Design and Implementation of Software for Assembly and Browsing of 3D Brain Atlases

Carl Gustafson[1], Oleh Tretiak[2], Louise Bertrand[1] and Jonathan Nissanov[1*]

[1]Department of Neurobiology and Anatomy, Drexel University College of Medicine, 2900 Queen Lane, Philadelphia, Pennsylvania, USA

[2]Department of Electrical and Computer Engineering, Drexel University, 3141 Chestnut Street, Philadelphia, Pennsylvania, USA

Abstract:

Visualization software for three dimensional digital brain atlases present many challenges in design and implementation. These challenges include the design of an effective human interface, management of large data sets, display speed when slicing the data set for viewing/browsing, and the display of delineated volumes of interest (VOI). We present a software design, implementation and storage architecture that addresses these issues, allowing the user to navigate through a reconstructed volume quickly and smoothly, with an easy-to-use human interface. The software (MacOStat, for use with Macintosh OS) allows the user to rapidly display slices of the digital atlas at any arbitrary slicing angle, complete with delineated VOIs. The VOIs can be assigned colors of the user's choosing. The entire atlas, or selected portions, may be resliced with slices stored as individual image files, complete with delineations. These delineations may be transferred to corresponding sections of experimental materials using our analysis program (Brain). The software may be obtained from the laboratory's web site: http://www.neuroterrain.org

Keywords:

3D brain atlas, 3D reconstruction, 3D visualization

*Corresponding Author:

Tel: 1-215-991-8410; Fax: 1-215-843-9367

Introduction

Neuroanatomical atlases are widely used to guide delineation of stained sections, assist in electrode placement, and aid in surgical planning. Numerous atlases are available [e.g. 1,2,3,4]. In general these atlases are made of a series of non-consecutive 2D images of stained sections along with graphical outlines of standard nuclei and tracts; they are usually provided as printed manuscripts. These atlases suffer two major deficiencies: they support only a limited number of orientations—typically offering coronal, sagittal and horizontal views, often incomplete (for example, in rodent atlases, the olfactory bulb is often not included in horizontal or sagittal views)—and they are sparse with substantial gaps present between sections. Three-dimensional digital atlases can overcome these deficits and a number are now available for the rat [5], human [6,7], mouse [8], rhesus monkey [10], fly [11] and other species.

To make use of these digital atlases, visualization software is required. There are generalized volume visualization applications available via both commercial (Voxblast [12], Amira [13] IDL [14]) and public (Open Visualization Data Explorer [15]) licenses; however, these are not particularly suitable for atlas display. While they do offer both voxel and vector display and do support arbitrary plane of view, all necessities in this specific setting of 3D visualization, they lack support for other important functions of atlases such as display of stereotaxic coordinates display, and textual annotations. They also fail to provide a straightforward means of matching experimental sections to equivalent atlas planes, natural navigational tools, or accommodation for the large size typical of digital atlases. We designed an application (MacOStat) that overcomes these deficits.

Design Considerations

The primary design consideration in developing MacOStat was usability. By this is meant both an intuitive human interface, with directly manipulatable interface elements, and sufficient processing speed to make live navigation possible. An intuitive interface is of critical

importance in any computer software but it is especially important in software attempting to map three dimensional data into a two dimensional display space; the third dimension must be handled in a consistent and predictable manner, and the available commands for manipulating the display should be direct and obvious to the novice user. However, there is currently no widely-accepted standard to handle these requirements. Most applications rely on a series of orthogonal views showing the projection of the slicing plane on the volume, along with a view normal to that plane [16]. This type of interface is conceptually simple - one can easily specify the slice location. Unfortunately, live navigation using this interface is not easy — an obstacle to useablity with this scheme is that manipulation is not direct — it is necessary to adjust settings in several separate windows to navigate the atlas, while monitoring the result in yet another distinct window.

Speed is the other side of the usability coin - as even moderate-resolution atlases are quite large (a 17.9μm/pixel isotropic mouse brain atlas [9], without any delineation, can easily exceed 100MB), efficient processing is critical to providing a satisfactory user experience. It is difficult, if not impossible, to accurately navigate through an atlas unless refresh rates are substantially faster than a frame per second. Handling this massive amount of data, and selecting and displaying individual slices quickly is a another primary challenge. In addition, both fine- and coarse-grained movement is required - fine grained movement alone quickly becomes tedious when traversing large distances, and coarse-grained control makes precise positioning impossible.

To meet these criteria, we developed a system that eliminates the entire orthogonal-view paradigm, relying instead on only showing a slice of the atlas for fine-grained navigation, and a wire-frame representation of the slice's location within the atlas' bounding volume for coarse-grained movement. These displays are directly manipulated - the user selects an action from a toolbar, then clicks and drags the mouse over the display; the display moves in direct response to the user. To support rapid display and efficient access to atlas data, we have developed a novel data storage technique that is generally applicable to volume (voxel) data.

A second design consideration is an architecture that facilitates maintenance and enhancement. It is inevitable that software applications will need fixes and additional features. This is especially true in relatively new content domains, such as interactive digital atlases. Unless the basic program architecture is initially well factored, and modular in nature, future attempts at enhancements will result in increasing fragility, resulting in either costly redesign, or a reluctance to add additional features.

To realize this goal, we use object-oriented design techniques, and a commercial application framework, MacApp [17]. The framework uses a Model-View-Controller (M-V-C) design pattern well-adapted to user-centric interface design, and is easily modifiable, via inheritance techniques, to our content domain. Atlas data is abstracted into generic voxel maps, with appropriate metadata handled by both customized framework and purpose-built classes, acted upon by generic slicing classes. A hierarchy of command classes abstract user interactions. Customization of framework view classes provides the display functionality. An additional advantage of object-oriented design is that it facilitates reuse of existing code bases. In our case, we have an image acquisition and analysis package (Brain [18]), developed in-house, that we use for 2D delineation of the data sets used in atlas creation. Many of the classes related to the data model in Brain have been reused in the development of the atlas browser. The reuse already developed and debugged code greatly shortens the application development cycle.

System Overview

We use an M-V-C pattern for our system. All the data structures (the model) that comprise an atlas (gray-scale data, delineation, size and calibration, metadata, etc.) are aggregated by a class descended from the MacApp base class TDocument (Figure 1). The atlas is displayed (the view) using classes derived from the MacApp base class TView (Figure 2). Finally, all interactions with the user (the controller) are handled by derivatives of the base class TCommand (Figure 3). Finally, we have

developed an additional class hierarchy independent of the MacApp framework used to handle data slicing for later display (Figure 3).

The Model

The atlas browser has two independent components — the program to construct atlas data files from separate, pre-aligned frames (Gluclose), and the browser itself (MacOStat). To the greatest extent possible, the two share the classes that implement the data model (the class hierarchy from the atlas browser is shown in Figure 1). Both also share classes with our analysis package, Brain; atlas data files are constructed from Brain image files, and the atlas browser must export resliced images in Brain file format. This means that our root data model has two independent branches, one for the atlas document and one for the image documents that can be matched to the atlas. The document objects used to construct an atlas (not shown in Figure 1) contain additional data elements beyond those required by the Brain application— for example, the position and orientation of the image within atlas space. To accommodate this, we created a subclass of the basic Brain document to hold orientation data. All document objects have methods to read and write data files when appropriate; this allows us to use existing classes, and add additional capabilities via inheritance. Part of the initialization for document classes usually includes the creation of view objects to display the document's data. Because this display is both not needed and computationally expensive when constructing an atlas from images, we further subclass the document to eliminate this view creation step.

In addition to the document classes based on code developed for Brain, we also have been able to reuse classes developed to represent delineated regions of interest and calibration curves. Added to this are 3D specific classes to represent both gray-scale and binary volume data (image voxels and volumes of interest) and metadata, as well as various collection classes used to manage these data objects, which were derived from framework collection classes.

The View

Previous work provided a view class that used an offscreen buffer for drawing; this class was used to display individual image files opened for matching purposes. A similar class was developed to handle display of the sliced atlas image. Additional classes were derived from framework classes for tool and informational views, and so forth.

Most views are subclassed directly from the base framework view class, but some intermediate classes are used, primarily in the views that can be manipulated via mouse (Figure 2). In TMouseOrientableView, the code that responds to mouse actions resides in a superclass of the final view objects, as all views that can be manipulated via mouse (wire frame coarse-positioning and slice fine-positioning views) may then use common code.

Many views are designed to be embedded in other views, to control various aspects of view behavior. For example, T3DBufferedView is the ultimate destination for sliced atlas images and the associated sliced volumes of interest. It uses an offscreen image buffer to assemble the final image prior to display. This view is nested into a view descended from TMouseOrientableView, which controls responses to mouse movement. While it is possible to derive T3DBufferedView from TVolumeView or TMouseOrientedView, we also allow multiple views to be nested in TVolumeView - for example, it is possible to browse several different data sets, but keep all data sets at the same exact slicing orientation. By separating the view classes, we can allow TVolume view to handle both tracking and scrolling of the views, and transfer information to all instances of the data model; the code to handle this if the hierarchies were combined would be significantly more complex.

The Controller

A number of command classes were derived from the framework command hierarchy to encapsulate user actions such as navigation and reslicing. One advantage to this structure is that each command object contains the information needed to undo any actions; commands may be placed on a stack for multiple levels of undo if desired. By supporting undo for almost all user actions, the human interface becomes much

more forgiving, allowing the user the confidence that any mistakes can be recovered.

In addition to the suite of command classes, a set of classes used to reslice volume data was created. In essence, there is one slicer class for each type of volume data (gray-scale and binary) and display variant (8 or 32 bit pixels, for example). These slicers, which are persistent objects, are controlled by the command objects spawned by the user's actions. In this case, much of each slicer's code is independent of the other slicers in the suite, primarily for performance reasons - the overhead of calling virtual functions (the basic mechanism allowing a subclass to provide specialized behavior) is normally negligible, but in the tight loops used in slicing, it has the potential to become significant.

File Structure

Atlas files are stored in a tagged format. Each data item in the file is preceded by a tag identifying the data, and by a length field specifying how long the field is. This structure provides several benefits; the first is position independence — each chunk of data is interpreted independent of the data around it, and is not required to appear at a particular offset in the disk file — it may be inserted anywhere in the file. The second advantage is that the file structure is effectively separated from the program's version. A file created by a later version of the atlas construction software, Gluclose, may still be read and understood by an earlier version of MacOStat, it simply skips unrecognized tags. This in turn means that development and distribution of atlas construction and browsing software can proceed independently. A common problem in many applications is the lack of forward file compatibility, this block structure eliminates the problem in all except the most extreme cases, and thus allows the developers more freedom in adding useful features.

Implementation Details

Data structures, either in the form of classes or other formalisms, make up the core of an application. A well-designed data structure must model the reality being represented and must provide a structure easily

accessible to, and supportive of, the internal program code. Object-oriented design methods (in which classes include both the data and the functions to manipulate that data) are often used to meet these requirements. Good classes present an opaque interface to the application allowing access to the data the instantiated objects maintain, while hiding the actual implementational details from the application. This data hiding results in better modularity, which allows future enhancements to be made without disrupting the existing the program code.

Macrovoxels

As an example of the value of opaque data structures, let us first consider the method we use for storing atlas image voxels. A typical method of organizing volume data is in the form of a monolithic array of voxels; we do not use this structure for reasons that will be detailed later. Instead, we group voxels into clusters we call macrovoxels, and organize these macrovoxels into a structure we refer to as a voxel map (Listing 1, 2). To access an individual voxel, one requests a specific macrovoxel by providing the voxel map with the macrovoxel's address in atlas space, and then requests the individual voxel by providing that voxel's address to the macrovoxel. A voxel value is returned. At no point does the voxel map need to understand how the individual voxel data is stored, so macrovoxels supporting binary, 8, 12, 16 or wider bit data may be stored in the voxel map. In practice, the voxel map's client needs to know the bit dimensions of the individual voxels, as it needs to set the correct pixels in the screen buffers for display, but no other part of the application needs to be aware of this implementation detail.

We use the macrovoxel structure for both performance and storage considerations. Computer memory and file systems are essentially one-dimensional systems, where the only access to an individual data item is by some index from a defined location. Mapping multi-dimensional data into memory is essentially an exercise in bookkeeping, however, locality of data is preserved only along one dimension - if the voxel at [X,Y,Z] is adjacent to the voxel at [X+1,Y,Z], and the maximum dimensions of the atlas are (m,n,o), then the distance to [X,Y,Z+1] is m x n voxels away. This is an important consideration when accessing large data sets for two

reasons. First, modern processors use a cache to provide quick access to recently used data, and nearby data - a fixed amount of data is moved into a cache (a cache line), on the first access, and after this, the cached copy of the data is accessed, rather than the data in RAM. Cache memory access is much faster. If the data to be accessed during slicing is localized, most of it can be loaded into cache, and processing speed enhanced. If it isn't cached, then the "cache hit rate" drops, and processing slows down. Second, modern systems utilize a virtual memory manager (VMM), which trades disk storage which is slow but capacious, for RAM memory, which is more limited in capacity but much faster. When a particular memory location that has been paged to disk is required, the VMM brings it into RAM memory. This is a slow process, and so it is desirable to minimize these page loads.

The macrovoxel architecture reduces some of the drawbacks of large monolithic arrays. We group nearby voxels into a chunk, typically 16 voxels on edge. These chunks are then arranged into a structure we call a voxel map (Figure 4). The voxel map comprises the atlas data, and is the only entity to which the document class needs to maintain a reference. By grouping voxels into these macrovoxels, we now have a data block that is roughly the size of a VMM page, and also fits into a small number of cache lines. Thus, any voxels (row, column, plane) that are spatially adjacent are now also physically adjacent in memory, which improves memory access performance. Consider an attempt to reslice an atlas: if the slicing plane is coincident with the data set (i.e.. examining only a single plane of data in an array) then only the data to be displayed, plus possibly a cache line and/or VMM page frame on either side of the data plane need to be moved. In this case, slicing the monolithic array will be faster than a macrovoxel array, as in the latter 16 planes (based on a 16-voxel on edge macrovoxel) will need to be moved. If, however the slicing plane is perpendicular to the data set major axis (cuts across many array planes), then given the nature of the array, much more data will need to be moved for the monolithic array than for a macrovoxel array (Figure 5).

An additional advantage of the macrovoxel architecture is that it allows us to eliminate empty voxels from storage. By empty voxels we mean those

voxels that exist outside the contour of the 3D image we are representing. Biological images are rarely in the form of cubes or prisms, but that is how most arrays (including macrovoxel arrays) represent data - this representation means that location information may be computed given the three dimensions of the array, rather than stored in some fashion. The result is a reasonably compact data representation. This fails, however, if the object being represented deviates significantly from a prism. In our domain, brain imaging, the subject matter more closely approaches an ovoid. This means that the voxels in the corner of the atlas volume have no useful data in them. By breaking our atlas space up into macrovoxels, we can identify macrovoxels that contain these empty voxels, and replace that macrovoxel's entry in the volume map with a reference to a single empty or "white" macrovoxel. This allows us to eliminate large amounts of empty space, resulting in a reduced disk and memory footprint, with a savings of typically 30%. (Figure 6)

Slicing

The actual slicing (Listing 3) is handled by taking the corners of the virtual knife (the plane to be displayed on screen), and converting from an atlas-based coordinate system, where the axis units are arbitrarily defined relative to the atlas geometry, to a coordinate system where the units are based on macrovoxel indexes. Atlas-based coordinates are centered in atlas space, which facilitates the rotation and translation of the virtual knife, while macrovoxel-based coordinates are based on the upper front left macrovoxel, to facilitate memory accesses. This conversion results in the useful property that the macrovoxel containing any given voxel is specified by the whole number portion of the voxel's coordinate. The fractional part is then scaled by the macrovoxel size (usually 16 voxels) to give the exact voxel. This scaling thus allows voxel address computation almost as efficiently as if a monolithic array were to be used.

Once this scaling is completed, we divide opposite edges of the virtual knife into segments corresponding to one of the dimensions of the display space. From each of these segments, we project transect vectors to the opposite side of the virtual knife, and divide these into segments

corresponding to the other dimension of display space. Each of these segments now can be mapped directly to a voxel in our macrovoxel array, and the image buffer is populated by iterating over the segments on the virtual knife (Figure 7).

Finally, we provide a class to specify the virtual knife location. Any changes to slicing angle or position are forwarded to this class, to which the slicing classes refers. Using this mechanism, the slicing classes need no knowledge of the basic document objects - they only need to know about the data set to be sliced, the screen buffer the sliced image is written to, and the virtual knife location.

Atlas construction

Atlas construction (via Gluclose) is essentially the reverse of atlas slicing. First, an array of macrovoxels is created. Here, the entire array needs to be allocated, as the final population will not be known until construction is complete. Each pre-aligned frame comprising the data set is loaded, it's location in atlas space determined either by data included with that file or by it's position in the list of frames, and it's pixels converted to voxels and written to the macrovoxel collection. During this process, gray values are equalized or remapped based on any included calibration curves. Once all frames have been loaded, the collection of macrovoxels is checked for data content, and empty macrovoxels eliminated. The completed data set is then written to disk. Delineations, in the form of outlines, are accumulated and handled in a similar fashion. The final calibration curve, a table of VOI names and display data, and coordinate system transformation data are also written to file.

Human Interface

The human interface of MacOStat provides the user with a view of the data set at the current virtual knife position and any associated delineation, a schematic view of atlas space showing the position of the virtual knife, and the necessary controls and menus to control the application (Figures 8,9).

Using a control strip,(a utility window with icons representing commands) the user can switch basic slicing axis or presentation: coronal, sagittal, or horizontal. Rotation and translation of the virtual knife is also controlled by this strip.

Navigation through atlas space is done by translating and rotating the virtual knife. These actions are carried out via mouse movements: The user selects the action desired from the navigation control strip, then clicks and drags the mouse over either the view of the atlas slice, or over the wire frame diagram of the virtual knife and atlas space. Dragging over the wire frame provides coarse movement control, and the slice view provides fine movement control. In both cases, the display content changes to provide direct feedback to the user.

In addition, the rotation and translation of the virtual knife is provided in the upper left corner of the wire frame window. In addition, a set of text entry boxes are provided to allow the user to specify these values numerically. Finally, a facility is provided to remember particular virtual knife orientations, and recall them via a popup menu. These saved positions can may be written to file, and loaded into memory.

Dragging the mouse over the atlas slice display itself also displays the X, Y and Z coordinates of the mouse, expressed in atlas units. Data needed for conversion of atlas coordinates to stereotaxic coordinates can be embedded in the atlas data files, and will be used if available. In this case, the mouse pointer location is expressed in stereotaxic coordinates rather than the atlas-relative coordinates.

Atlas data sets may also contain delineation data; this data is shown either as a colored outline on the gray-scale image, or as a translucent colored area. Colors are user-selectable. In addition, the user may turn the delineation display on or off, and may define a subset of individual structures for display (Figure 9). Display lists complete with color specifications may be saved to a file, and reloaded as desired. The file itself is in the form of tab-delimited text, and so may be viewed and edited with standard word processing and spreadsheet applications. Each structure is specified by both an abbreviation and a name.

Finally, it is possible to save sequential virtual slices to individual data files. To save these slices, the starting and stopping positions are marked, and the number of slices and manner in which the virtual knife position is interpolated is specified. For example, if the starting and stopping slices are not parallel, one can specify that only the starting or stopping knife angle should be used, the mean angle, or that the angle should be interpolated across the span. Using this facility, the user can open individual experimental sections and match them to the 3D atlas to rapidly generate a set of matching planes that can then be employed in delineation of the experimental data using Brain.

Software Availability

The software (MacOStat) may be downloaded without fee from www.neuroterrain.org. It requires MacOS 8.6 and CarbonLib 1.3 or later.

Acknowledgment

This work was supported by NIH award P20 MH62009 and US Air Force agreement F30602-00-2-0501

References

[1] Franklin, B. J. Keith, G. Paxinos, The Mouse Brain in Stereotaxic Coordinates. Academic Press, San Diego, California 1997

[2] Paxinos, G., C. Watson, The Rat Brain in Stereotaxic Coordinates. Academic Press, San Diego, California 1998

[3] Swanson, L. W., Brain Maps: Structure of the Rat Brain. Elsevier, Amsterdam, Netherlands, 1992

[4] Mai, J. K., Assheuer, J., G. Paxinos, Atlas of the Human Brain. Academic Press, San Diego, California 1997

[5] Toga, A. W., E. M. Santori, R. Hazani, K. Ambach. Rat Atlas Image Database, http://www.loni.ucla.edu/Research_Loni/atlases/rat/

[6] Sundsten, J W. Digital Anatomist: Interactive Brain Atlas, http://www9.biostr.washington.edu/da.html

[7] Kikinis, R. A DIGITAL BRAIN ATLAS FOR SURGICAL PLANNING, MODEL DRIVEN SEGMENTATION AND TEACHING, http://www.spl.harvard.edu:8000/pages/papers/atlas/text.html

[8] Sidman, R. L., B. Kosaras, B. Misra, S. Senft. High Resolution Mouse Brain Atlas, http://www.hms.harvard.edu/research/brain/

[9] Bertrand, L., J. Nissanov, 3D Atlas of the Mouse Brain, Computer Vision Laboratory for Vertebrate Brain Mapping, Philadelphia, 2001. http://www.neuroterrain.org/

[10] Jones, E G. et al, (Resus atlas) UC Davis/UC San Diego Human Brain Project, http://neuroscience.ucdavis.edu/hbp/project2.html

[11] Flybrain, http://flybrain.neurobio.arizona.edu/Flybrain/html/

[12] VoxBlast http://www.vaytek.com/VoxBlast.html (commercial)

[13] Amira -- visualization and reconstruction for 3D image data. http://www.amiravis.com (Commercial)

[14] IDL -- data analysis, visualization and application development. http://www.rsinc.com/ (Commercial)

[15] Open Visualization Data Explorer -- an application and development software package for visualizing 2D/3D data. http://www.research.ibm.com/dx/ (IBM Public License)

[16] Lohmann, K., Gundelfinger, E. D., Scheich, H., Grimm, R., Tischmeyer, W., Richter, K., Hess, A. (1998) BrainView: a computer program for reconstruction and interactive visualization of 3D data sets. *J Neurosci Methods* 84(1-2) 143-154.

[17] Apple Computer, Inc., MacApp, http://developer.apple.com/tools/macapp/

[18] Nissanov, J., D.L. McEachron. 1991. Advances in image processing for autoradiography. J. Chem. Neuroanat. 4:329-342.

Figures

Figure 1: Data model class hierarchy and interactions

Figure 2: View class hierarchy

Figure 3: Controller class hierarchy and interactions

Figure 4: Arrangement of microvoxels into macrovoxels, and macrovoxels into an atlas.

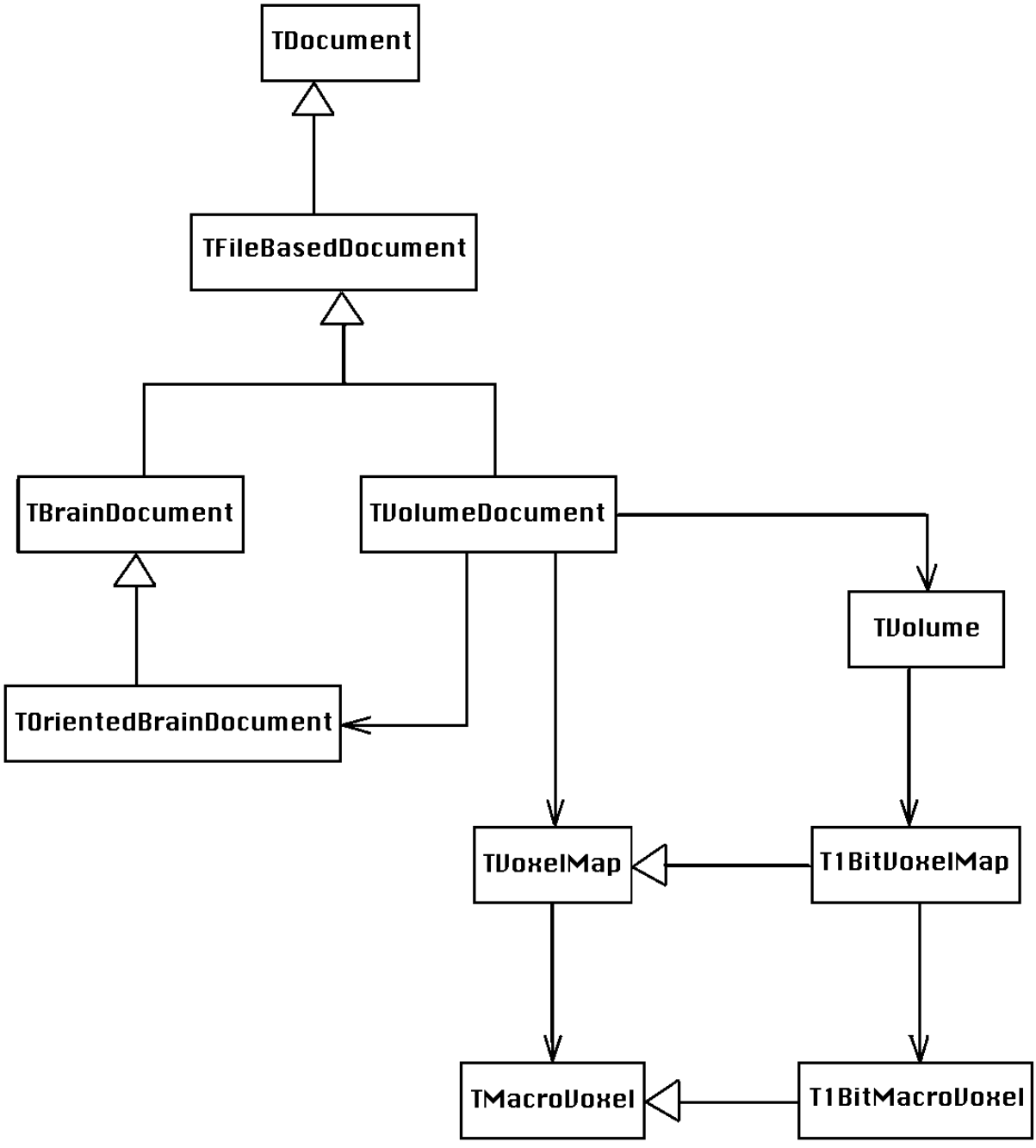
Figure 5. Memory/cache accesses required for (a) macrovoxels vs. (b) monolithic array design.

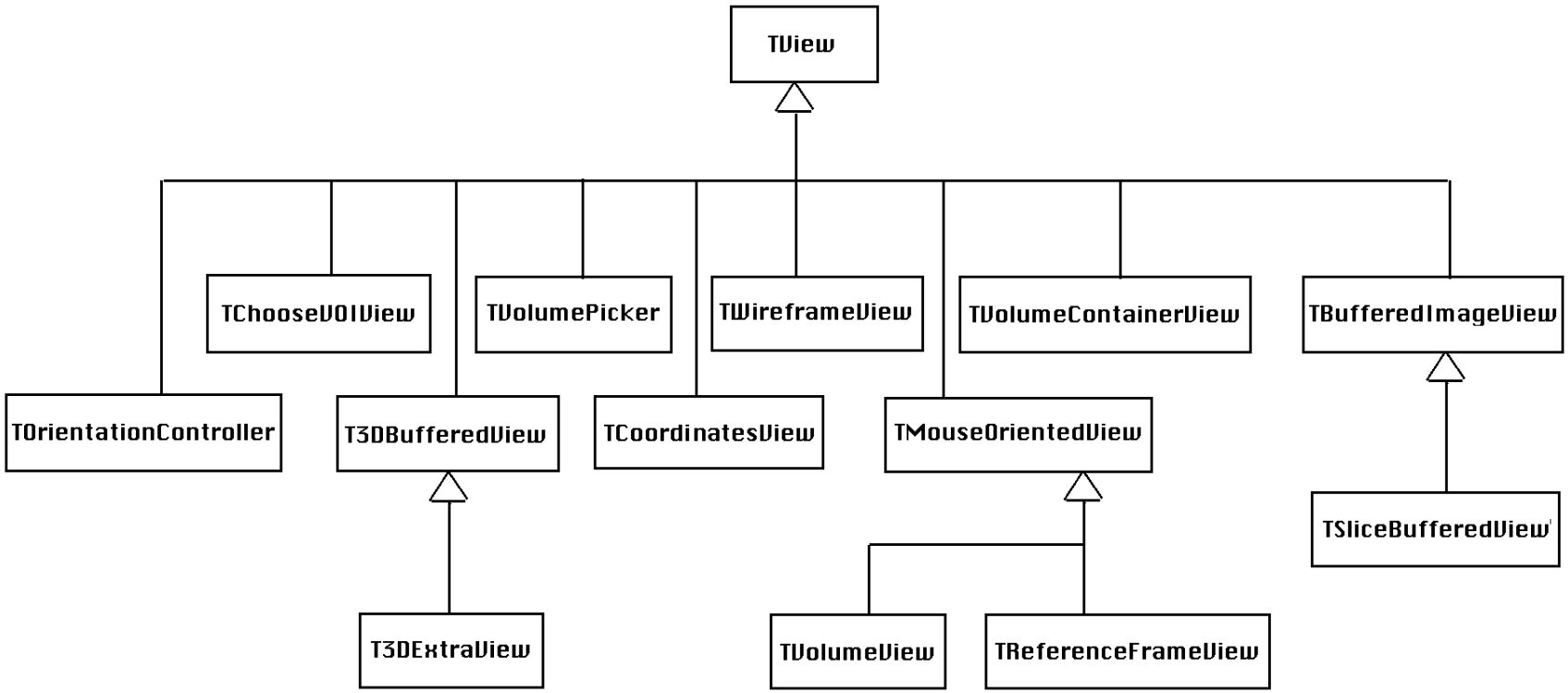
Figure 6. Actual space occupied by image data within the atlas boundary.

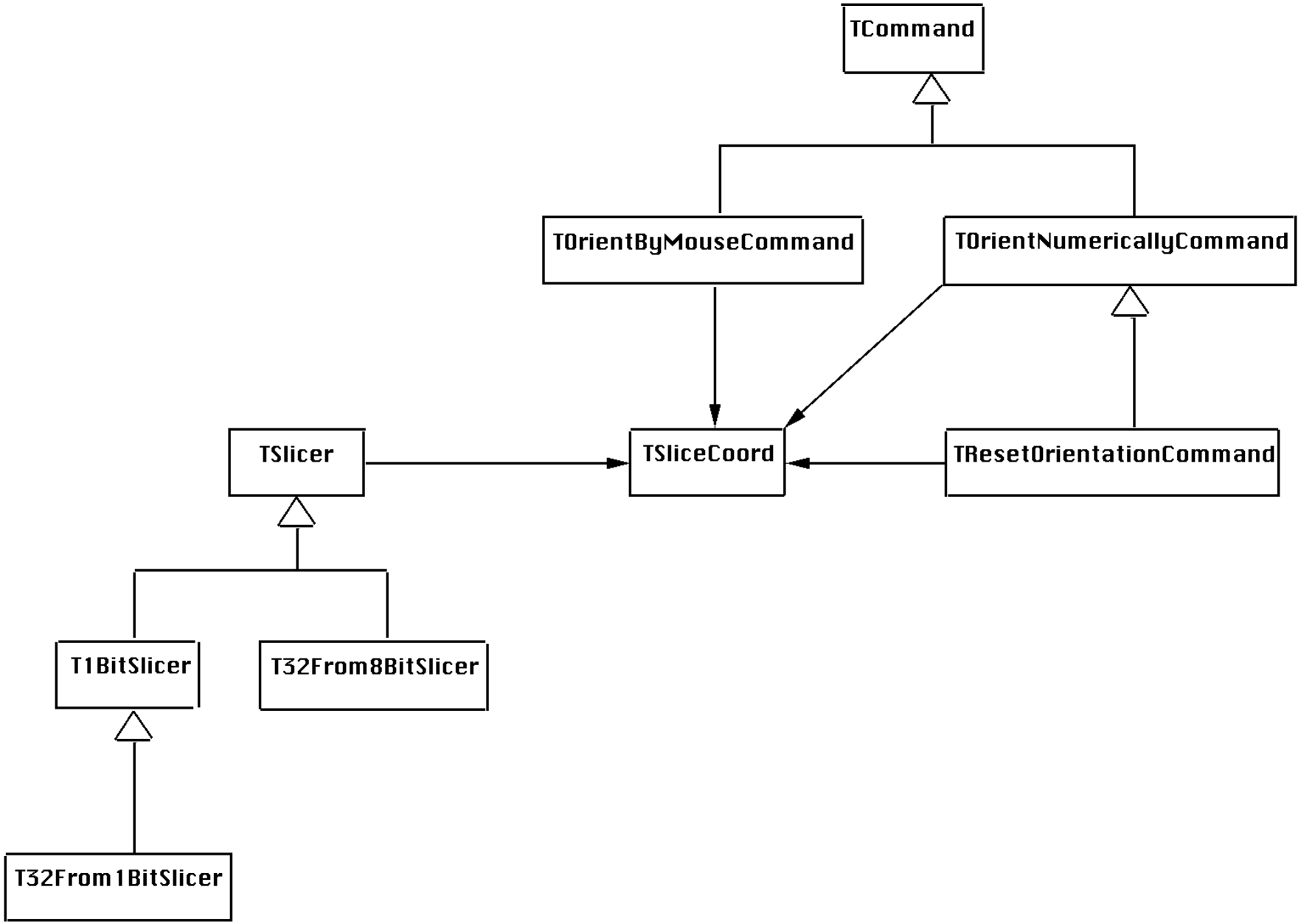
Figure 7. Atlas reslicing. Transect vectors in the slicing plane (virtual knife) are used to select microvoxels for display.

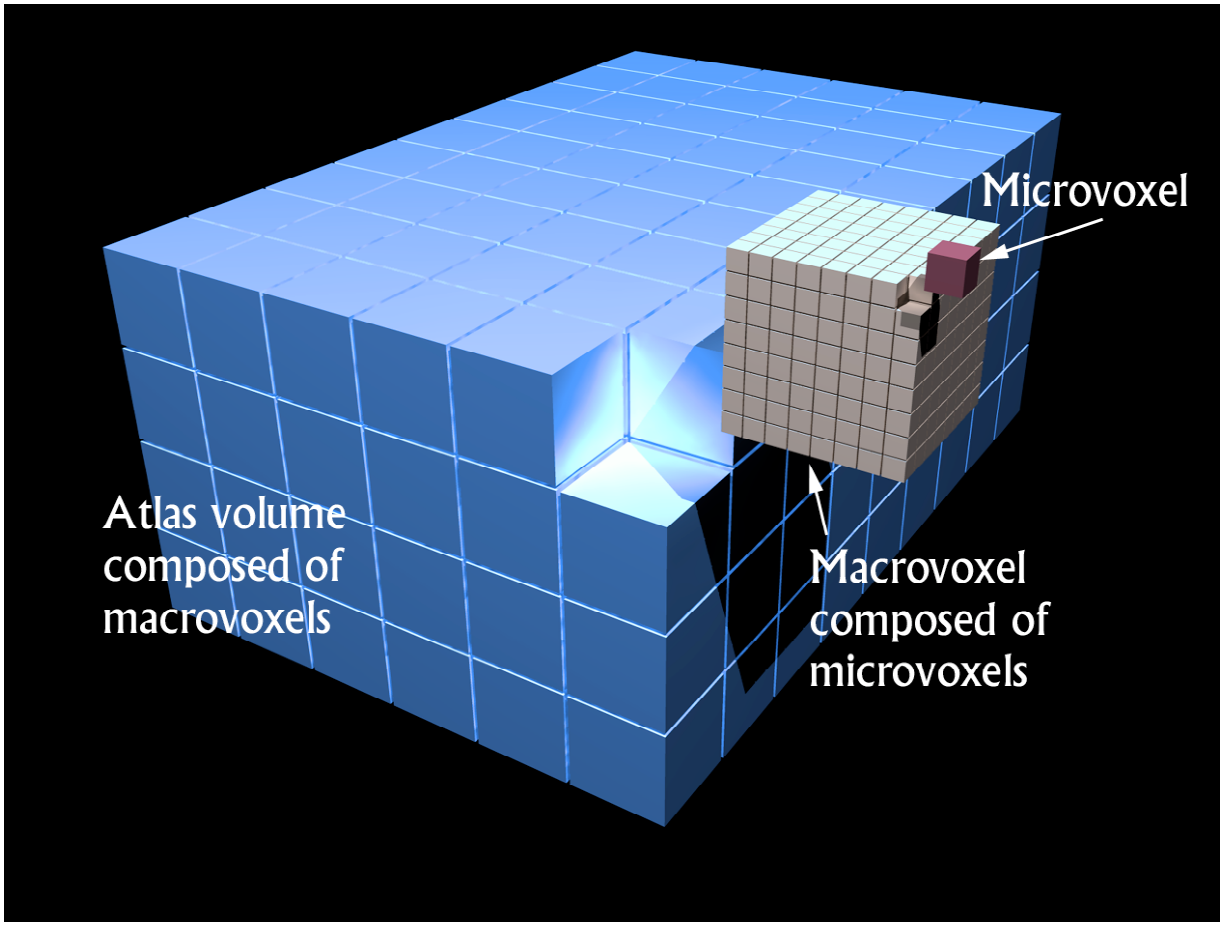
Figure 8. Navigation and control palettes

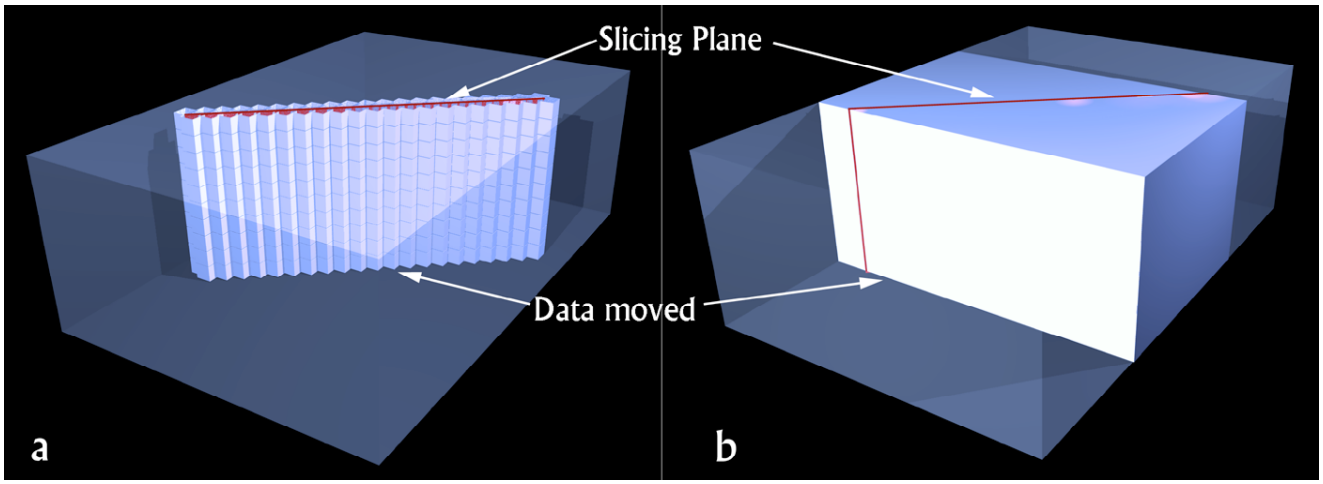
Figure 9. Horizontal, coronal, and saggital slices, with shaded VOIs, and VOI selection dialog, taken from our mouse atlas [9]

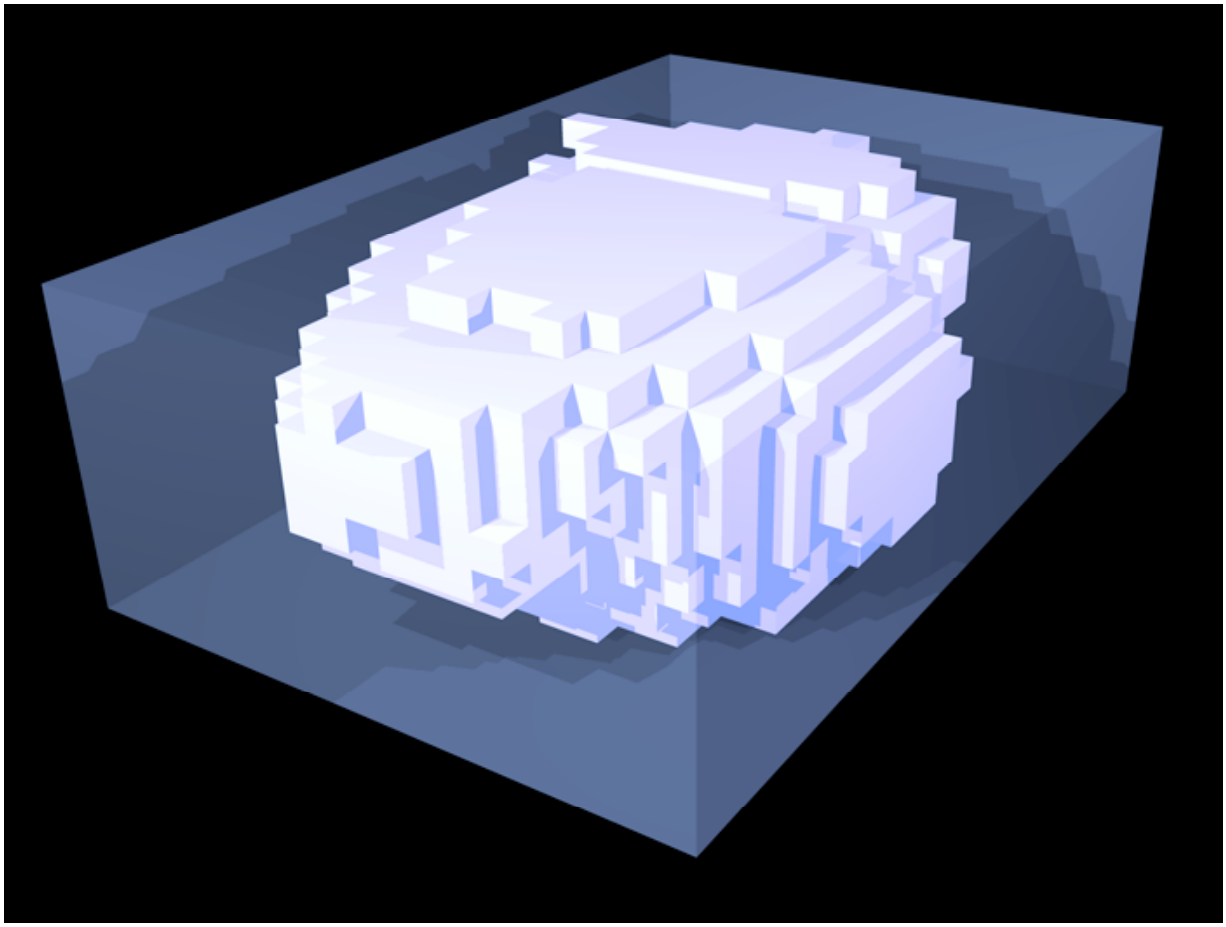

TDocument
TFileBasedDocument
TBrainDocument
TVolumeDocument
TVolume
TOrientedBrainDocument
TVoxelMap
T1BitVoxelMap
TMacroVoxel
T1BitMacroVoxel

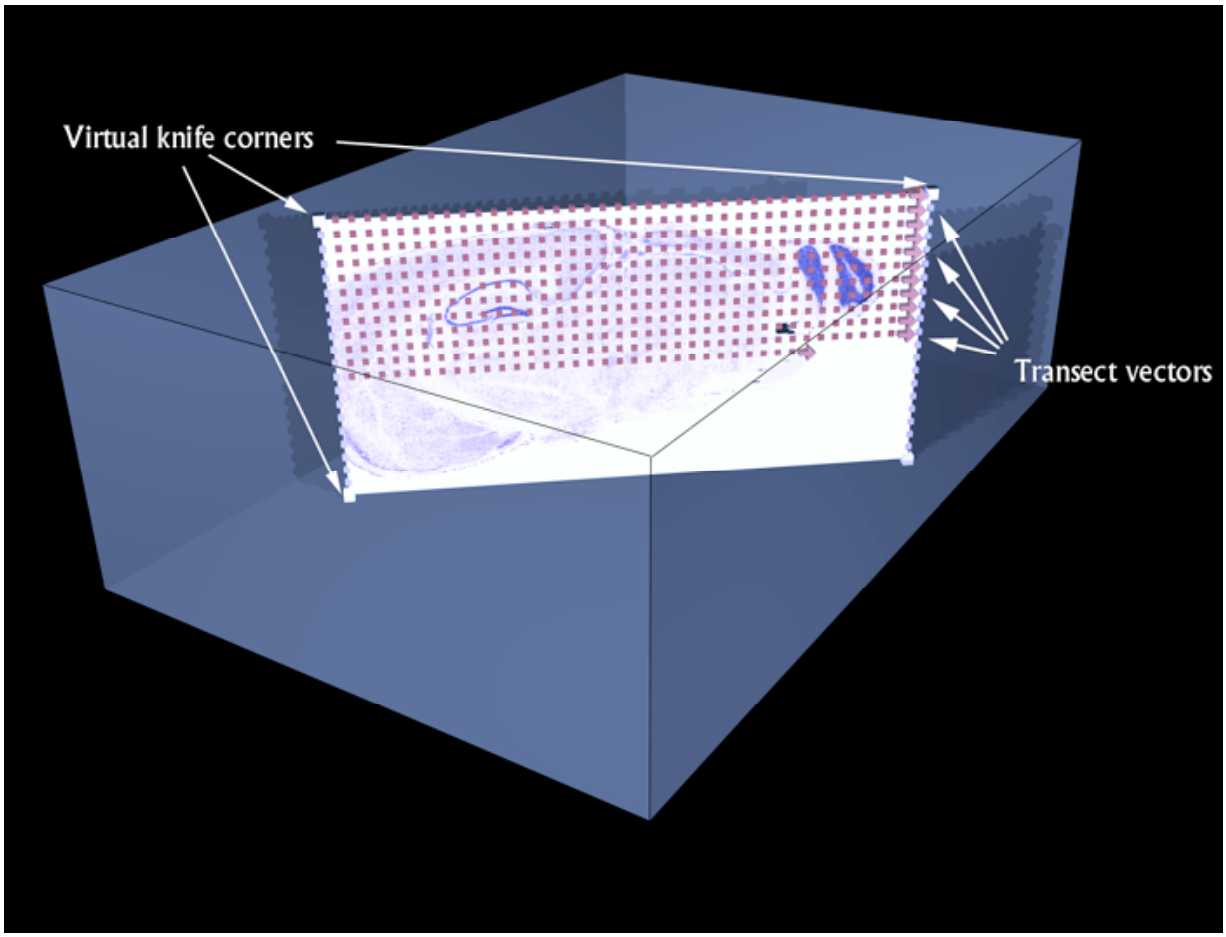

TView
TChooseVOIView
TVolumePicker
TWireframeView
TVolumeContainerView
TBufferedImageView
TOrientationController
T3DBufferedView
TCoordinatesView
TMouseOrientedView
TSliceBufferedView
T3DExtraView
TVolumeView
TReferenceFrameView

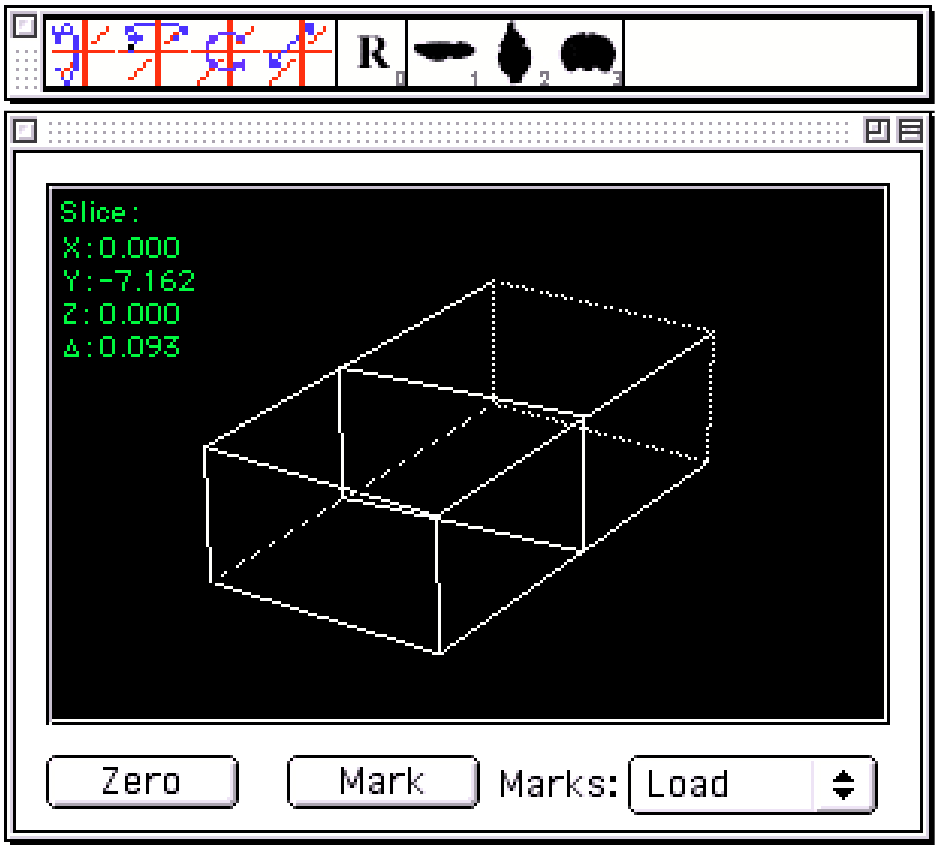

TCommand
TOrientByMouseCommand
TOrientNumericallyCommand
TSlicer
TSliceCoord
TResetOrientationCommand
T1BitSlicer
T32From8BitSlicer
T32From1BitSlicer

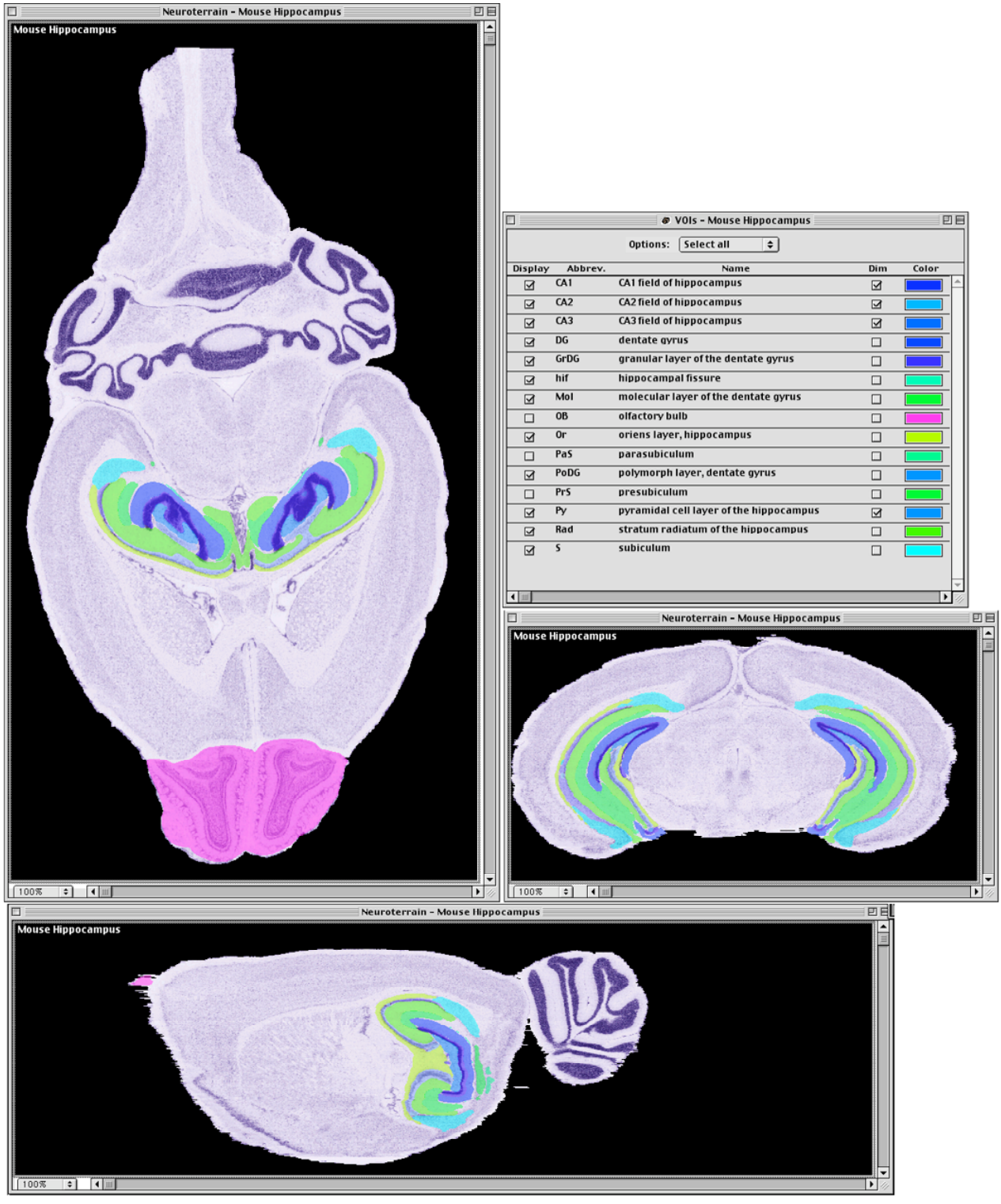

Microvoxel
Atlas volume composed of macrovoxels
Macrovoxel composed of microvoxels


Slicing Plane
Data moved
a
b




Virtual knife corners
Transect vectors


R
Slice:
X:0.000
Y:-7.162
Z:0.000
Δ:0.093
Zero
Mark
Marks:
Load


Neuroterrain - Mouse Hippocampus
Mouse Hippocampus
100%
VOIs - Mouse Hippocampus
Options:
Select all
Display
Abbrev.
Name
Dim
Color
CA1
CA1 field of hippocampus
CA2
CA2 field of hippocampus
CA3
CA3 field of hippocampus
DG
dentate gyrus
GrDG
granular layer of the dentate gyrus
hif
hippocampal fissure
Mol
molecular layer of the dentate gyrus
OB
olfactory bulb
Or
oriens layer, hippocampus
PaS
parasubiculum
PoDG
polymorph layer, dentate gyrus
PrS
presubiculum
Py
pyramidal cell layer of the hippocampus
Rad
stratum radiatum of the hippocampus
S
subiculum
Neuroterrain - Mouse Hippocampus
Mouse Hippocampus
100%
Neuroterrain - Mouse Hippocampus
Mouse Hippocampus
100%