the root mip-level volume is known as the root MipChunk and does not have a parent.

Each MipChunk summarizes relevant information into associated metadata. The most significant portions of this metadata are sets of data values that uniquely correspond to segmented regions in the volume data. These sets define the values to be either directly or indirectly contained by the MipChunk.

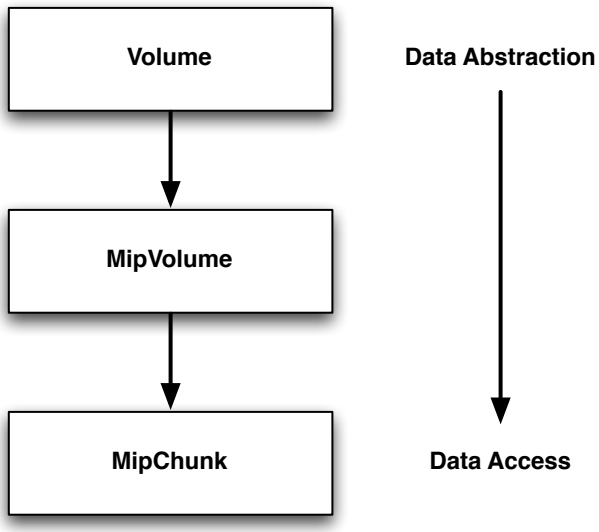
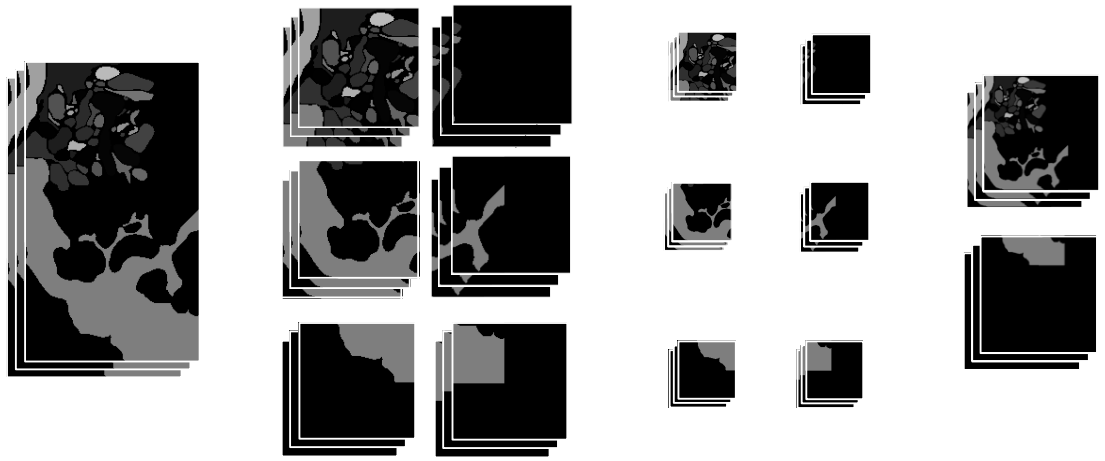


Figure 4-1: The classes used in the data abstraction hierarchy in *volume* component.



(a) Source volume (b) Leaf chunks (c) Subsampled chunks (d) Parent chunks

Figure 4-2: A source volume (a) that has been copied into cubic leaf level MipChunks (b). The leaf level MipChunks are subsampled (c) and merged to form cubic parent level MipChunks (d).

- **Directly contained.** Directly contained values are those values which can be found in the volume data contained by the MipChunk.
- **Indirectly contained.** Indirectly contained values are the union of indirectly contained values of the children of a MipChunk, or are the same as the directly contained values if the MipChunk is associated to a leaf mip-level (called a leaf MipChunk).

As a result of managing this associated metadata, any value in the data organized as a MipVolume can be bounded into specific MipChunks by performing a simple oct-tree structured search.

4.1.4 Hierarchical Culling

Utilizing the oct-tree structure of MipChunks, the system uses hierarchical rendering techniques to efficiently limit processing to relevant segmentation data in a MipVolume during the visualization process. Significantly, this allows the rendering process to quickly discard large regions of volume data when traversing the oct-tree structure by using information in parent MipChunks to determine if the child MipChunks even needs to be considered. The two hierarchies considered are geometric and data.

- **Geometric.** A common rendering optimization technique is the use of frustum culling with a hierarchical rendering tree to quickly discard objects from a scene [2, 11]. Frustum culling uses the geometric viewing frustum formed when drawing a 3D scene onto a 2D screen with specified minimum and maximum viewing distance from the point of a viewing center. If this frustum intersects a parent object in the rendering tree, the object is considered visible and the visibility of the enclosed children objects must also be checked. Otherwise the parent can be discarded, saving the effort of testing the children. Using the spacial extent of a MipChunk the same principle is applied using the tree structure of enclosed spacial regions. When a parent MipChunk is determined to be non-visible, then neither are any of the data that is contained within its children, so the rendering process need not attempt to render any of its children MipChunks.

  Further optimization can be achieved using a refinement distance metric to limit refinement to only MipChunks that are within a specific distance of the view center of the viewing frustum. This allows the system to improve performance by using higher-resolution MipChunks when closer to the viewer and low-resolution subsampled MipChunks when further away. This improves the perceived quality of 3D objects by using high-quality objects when they are visible and close in proximity to the viewer. The performance benefit of this method is detailed in section 4.3.1.

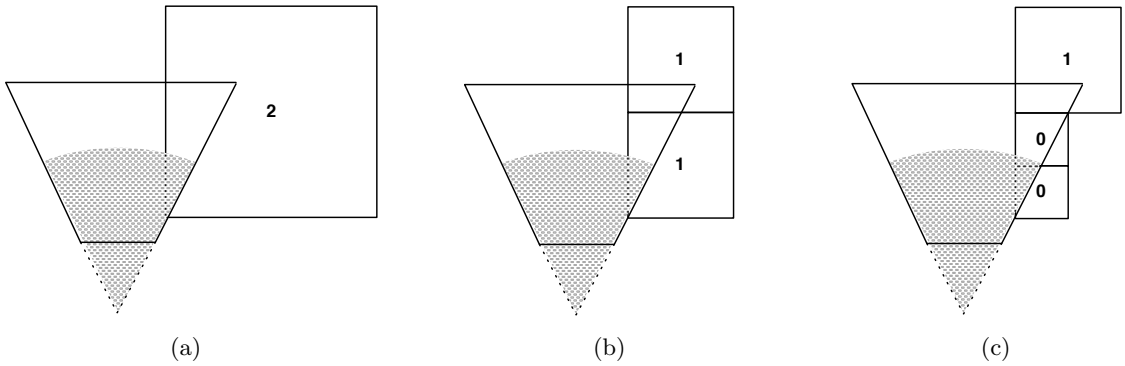


Figure 4-3: An example of the geometric based hierarchical culling process used with the viewing frustum. The shaded area designates the region contained by the refinement distance from the view center. (a) shows an initial frustum intersection with a MipChunk at mip-level 2. As part of this MipChunk is contained within the refinement region, the frustum performs intersection tests with the MipChunk's children. Non-intersecting children are discarded as shown in (b). The refinement process is repeated for the closest child mip-level 1 MipChunk to create the final MipChunk layout shown in (c). The resulting layout produces two leaf level (mip-level 0) MipChunks that are rendered near the viewer and another MipChunk of mip-level 1 that is rendered further away.

- **Data.** A complementary hierarchical culling procedure can be applied using the data value contents of MipChunks. Using pre-processed MipChunk metadata, MipChunks and their children can quickly be culled based on their possible value content as indicated by the sets of indirectly contained data values. This culling technique is utilized in the visualization process by intersecting a set of data values to be rendered with the indirectly contained data values of a MipChunk to determine if further culling is necessary. The resulting set of "relevant data values" indicates which of the initial values are contained by the spatial extent of the tested MipChunk. If the resulting relevant set is empty, then the MipChunk and its children can instantly be culled. Otherwise if the set is non-empty, then the children MipChunks need to be intersected as well to further refine the data region of MipChunks that contains the data values of interest. Significantly, these recursive intersections can be optimized by using the resulting relevant set of a parent intersection when intersecting with children to help bound the number of values used in later intersections. This is because the relevant sets can be no larger than the set intersected with the parent and only contains values desired to be rendered that are possibly contained within the children MipChunks.

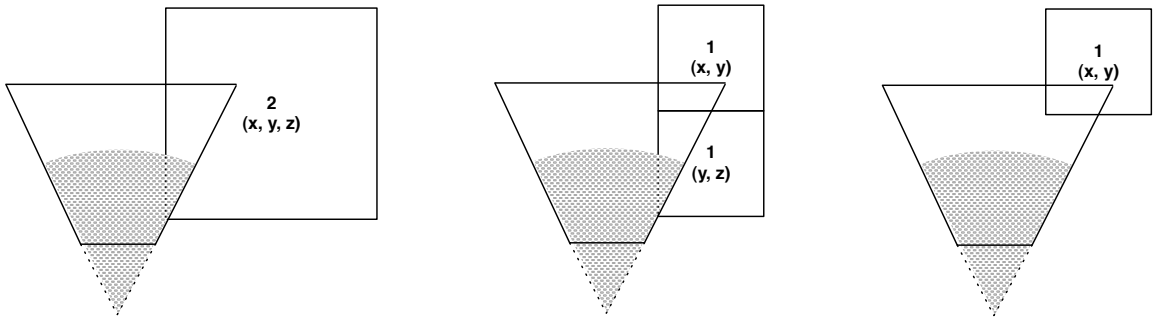


Figure 4-4: An example of data based hierarchical culling augmenting the geometric culling process. The shaded area designates the region contained by the refinement distance from the view center. MipChunks are shown with their mip-levels and sets of indirectly contained segment values. In this example, segment values **x** and **w** are desired to be rendered, however the MipChunk shown in (a) only contains segment values **(x,y,z)**. The intersection of **(x,w)** with the indirectly contained values **(x,y,z)** of the visible MipChunk produces the relevant data value set **(x)**. The MipChunk is refined and child MipChunks are geometrically culled as shown in (b). Data culling is performed by intersecting the relevant values set with the indirectly contained values of the children MipChunks, leaving only a single MipChunk as shown in (c). This MipChunk is outside of the geometric refinement distance, and needs no further refinement. Segment data of value **x** is rendered in the remaining MipChunk of mip-level 1.

The combination of these two hierarchical culling techniques allows the system to quickly and efficiently discard regions of a segmentation volume to consider when rendering in the visualization process. The inexpensive frustum intersection tests makes spacial culling the primary step of culling data while the possibly more costly set intersection technique of data culling is the second step of region refinement.

4.2 Segment

The *segment* component of the system functions as an interface and manager of abstract Segment objects. While the topology of a segment object is determined by the data in a segmentation volume, an additional level of abstraction increases user functionality and improves editing performance.

4.2.1 Segment Objects

Further abstraction extends the functionality of 3D segment regions by associating each segment ID to a Segment object class. This class allows users to assign a specific segment region attributes such as a name, color, and notation. These segment attributes are a simple but powerful way for users to organize segmentation data.

4.2.2 Data Mapping

As described in section 2.1, a common action performed when editing segmentations is merging multiple segments into a single segment. If every segment was specified by a unique value found in the segmentation data, a merge action would require changing the values in each voxel to so as to match the value of the final merged segment. The cost of this method scales proportionally to the volume of data associated with the segments that will be merged into a different segment. This is a significant performance expense when merging many large segments. In contrast, through a level of indirection, the proposed system is able perform common editing actions that appear to modify significant quantities of data nearly instantly. This is accomplished by bidirectionally mapping a set of data values to a unique segment identification (ID) number. A single segment is therefore associated with multiple regions of differing data values. In particular, this indirection allows various segments of unique IDs to be quickly merged simply by finding the union of the data value sets associated with the segment IDs to be merged and mapping it to the new ID.

4.3 Mesh

The *mesh* component of the system provides the interface for generating and displaying 3D mesh objects. Mesh generation utilizes the performance of the *volume* component to extract volumes of segmentation data to reconstruct mesh representations. Mesh rendering takes advantage of the segment abstraction provided by the *segment* component to apply visual properties to the 3D objects and render multiple regions of segmented data as a single Segment object. The *mesh* component is composed of a MipMesh class which contains and draws meshed data and a MipMesher class which generates MipMeshes from segmentation data.

4.3.1 MipMesh

A MipMesh encapsulates the data structure and functions needed to draw a tessellated surface representation of a portion of a 3D segment object to the screen. Specifically, a MipMesh is a reconstructed surface mesh defining a region of similar values contained within the volume data of a single MipChunk. As the segmentation volume of a MipChunk can contain multiple data values, multiple MipMeshes can be associated to the same MipChunk. Furthermore, this leads to a five-dimensional MipMeshCoordinate system where four dimensions represent the four-dimensional MipChunkCoordinate and the fifth dimension represents the value of the data region that the mesh surface describes.

A traditional approach to 3D object meshing would associate each mesh to a single unique data value, regardless of the boundaries of MipChunks. In this way, only a single mesh object would need to be rendered to represent all the data of a single value. In contrast, the method of breaking a single mesh into pieces, or MipChunks, as proposed by this system offers significant scalability benefits.

- **Complexity bounding.** By limiting the extent of segmentation data a mesh can represent to a single MipChunk, the system is bounding the size and complexity of the associated surface mesh that can be reconstructed from that volume. The mesh generation processes is then limited to MipChunks containing the region to be meshed as opposed to a single loosely bounded extent. The rendering process benefits from the bounded mesh size and complexity because meshes are limited to manageable memory size, ensuring that meshes can be loaded from disk and drawn in a reasonable amount of time.

- **Fractional surface drawing.** Breaking the tessellated surface of a region into multiple MipChunk bounded MipMeshes also allows the system to take advantage of the hierarchical culling described in section 4.1.4 through fractional surface drawing. In the traditional meshing approach, even if only a small part of the surface was visible, the entire mesh surface would need to be drawn. Many of the triangles of the mesh will be needlessly rendered since they would be not be visible outside of the viewing window. With the fractional surface representation of MipMeshes and hierarchical spacial culling of MipChunks, only the surface contained within a MipChunk that is determined to be visible will be rendered, significantly reducing the number of non-visible triangles that need to be drawn.

- **Multiple Resolutions.** Using the subsampling relationship between parent and child MipChunks, MipMeshes correspondingly have a multiple resolution relationship with segmented regions. This means that MipMeshes associated with parent chunks are meshes of subsampled child MipChunks. As a result, these lower resolution meshes have less geometry and are smaller in size and faster to render. A user is then able to choose the level of MipChunk refinement and thereby MipMesh resolution depending on the performance or mesh quality preference.

The use of MipMeshes to break surfaces into multiple meshes incurs expenses as well. Mesh organization becomes more complex as each MipChunk needs to keep track of the set of MipMeshes associated to it. As each mesh requires a MipMesh object to describe it, there is an increased memory overhead for using multiple MipMeshes to render what could be displayed with a single mesh. Furthermore, when multiple MipMeshes are used to describe a single surface, they must contain redundant border geometry along their seams so as to appear to be a uniform surface. Due to the independent nature of each MipMesh, this boundary geometry data is impractical to share and so adjacent MipMeshes must contain redundantly copies of this data. Yet as scalability of the proposed system is a higher priority than data efficiency, the performance benefits from using MipMeshes greatly outweigh the described memory costs.

4.3.2 MipMesher

The MipMesher class is used to generate MipMeshes related to a specific MipChunk. This is accomplished by specifying data values contained in data volume of a MipChunk, and

using a multi-staged meshing pipeline to extract surface tessellation of regions associated to these data values. The stages of the pipeline are ordered as follows:

1. **Surface reconstruction.** Using the Marching Cubes algorithm as described in section 3.2.2, a high-resolution 3D surface tessellation can be extracted from a volume of segmentation data for a specific value defining region in the data. This extracted set of triangles is extremely dense due to the per voxel resolution sampling performed by the algorithm to generate an interpolated iso-surface of the region [10].

2. **Decimation.** Mesh decimation is the processes of reducing triangle count while preserving topology [9]. Specifically, the system employs a surface simplification technique to iteratively contract mesh vertex pairs while preserving salient surface features as defined by a quadratic error metric [7, 8]. Due to the high-resolution tessellation resulting from Marching Cubes, the mesh extracted in the previous stage is decimated to a user defined target percentage. This allows the user to prioritize between mesh data size and detail.

3. **Transformation.** The mesh geometry is transformed to the normalized extent of the MipChunk which is described in section 4.1.1. Pre-transforming to the normalized extent avoids the cost of repetitively transforming MipMeshes during the rendering process.

4. **Smoothing.** Even after decimation, the mesh can still retain much of the blocky appearance caused by the Marching Cubes algorithm when reconstructing surfaces from discrete voxels. The user can specify the use of a mesh smoothing filter that "relaxes" the mesh by moving vertices to achieve a more even distribution [6]. Specifically, the system employs a Laplacian based smoothing filter that can be used iteratively to achieve varying levels of smoothness based on user preference.

5. **Normal generation.** After the vertex geometry has been modified by the reconstruction, decimation, transformation, and smoothing stages, it can be used to calculate normal data. Using the location of neighboring vertices, the surface normal at each vertex is calculated and added to the mesh data to allow the mesh to have shading interaction with light sources in the 3D environment.

6. **Stripping.** The set of triangles is finally reorganized into contiguous strips. As adjacent triangles share two vertices, strips of triangles approach a two-third reduction in geometry as the strip length grows compared to the number of vertices in a set of individual triangles. As a result, a mesh that is made of contiguous strips of triangles has a smaller memory requirement and renders faster than a mesh of individual triangles.

After a MipMesh has been generated it is saved to disk so it can be quickly loaded and rendered during the visualization processes. Careful MipMesh data management and the use of the outlined meshing pipeline allow the MipMesher class to efficiently form compact representations of 3D data regions.

4.4 View

The *view* component is responsible for interactively interfacing the user with the 3D representation of segmentation data. The component directly handles both mouse input and system events to cause the screen to refresh with updated material. The *view* component breaks apart functionality into several classes.

- **View.** The View class represents the actual screen space that 3D objects can be rendered in. This window exists to respond to events from a user or the system, make a visible change in the system, and initiate the rendering processes to make the change visible to the user.

- **3D user interface.** When the View class receives user input, such as mouse movement or keystrokes, they are sent directly to the 3D User Interface class. This class filters input events for relevancy and interprets them as changes to the system. These changes may include camera movement, segment object property changes, or even voxel modifications if the user is in editing mode.

- **Camera.** The Camera class manages properties related to the position, orientation, and perspective of the user in 3D space. These properties can be modified by the 3D User Interface class to give the user the ability to freely move in space to visualize the data. Each View class has an associated Camera, allowing views to be used independently so that a user can look at multiple perspectives of the same object.

The ability to tightly control the properties of multiple Cameras is particularly useful during editing when multiple perspectives enable a better understand how a subtle 3D topographical modification is altering connectivity with surrounding objects.

- **Widgets.** Widgets classes are rendering layers that added to the View window after all segment objects have been drawn. Examples of Widget functionality highlighting selected segments and displaying camera location information. Widgets can be dynamically enabled or disabled from a view, allowing the users to customize how their visualization is augmented depending on the task.

4.5 System

Th *system* component serves to synchronize the other components whose functionality depend on events or specific system states. The following classes are critical to managing the shared data that unifies these events or states.

4.5.1 Event Manager

Actions taken by the user can impact many pieces of the system. For example, changing the color of a segment object requires having all view classes redraw so as to reflect this change. Another example includes changing the system user mode from 'Edit Mode' to 'Navigation Mode' as detailed in section 3.2.3. This mode change causes all edited MipChunks in a MipVolume class with edited data must be rebuilt to reflect these changes. These various classes are notified through an event system controlled by an EventManager class. The EventManager maintains a correlation of specific events and instances of classes that have registered to receive such events. If the EventManager is notified of an event that occurred anywhere in the system, all registered instances of classes are in turn notified of the event and data that may correspond to the event. In this way, all Views can redraw themselves when they are informed that the properties of a segment object has been modified and a MipVolume can rebuild necessary MipChunks after being notified the system state has changed.

4.5.2 State Manager

Although events serve to notify classes of changes to the system, sometimes components need to determine systems state so as to function differently. The StateManager maintains

the state of various aspects of the system. For example when drawing selected Segmentation objects, the system checks the StateManager to for the current user mode so as to draw a mesh representation of Segment object if in 'Navigation Mode' or a voxel representation if in 'Edit Mode'. In this way, the StateManager is a shared database of states for various components to use.

Chapter 5

Omni: A Case Study

This chapter explores the practicality of implementing the proposed system into a complete end-user application called Omni. The background of the application development is outlined in section 5.1 with application goals discussed in section 5.2. Section 5.3 discusses implementation details for achieving the outlined goals and application performance is evaluated in section 5.4.

5.1 Background

The Seung Lab at the Massachusetts Institute of Technology currently uses a wide variety of software to model and analyze connectomes (neural networks of the brain). Current tools used for connectomic analysis in the lab have enabled significant progress in tracing and visualizing neural data, specifically the construction of 3D models from fragments of segmentation data. Yet specific weakness in design and features of the various software systems used have limited the utility of using a single application. As a result all three software packages discussed in section 2.2 are used in the analysis process, each individually used to perform a specific task to offset a weakness or lack of ability in the other packages. Yet the inconvenience of having a workflow through multiple applications and the overlapping deficiencies of all three systems has resulted in research for a unified software solution for connectomic analysis. The proposed solution is Omni, a tool designed for an integrated workflow utilizing the scalable 3D performance and interactive 3D editing described in this thesis.

5.2 Goals

This section briefly outlines the goals of the Omni application. These goals are specific to the needs of the Seung Lab at MIT so as to maximize the utility of the application. The areas discussed are platform, workflow, and performance.

5.2.1 Platform

The lab currently employs a variety of computer platforms including Apple, Linux, and Windows. The most popular machines in the lab are the Unix-based Apple and Linux platforms. For this reason it was decided that Omni should undergo cross-platform development to support these two types of Unix-based systems. [1]

5.2.2 Workflow

As a unified replacement for a few individual software systems, it was important that Omni provide a complete workflow for importing, editing, and visualizing segmentation data. In particular the application must combine the familiar 2D visualization and editing from other systems with the 3D counterparts detailed in this thesis. Finally the application must be able to import and export in the common regularly structured volume data format that the lab uses to process data. This will allow the application to be able to import channel and segmentation data while allowing it to export any modified segmentations. This uniform import and export format gives Omni end-to-end functionality as a tool for all visualizing and editing.

5.2.3 Performance

The development of Omni is also a result of increasing performance demands of visualization interactivity. As detailed in section 2.2, related software suffers significant increases in load times and decreases in frame rates when interacting with ever increasing data sets. The aim of the Omni application is to leverage the scalability of the system proposed in this thesis to achieve interactive 2D and 3D visualization and editing without frequent and noticeable loading.

[1] The Omni application currently supports Apple OS X 10.5 and Linux Ubuntu 8.10.

5.3 Implementation

This section describes the significant implementation details that facilitate the platform, workflow, and performance goals outlined in section 5.2.

5.3.1 Language

Omni is built entirely using the middle-level C++ programming language. C++ was chosen because of its unique combination of performance, object-oriented support, and large library base. To meet the performance goals desired for the application, it is clear that Omni needs to be built in a compiled language such as C++ that can control the low-level details of memory management so as to most effectively access and modify the vast amount of data it manages. This is contrast to scripted languages, such as MATLAB, which must be interpreted and do not have fine-grain control over the efficiencies of memory interaction. Because of the heavy use of abstraction in the proposed 3D visualization and editing system, a language that could support object-oriented design would be most effective to construct classes such as Volume, MipVolume, and MipChunk (see section 4.1.1). Finally, the C++ language has the advantage of having wide variety of compatible libraries that are available under open-source licensing. Using these external libraries reduces the amount of proprietary code needed and enables faster application development.

5.3.2 Libraries

Careful work was performed to find libraries and a build system to support cross-platform development. The following details the external libraries used and their function in the application:

- **VTK.** The Visualization Toolkit (VTK) is an open-source, cross-platform library used for image processing and 3D visualization. The application uses VTK for reading image slices as well as the complexities of the meshing pipeline.
- **Boost.** The Boost C++ Libraries offer open-source and cross-platform serialization and regular expression parsing.
- **vmmlib** The cross-platform vmmlib vector and matrix math library developed at the Visualization and Multimedia Lab at the University of Zurich is used to perform 2D and 3D geometric computation throughout the system.

- **OpenGL.** The cross-platform graphics interface of OpenGL is used for rendering 2D and 3D objects in the application.

- **Qt.** A cross-platform application and user-interface framework. Primarily used for the design and implementation of the graphical user-interface, the Qt framework provides the base for many of the common application features such as undo-redo and window management.

5.3.3 Data Formats

To support the end-to-end workflow design, the Omni application must import and export volume data formats that the lab can work with. This section discusses the various volume data formats which Omni must support. These are the formats that the application can import from, the format that it internally maintains, and finally the format which it can export to.

Volume data from the lab is generated as either multiple files as slices in a 2D structure or a single file with a 3D structure. The 2D slices are commonly sets of traditional image data in PNG or TIFF file format, while the 3D structures are maintained in a HDF5 format. The application can import both channel and segmentation data as either slices of PNG, TIFF, or JPEG files or as a complete HDF5 volume.

Internally, the system converts all volume data into HDF5. This is done to increase storage depth and performance. To support segmentations with potentially billions of uniquely segmented regions, the internal file format for segmentations uses HDF5 with 32-bit voxel values for an extremely large bit-depth support. In contrast, traditional 2D file formats have a conventional limitation of 16-bits per voxel. [2] Channel data is also stored in HDF5, but at depth of 8-bits since this is standard depth of raw image data. Furthermore, image formats such as PNG do not have an uncompressed format and must be compressed or decompressed whenever accessed. This adds the computational price of compression to the already costly processes of reading and writing volume data on disk. Omni avoids these costs by using an uncompressed data format within HDF5, helping reduce the penalty of costly disk access.

Unlike channel data, a segmentation can be modified. This modified data may need

[2] More specifically, this is the maximum bit-depth per color or alpha channel per voxel. But the needs of the system are contiguous bit-depth, so without going through the effort of partitioning values and storing parts into individual channels, the practical limitation is 16-bits.

further processing by another application. Omni allows this volume data to be exported. Yet as a result of the internal HDF5 format having a higher bit-depth than conventional 2D image formats, the application is limited to exporting only in HDF5. If necessary, software such a MATLAB could be used to filter the HDF5 volume data to a conventional bit-depth that can then be converted to a 2D image format.

5.3.4 Volume Component

The *volume* component of the proposed system has performance critical functionality that must be considered during implementation. The following discusses how the implementation optimizes data structure management and volume data access such that the other system components can quickly access needed data.

Much of the *volume* component is organized as hierarchy of containing classes. For example, a Volume maintains Segmentations and Channels which are forms of MipVolumes, while a Segmentation maintains Segment objects. It is important that the organization overhead does not hinder access performance. Fast management is achieved through a templated GenericManager that maps each managed object to a unique 'OmId'. The templated nature of the class allows the GenericManager to be used to manage any type of class, allowing for significant code reuse. Using OmIds the GenericManager can assign state characteristics to objects adding or removing them to specific "enabled" or "selected" sets. Managed objects must inherit from a ManagableObject class so they have an assigned OmId and other attributes. In this way a GenericManager can access any given object with a specified OmId in logarithmic time and return a set of OmIds that share a specific state in constant time, allowing objects to be quickly accessed or filtered by state.

MipChunk volume data is heavily used throughout the application. To support a 2D visualization system, MipChunks must allow 2D slices to be quickly extracted from associated volume data. The 3D visualization system requires fast access to volume data so as to quickly extract meshes. Both systems must be able to quickly modify this data to support editing. But MipChunks can be costly to read and write because of the cost of disk access compounded by the internally uncompressed volume format that takes up significant space. Omni alleviates cost of MipChunk access with a cache structure called a GenericCache. Objects that inherit from a CacheableObject can be added to the GenericCache, mapping them to a key that allows logarithmic access to the object instead of the costly process of

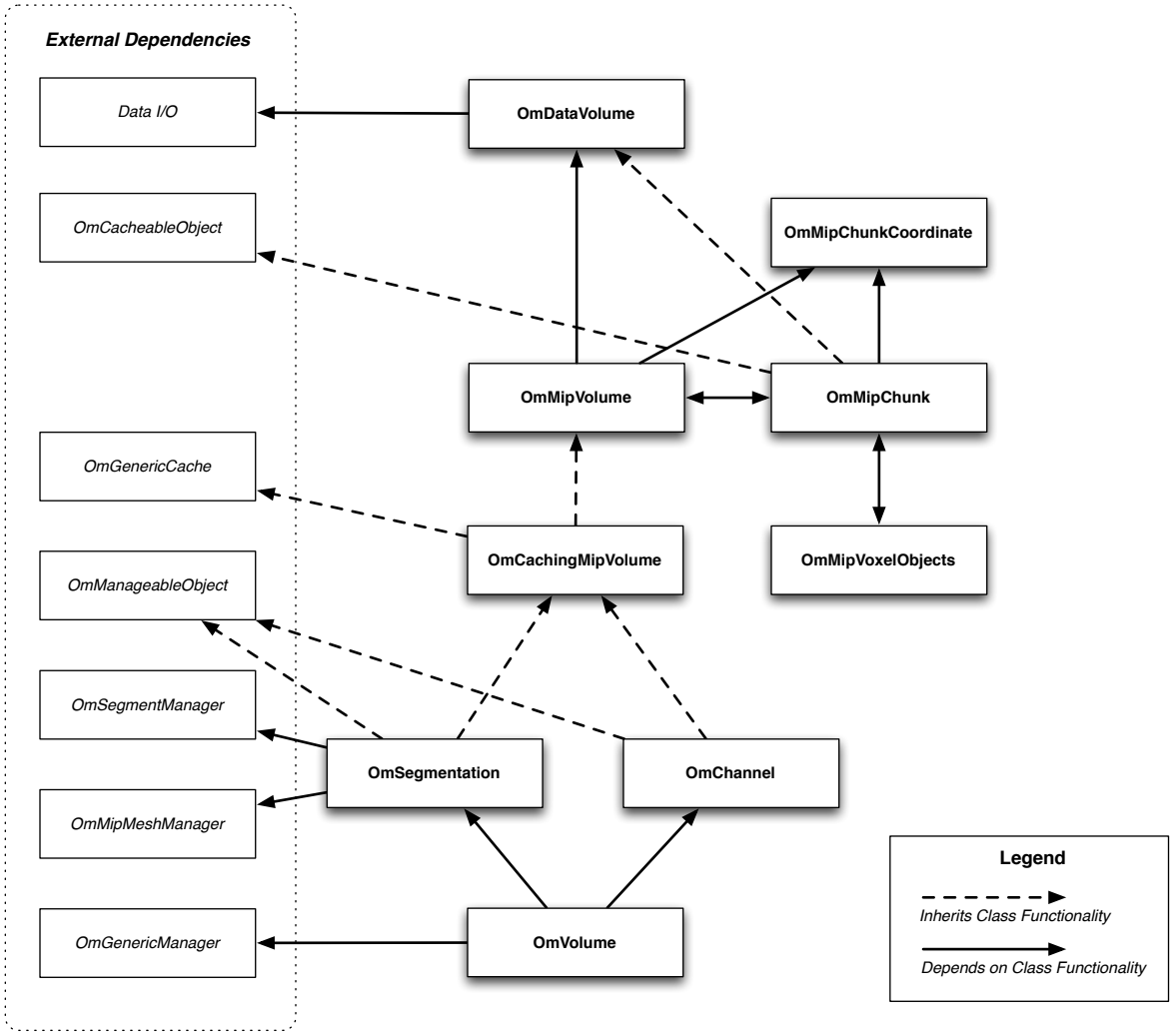


Figure 5-1: Dependency diagram of major classes in the *volume* component of the Omni application.

reading from disk and, if any modifications were made, writing back to disk again. Instead the cost of writing modifications is more effectively batched once the cache becomes full. The cache is emptied by least recently accessed ordering to a specific target percentage, such that the remaining objects in the cache are those that have been most recently accessed. As a result, the volume data that is most likely to be relevant is usually cached and inexpensive to access. As a CacheableObject, MipChunks take advantage of all the benefits of caching through a CachingMipVolume class that inherits from both the MipVolume and GenericCache classes. This greatly improves the rate of 2D slice extraction and 3D object generation. Furthermore, since the caching also stores the associated metadata of the MipChunk, the cache process dramatically improves the data based hierarchical culling as well.

MipVolume building is the process of creating multiple resolutions with pyramidal structure of an original source data volume. To take advantage of the caching mechanisms of CachingMipVolume, the building process generates subsampled MipChunks in a recursive depth-first tree with reverse ordering. That is, as the branches of the oct-tree structure of the MipVolume are recursively built, all children MipChunks must be built before a parent MipChunk can be built. In this way, newly built MipChunks that have been cached are extremely likely to be reused in the building process of parent MipChunks. The performance of this processes is detailed in section 5.4.

The use of the GenericManager and GenericCache classes allow the *volume* component to be quickly and efficiently manage much of the data that the rest of the system relies on.

5.3.5 Mesh Component

Much like the *volume* component, the *mesh* component also uses several techniques to improve performance. These techniques make use of multi-threaded coding to provide parallel functionality. During the mesh generation processes, the highly repetitive processes of extracting meshes from volume data is streamlined with a multi-threaded meshing system while the mesh rendering process uses a threaded cache system to improve user interactivity.

A bottleneck identified early in the development of the system was the mesh generation pipeline. Detailed in section 4.3.2, the MipMesher reconstructs a surface mesh associated with the value forming a data region with a MipChunk. This multi-staged processes is computationally significant and due to limitations of the VTK library used to perform this pipeline, it can only be performed on a single data volume for a single data value region for each iteration. The solution is to run the pipeline in parallel, giving each thread an OmMesher class that contains a copy of the source volume data to be meshed. Directing these threads is an OmMipMesher class that contains a central "todo" set of values associated to segmented regions in the MipChunk. Each thread removes a value from the set and performs the meshing pipeline to generate the associate MipMesh, storing it to disk upon completion. Once all "todo" values have been completed, the OmMipMesher is ready for the next MipChunk to be meshed. Since the threads perform independent pipelines, the number of meshing threads can be controlled by the user, enabling user flexibility in how many resources are devoted to the build process.

The caching of MipMeshes also needed to be threaded for performance reasons. As

MipMeshes are loaded from disk and drawn to screen, there are significant interactivity issues if the user needs to wait until all visible meshes are loaded before a complete frame can be drawn. While a GenericCache helps the problem by bounding the loading to only MipMeshes not yet cached, significant orientation change of the Camera can reveal enough new MipMeshes that the delay hinders the user experience. Again, the solution is parallel processing to create a templated ThreadedGenericCache. Unlike the GenericCache which forces the rendering process to wait until a requested MipMesh is loaded, the non-blocking nature of the ThreadedGenericCache informs the rendering process that the MipMesh is unavailable and adds it to a queue of MipMeshes to be loaded. This allows the rendering process to continue so as to attempt to render the next MipMesh while the ThreadedGenericCache concurrently has a "fetch" thread that loads queued MipMeshes from disk. After a designated elapsed time of fetching MipMeshes from disk, the "fetch" thread notifies the View to redraw the screen so that MipMeshes that are now available to the render process can be drawn to the screen. This threading is crucial in allowing the user to smoothly navigate through vast numbers of MipMeshes without suffering significant loading delays.

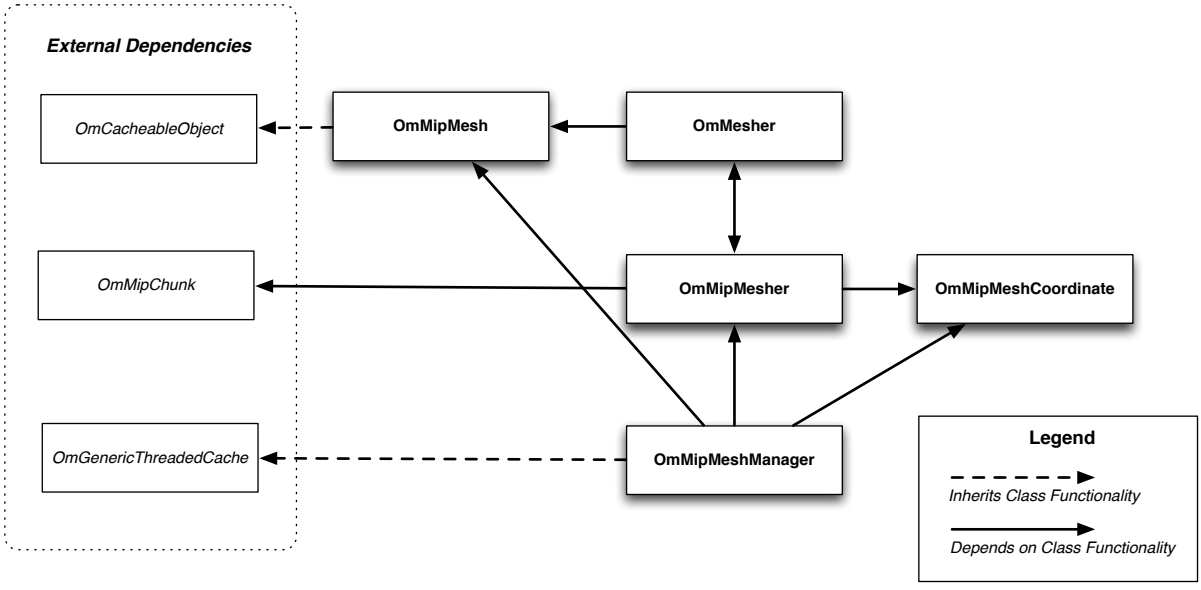


Figure 5-2: Dependency diagram of major classes in the *mesh* component of the Omni application.

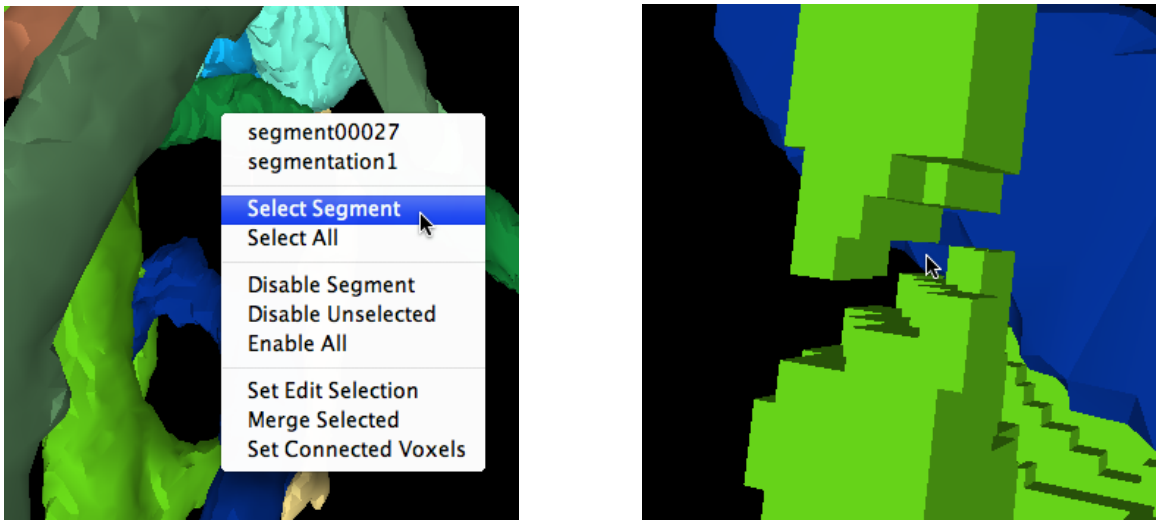
5.3.6 3D User Interface

The 3D user interface of Omni is implemented using a combination of functionality provided by the OpenGL library and the Qt toolkit. This interface allows the user to directly interact with the segmentation data in a 3D space. The primary interactions supported include Camera movement, Segment object interaction, and Segment voxel editing.

Camera control is similar to what is found in many other 3D visualization software, using traditional "click-and-drag" mouse interaction to change a variety of Camera properties. Depending on the mouse button depressed, the camera can be rotated, panned, or zoomed about cross-hairs that mark the "focus" of the Camera. Additionally, the cross-hairs are always axis aligned so as to help orient users to the scene.

Segment object interaction, such as double-clicking a rendered mesh to select a specific Segment object, is performed via rendering in "selection" mode in OpenGL. This renders a small region of the screen underneath the mouse such that the object rendered are uniquely named and stored in a "selection buffer". Analyzing this buffer to find the closest object allows the system to change properties such as selection and give the user the impressing that they can directly interact with Segment objects using the mouse. This selection rendering technique is also used to generate relevant contextual menues. After determining the closes Segment object under the mouse when right-clicking, Omni uses Qt to generate a visible contextual menu on screen. This menu can be used to easily "enable" or "select" specific Segment object or perform editing actions such as merging a set of selected Segment objects.

Voxel interaction during 'Edit Mode' makes use of the depth buffer in OpenGL to determine the voxel associated with a specific pixel that a user clicks on in a rendered image. For every rendered frame, the depth buffer stores the 3D distance from the Camera to the object each pixel represents. This object location can be determined using a process called "unprojection". Given a specific pixel and the distance associated to it in the depth buffer, the perspective and 3D location of the Camera can be used to unproject the pixel to the 3D location in space. A simple transform converts this location to the containing voxel, allowing users to specify specific voxels by using the mouse to click a 2D pixel on the screen. So if a user wants to remove voxels from a Segmented object, they need only click over the specific rendered voxel in the segmentation. The process is extended by offsetting the unprojected location closer to the camera, allowing users to select voxels "on top" of



(a) Segment object interaction

(b) Segment voxel interaction

Figure 5-3: Examples of the 3D user interface in Omni. The use of selection mode rendering to create a context menu (a) and pixel unprojection to enable interactive voxel removal (b).

the voxel associated to the 2D point that the mouse clicked. This allows users to easily augment a segmentation with a new voxel layer simply by clicking on a voxel to be covered.

5.4 Preprocessing Performance

To achieve interactive 3D visualization performance, the system performs significant data pre-processing performed in two stages. First the source segmentation volume is built into a MipVolume to allow for efficient multi-resolution volume data access. Secondly, MipMeshes are built from all segmented regions in every MipChunk within the MipVolume. This generates multiple resolution surface meshes for all segments in the segmentation. The building process is intended to amortize the cost of using the MipVolume and MipMesh structures that would be otherwise too expensive to generate interactively.

Figures 5-4 and 5-5 demonstrate the exponential growth in build times and data size as the dimensions of the source volume double from 175^3, 350^3, and 700^3 voxels. The use of caching and threading can be seen to have an impact on preprocessing performance. Caching proves to have a marginal advantage during the MipVolume build process as shown in 5-4(a), while threading shown in 5-5(a) dramatically reduces MipMesh building times.

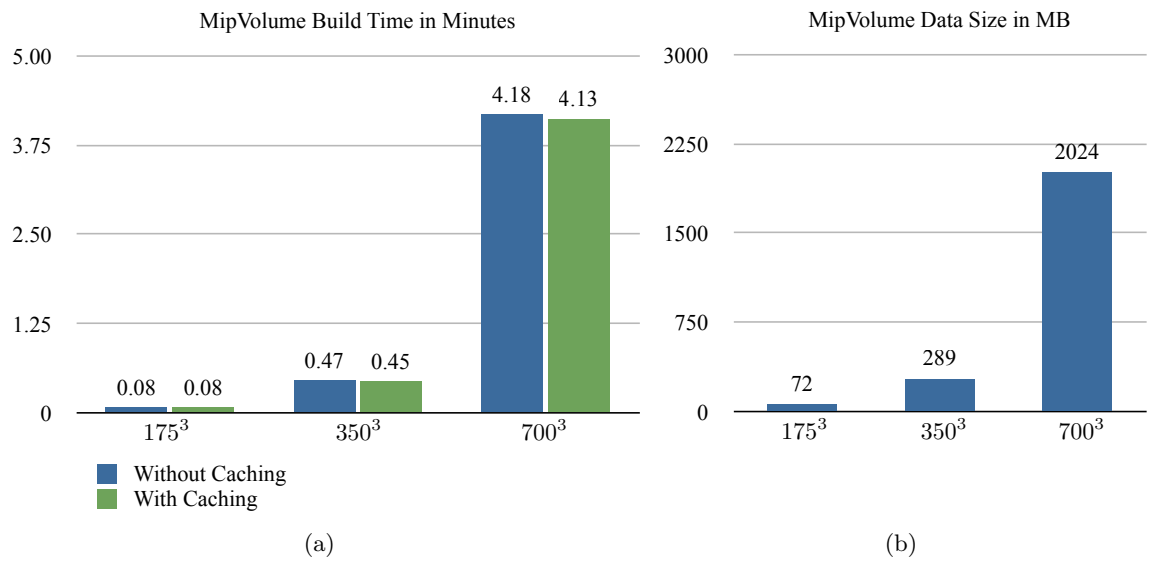


Figure 5-4: Performance comparison of building MipVolumes from cubicly shaped segmentation data volumes with 175^3, 350^3, and 700^3 voxels.[3] (a) shows a comparison of build times with and without a caching. The MipChunk cache size was 128MB. (b) shows the resulting size of the MipVolume data.

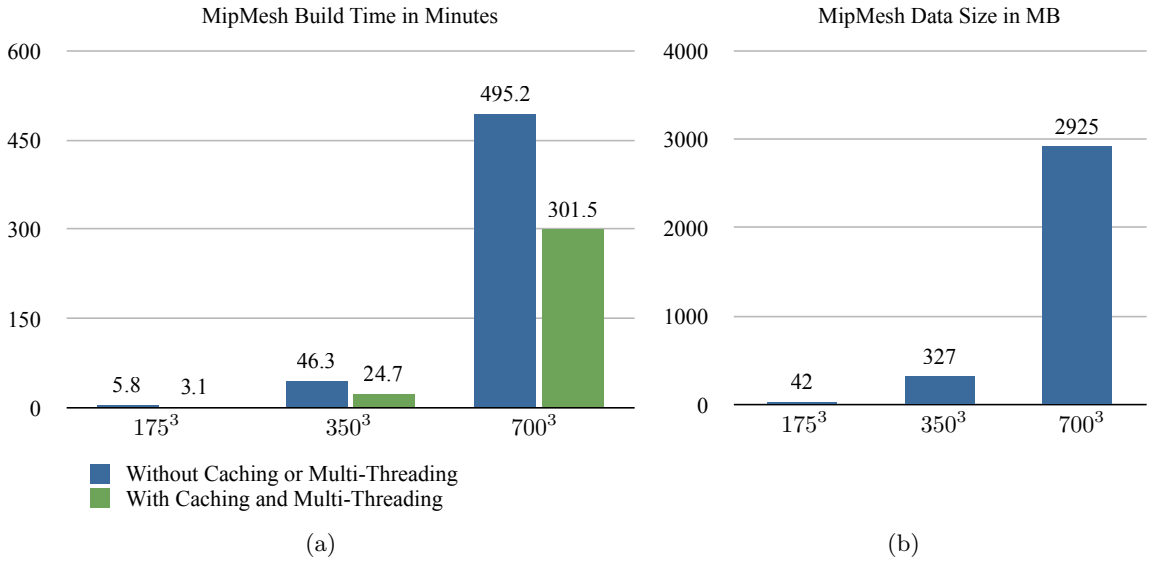


Figure 5-5: Performance comparison of building MipVolumes from cubicly shaped segmentation data volumes with 175^3, 350^3, and 700^3 voxels.[3] (a) shows a comparison of build times with and without a caching and multi-threaded support. The MipChunk cache size was 128MB and two threads were used. (b) shows the resulting size of the MipMesh data generated.

[3]All results are the average of two trials performed on a dual core Intel Xeon 3.0GHz CPU with 2GB of RAM running Ubuntu Linux 8.10

Chapter 6

Conclusion

This chapter concludes the thesis by detailing how the proposed system could be extended through future work in section 6.1 and finally summarizes the major contributions of the system in the closing remarks of section 6.2.

6.1 Future Work

The primary challenges of 3D visualization that the system specifically seeks to address are in the areas of scalability and 3D editing. As these are very broad problems regarding data management and user interfacing, the solutions proposed by the system are very open ended. This section describes how the solutions to these challenges could be further explored or extended.

6.1.1 Scalability

Scalable performance is a primary consideration for many components of the system. The key classes in the system that facilitate this scalability are the MipVolume and MipMesh. Each of these classes has the capacity for design modifications that could result in performance improvement.

In section 4.1.2, the use of MipVolumes to scale the performance of data access and modification of multiple resolutions of segmentation volume data is outlined. This model of data structure depends on eventual refreshing from the modified leaf-level data to rebuild higher level volumes until any changes have propagated to the root-level volume. While this cost is amortized by only performing rebuilds during user mode changes as described in section 3.2.3, rebuilds can be extremely wasteful by rebuilding entire chunks when only a small number of voxels have been modified. A possible solution to this would be the

use of immediate local updates. After a voxel is modified in the leaf-level, a MipVolume would determine if the current subsampling method used would cause the change in voxel value to modify the parent level volume. This processes is repeated until the modification does not affect the parent level volume or the root level volume is reached. As MipVolumes representing segmentation volume data use a simple subsampling technique, where a parent volume is a uniform subsample of one in eight child voxels, the probability of a modification propagating to a parent volume decreases by one-eighth each level increase. This makes these local updates a practical way of eliminating costly rebuilds by always keeping all volume levels of a MipVolume up-to-date.

Section 4.3 details the use of MipMeshes to scale mesh generation and drawing. The relationship between MipMeshes and MipChunks shows how MipMeshes extracted from higher-level MipChunks are lower resolution version meshes of larger data extents. These lower resolutions meshes provide the same data extent coverage using a fraction of the geometry of multiple MipMeshes from lower-level MipChunks. As a result, the same region of space can be drawn faster but at the cost of detail. A possibly scheme for further utilizing lower resolution meshes would be to maintain multiple resolutions of meshes with the same data coverage. This means that a mesh extracted from a single data volume for a specific data value would be decimated and saved at multiple target decimation percentages. Avoiding the cost of rebuilding all meshes at a different decimation target, the user could then instantly choose between various levels of meshing to select an appropriate performance and detail tradeoff. The clear cost of this option is that multiple sets of mesh resolutions would need to be managed and stored, but the ability for a user to dynamically modify system performance could be extremely desirable.

6.1.2 3D Editing

A novel feature of the system is the ability to edit segmentation volume data in a 3D environment. The view component provides basic user interaction with the volume using the mouse to selectively add or remove voxels from a Segment object. This simple functionality allow a user to slowly sculpt segmentation data to a desired form. However, when a segmented region of data requires major topographical changes, adding and removing a single voxel at a time can become quite tedious. An area for improvement for 3D editing would be to develop a set of various tools to use while editing. Much like a tool palette common to

most image editing software, a set of 3D editing tools could enable faster and more accurate modifications. Possible tools include:

- **Cutting tool.** When splitting merged segments, many times it is desirable to remove all voxels in a segmented region along a given 2D plane. This plane cuts the segmented region into at least two parts, so that a contiguous part can be relabeled as a new uniquely segmented region.

- **Paint brush.** Just as the thickness of a brush determines the width of pixels it covers when applied in image editing software, a 3D brush could vary in diameter of the hemisphere of voxels it could add to a specified location. This would allow a user to easy add thick layers of voxels to a segmented region.

- **Clone tool.** A clone tool could be used to quickly alter portions of target segmented regions so adopt some of the topography of a source region. This would be particularly useful to copy complex features of segmented regions between similarly shaped segments.

- **Copy and paste.** Channel data are often segmented multiple times with varying parameters. This can cause the resulting segmentations to be a mixture of correct and incorrectly segmented regions. Copy and paste functionality would allow a user to copy the correctly labeled regions across the various segmentation sources and paste them to a final destination segmentation.

6.2 Closing Remarks

As the field of connectomics advances so does the the amount of data that is generated. Smaller imaging resolutions create larger and more detailed raw image sets while improved algorithms increases the rate of segmentation data generation. To keep pace with the technology creating connectomic data, the system proposed in this thesis details a novel approach towards scalable 3D visualization and editing.

The scalable performance of the system derives primarily from a new way to abstract and organize connectomic data. A Volume class represents a rectangular region of segmented tissue. This region can contain multiple segmentations which are abstracted into a MipVolume class. This class organizes multiple resolutions of the segmentation data into

MipChunks that can be quickly accessed and modified. During the rendering processes, this organization allows the system to make use of both spacial and data hierarchical culling to quickly and efficiently bound the amount of work needed to display mesh renderings of Segment objects. This "divide-and-conquer" approach to limiting computational costs using an oct-tree structure facilitates efficient logarithmic performance scaling of the system.

The system also includes unique 3D editing functionality not found in related connectomic analysis software. While meshes are used to give the scalable 3D visualization performance needed to view many large segmentations at once, the companion voxel rendering of Segment objects provides the precision needed to perform detailed segmentation modifications. In contrast to common 2D segmentation slice editing systems, the user interface of the view component of the proposed system enables users to interactively sculpt Segment objects via mouse input to achieve a desired shape. This interface provides a more intuitive and easier way of editing 3D connectomic data.

To achieve the goals outlined in section 3.1, the system makes use of new ideas as well as common 2D scaling and editing techniques that have been modified to work in a 3D domain. The final system merges the performance of scalable 3D visualization and the functionality of 3D editing to produce an architecture for the next generation of connectomic tools.

Bibliography

[1] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 249–266. AMC, 2001.

[2] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounded boxes. *Jourals of Graphics Tools*, 2000.

[3] Jean-Daniel Boissonnat. Shape reconstruction from planar cross-sections. Technical report, Institut National de Recherche en Informatique et an Automatique, 1986.

[4] J. C. Fiala. Reconstruct: a free editor for serial section microscopy. *Journal of Microscopy*, 218:52–61, April 2005.

[5] John C. Fiala, K. Harris, and K. Sorra. *Reconstruct User Manual*, 1.1.0.0 edition, 2009.

[6] Peter Fleischmann. *Mesh Generation for Technology CAD in Three Dimensions*. PhD thesis, Technischen Universität Wien Fakultät für Elektrotechnik, 1999.

[7] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97*, 1997.

[8] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *IEEE Visualization 1999 Conference*, pages 59–66, 1999.

[9] Michael Knapp. Mesh decimation using vtk. Technical report, Vienna University of Technology, 2002.

[10] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[11] Nirnimesh, Pawan Harish, and P. J. Narayanan. Culling an object hierarchy to a frustum hierarchy. In *5th Indian Conference on Computer Vision, Graphics and Image Processing, Madurai, India*, pages 252–263, 2006.

[12] Lance Williams. Pyramidal parametrics. *Computer Graphics*, 17(3):1–11, July 1983.