Stereoscopic Visualization of Medical Data Sets

Thomas E. Hansen and John M. Zelle, Ph.D
Math, Computer Science, and Physics
Wartburg College
Waverly, IA 50677
thomas.hansen@wartburg.edu
john.zelle@wartburg.edu

Abstract

Recent initiatives in low-cost virtual reality (VR) have made the deployment of classroom VR applications feasible in virtually any educational setting[8]. This paper reports on our experience using freely available tools to construct stereoscopic visualizations of medical datasets such as CT scans and magnetic resonance imaging (MRI). Our initial investigations have centered on functional magnetic resonance imaging (fMRI) data. We present effective methods for visualizing this information through interactive and/or stereographic software implementations using VTK[4, 5], an open source scientific visualization package, FSL[6], a freely available fMRI analysis program, and Python[3], an open source scripting language. These ideas are incorporated into a turn-key application, PyMed3D, that facilitates the creation of anatomical visualizations suitable for use in the biology classroom.

Introduction

Virtual reality (VR) technologies are becoming increasingly popular in diverse arenas from scientific data analysis to entertainment. An important facet of this technology is the stereographic display of computer images to produce three-dimensional visual effects. We are interested in bringing aspects of virtual reality, primarily true 3D visualization, into the classroom. The possible uses of VR in education are nearly unlimited. With VR, students can explore otherwise inaccessible environments such as the surface of Mars or visualize abstractions like molecular models and magnetic fields. From the perspective of computer science education, VR offers a very attractive arena of application projects for students to tackle.

Recent initiatives in low-cost VR have made the development of such classroom applications feasible in virtually any educational setting[8]. Such systems use passive stereo display techniques with readily available and relatively inexpensive technology to produce a multi-viewer 3D experience. Although stereoscopic viewing in the classroom is now possible, there are, at the moment, few educational applications compatible with such a setup. The goal of our project is to develop educational applications for viewing medical datasets such as CT scans and MRIs.

Our initial investigations have centered on visualization of functional magnetic resonance imaging (fMRI) data. FMRI makes possible the mapping of hemodynamic changes in the human brain during mental activity. These "active region" mappings allow for noninvasive observations of brain activity, which can be used to gain understanding about the roles of specific parts of the brain, such as the visual cortex.

Typically, fMRI data is presented by showing 2D images that portray "slices" of the brain with regions of activity color-coded. The slices can also be "stacked" to construct a 3D representation. We have extended that idea a step farther to present the 3D model stereoscopically for a true 3D viewing effect. Building on free and open software toolkits, we have produced a software environment for stereoscopic visualization of fMRI, and similar volume data.

Compared to the traditional 2D image slice presentation of MRI data, we believe the interactive, stereographic representation will prove itself more accessible to novice neurobiology students and laypeople. With this effective way of viualizing fMRI data, we hope to provide a valuable teaching aid aimed at the neurobiology classroom.

The rest of this paper outlines our approach. This work may be of interest to those who either want to make use of the software that we have developed or use similar techniques to develop their own visualization applications.

Visualizing Medical Data

Form of Data

In order to create a stereoscopic visualization from medical data, the data must be transformed into a 3 dimensional representation. The first step is the aqusition of digitial images representing "slices" of the portion of the body being studied. MRI equipment uses strong magnetic fields and pulsed radio waves to record data. Raw MRI data is an array of numbers representing the relaxation of nuclear spin magnetization, which are ultimately represented in a series of slice images after the application of mathematical transformations that result in a gray value being assigned for each pixel of each slice[4]. An example series of slices is shown in Figure 1.

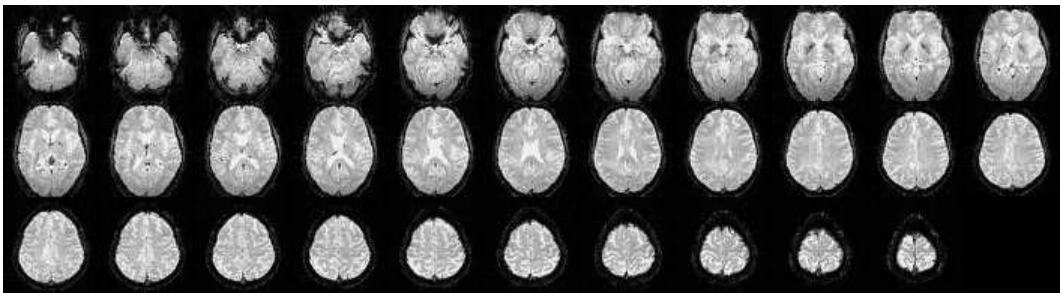

Figure 1: fMRI data: A lowresolution MRI scan aquired as part of a functional MRI scan. The series of images represent slices through a 3 dimensional volume.

A series of slice images can be easily interpreted by trained radiologists, but it not so easily digested by novies. If these slices are "stacked" up, they can be considered as a three-dimensional volume. This stack of images can then be used to produce three-dimensional respresenations for a more natural visualization.

In the case of fMRIs, an initial high–resolution scan of the brain is followed by a series of lower resolution scans. These lower resolution scans are then analyzed to produce "active region" maps within the initial scan. Each of these active region mappings can be treated as a separate 3D volume that is positioned somewhere inside of the original full brain volume.

3D Visualization Techniques

Two distinct techniques are available for visualizing volume data: surface extraction, and volume rendering (see Figure 2). The surface extraction approach attempts to model tissue boundaries within the volume using the polygonal meshes frequently used in computer graphics. Surface extraction or Contouring is done by seperating an image into distinct regions based on similar scalar values, an isosurface. By carefully specifying a scalar value range, contouring can be used to extract specific isosurfaces, for example outlining bones in a CT dataset.

The classical method for volume rendering involves ray casting. A ray is cast through each pixel of the display and its path through the volume is traced. A compositing function is used to “summarize” what the ray encounters as it passes through the volume, and this function is used to color the pixel. Basic ray casting techniques and algorithms are conceptually much simpler, than the algorithms involved in surface extraction. The accuracy and detail achieved by volume rendering techniques make it favorable for medical purposes, where physicians need to produce diagnoses based on the visualization[4]. Unfortunately, the flexibility and detail of ray tracing comes with a hefty drawback when it comes to rendering speeds.

(a) (b)

Figure 2: Example visualizations. (a) was produced by Surface Extraction. (b) was produced by Volume Rendering.

There are tradeoffs between these two visualization approaches. Extracting isosurfaces can be very time consuming, especially when the input data are large. On the other hand, once the isosurface is extracted, the hard work is done, and the resulting polygonal representation can be rendered at interactive speeds using common graphics harware.

Volume rendering has no such computational intensive initialization; instead each frame rendered requires approximately the same computational effort. Unfortunatly, this ”per-frame” rendering results in visualizations that are too slow for interactive presentation on stock hardware. While hardware acceleration for volume rendering is available, it is both sparse and very expensive.

Given these observations and our goal of creating interactive stereoscopic visualizations, we have focused on producing visualizations that utilize surface extraction tecniques, rather than volume rendering. In experimentation with the two techniques we also discovered that the polygonal meshes visualize better in stereoscopic environments, which further strengthens our bias towards the use of surface extraction.

Using the VTK

Platform/Toolchain Considerations

Figure 3 depicts the tools we are using and the overall structure of our system, *PyMed3D*. The heart of our software platform is Kitware's open source *Visualization Toolkit*, or *VTK*[4, 5]. As described on the VTK website:

> The Visualization ToolKit (VTK) is an open source, freely available software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers around the world. VTK consists of a C++ class library, and several interpreted interface layers including Tcl/Tk, Java, and Python.

The VTK is cross platform, but all of our experiments have been done under Linux (Kubuntu 5.10) using the KDE desktop environment.

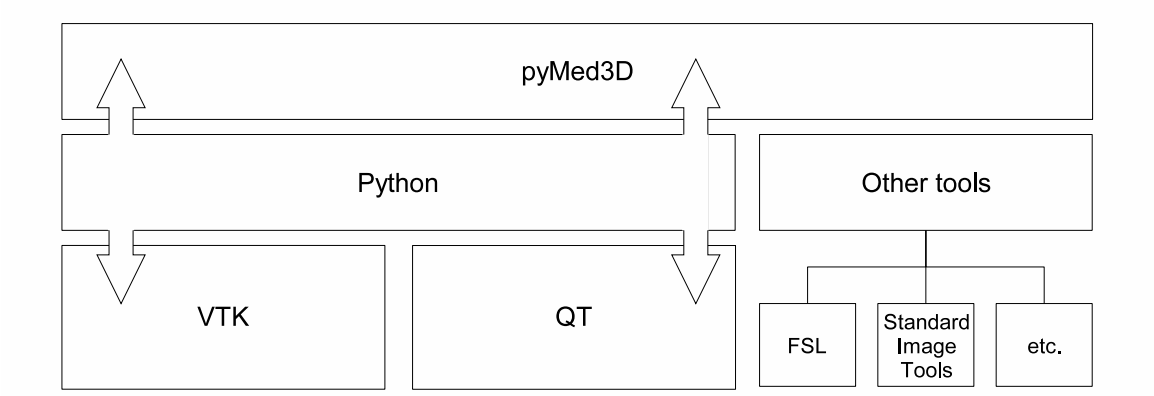

Figure 3: Software foundation for PyMed3D

The VTK is particularly attractive because of its Python bindings. Python is a very high-level, object-oriented open source scripting language that is frequently used in scientific computing contexts. Python is our main development language, and its use as a glue language has been integral to the success of this project. For visual presentation, we use KDE's native GUI toolkit, QT. A nice set of QT bindings exists for Python (PyQT [2]), which allows us to build complete QT/VTK applications quickly and easily in Python.

Before using the VTK to produce any specific visualizations, it is important to make sure that the data is present in a VTK compatible format. The VTK can read various image formats, such as DICOM, JPEG, PNG, BMP, TIFF as well as other, more specialized formats[7]. However, not all formats are supported. For example, our initial fMRI Data Set, acquired from Mark Dow at the University of Oregon, was created using FSL, a library of image analysis and statistical tools for fMRI, MRI, and DTI brain imaging data[6]. The image information we needed was stored in NIfTI format[1]. NIfTI is an extension of the

Analyze format specifically designed for functional MRI data. As the VTK has no class for reading NIfTI files, another way of getting the data into the VTK is needed.

Although extending the VTK with an appropriate NIfTI–reader would have be an interesting project, we avoided having to do this by using another tool. The FSL awutils package includes a utility `nifti2ascii` that converts NIfTI files into a simple ASCII format. The resulting ASCII files are very similar to the common Portable Pixmap (PPM) format, and it becomes a matter of writing a short 10–20 line Python script to convert the data into whatever common file format is needed/desired.

Visualization Pipeline

Visualization can be thought of in general as the transformation of data into various graphical forms that represent the original data. This transformation process will be unique for different visualizations, but oftentimes similar steps will be performed as part of the process. A functional model, to cope with this inherent modularity, is at the base of visualization with the VTK; it is referred to as the *visualization pipeline*[4].

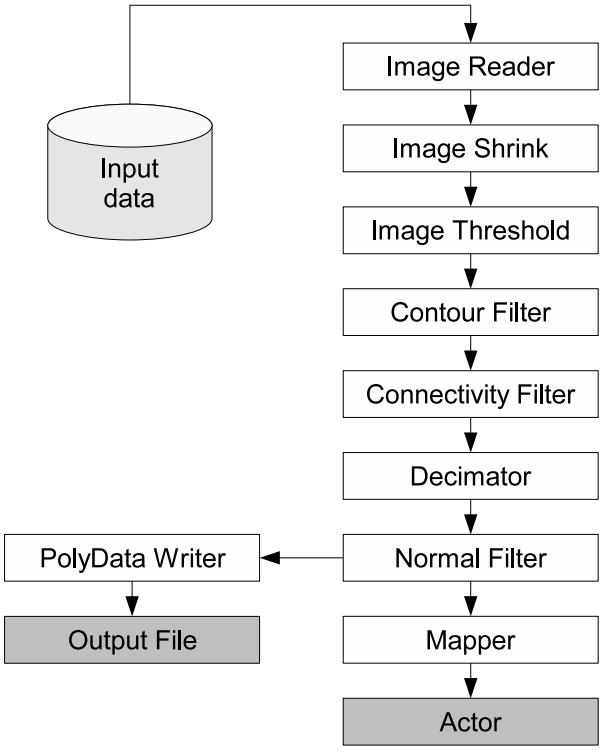

Figure 4: A Surface Extraction Pipeline

A visualization pipeline consists of data objects, process objects, and an indicated direction of data flow—the former two of which can be classified as pipeline stages. Process objects can further be divided into three subcategories: Sources, Filters, and Mappers/Sinks. We can think of the original data as a data object. To create a visualization, a careful selection

of pipeline stages needs to be connected; starting with a Data object, the data flow will lead through a series of filters, before it finalizes as it reaches the mapper which converts the data into a geometric visualization. Figure 4, shows an example of a visualization pipeline; here the data is transformed by a series of filters that ultimately create a polygonal mesh—displayed either as an actor on screen or saved to a file.

VTK API: Programming with the VTK

The various language bindings (Java, Tcl, Python) provided by the VTK make programming it accesible to programmers with little C++ experience. Especially the very object oriented design throughout the VTK API, makes translating other C++ code (e.g. in the VTK Documentation) to the supported language of choice almost trivial. We shall demonstrate the use of the VTK API by means a surface extraction pipeline as that in Figure 4 in our favorite language, Python.

The first step is setting up an apropriate reader to read the input data. In our case, we converted the fMRI to sets of PNG image files. Creating an instance of `vtkPNGReader` is rather straightforward:

```
#create and configure an instance of vtkPNGReader
reader =  vtk.vtkPNGReader()
reader.SetFilePrefix("brain")
reader.SetFilePattern("%s%03i.png")
reader.SetDataExtent(0,199,0,263,51,212)
reader.SetDataSpacing(1, 1, 1)
```

The first two lines after the call to the vtkPNGReader initialization configures the reader to read the relevant files; in this case the files are located in the same directory as the script and are named brain000.png through brain255.png. The `SetFilePattern` member function takes a format string as an argument to determine what files will be read. In this case `"%s%03i.png"` represents the 3 digit labeling ending in ”.png” after the prefix ”brain” set with the `SetFilePrefix` member function.

The member functions `SetDataExtent` and `SetDataSpacing` provide the reader with further required information. The data extent specifies what parts of the images actually need to be read. This code then, will make the reader consider slice 51 through 212, which are images of resolution 199x263. The data spacing, needs to be changed, if the spacing from one slice to the next is different from the spacing of one pixel to the next. For example, if only every second image was considered, the data spacing would need to be set to (1,1,2). With our data being spaced equally over all dimensions, this step was straightforward; another dataset we have worked with involved images of a neuron aquired through confocal microscopy in which the data spacing played a much more vital role.

After setting up the reader, pipeline creation becomes a matter of selecting the appropriate pipeline stages and connecting them. Connecting pipeline stages is achieved by passing objects returned by one stage’s `GetOutput` method as an argument to another stage’s

`SetInput` Method. For example, this code hooks an image-shrink filter to the output of the reader:

```
#connect the reader to an instance of vtkImageShrink3D
shrink = vtk.vtkImageShrink3D()
shrink.SetInput(reader.GetOutput())
```

The object returned by a `GetOutput` method can also be stored in a variable. This can be useful, when writing the programs, as it allows one to quickly add and exlude certain stages by simply commenting in or out the variable assignments.

```
#connect the reader to an instance of vtkImageShrink3D
pipe = reader.GetOutput()
shrink = vtk.vtkImageShrink3D()
shrink.SetInput(pipe)
```

We shall now briefly examine the other pipeline stages outlined in Figure 4 in regard to their purpose and functionality. The python code setting up the entire pipeline can be found in Appendix A. Obviously, the reference documents for VTK provide more detailed descriptions [7].

vtkImageShrink3D This can be used to reduce the size of initial data to make visualization and rendering faster, albeit with reduced detail.

vtkImageThreshold This is used to manipulate the intensity values in the original image. Through thresholding, areas of similar scalar values can all be clamped to a specific value to improve subsequent contour extraction.

vtkContourFilter This is the filter that generates an isosurface. The underlying algorithm used by a vtkContourFilter instance can vary, but most likely some implementation of the MarchingCubes algorithm will be used. Careful selection of a value for the contour results in surfaces representing different tissues in the image.

vtkPolyDataConnectivityFilter This filter is used for selecting among various contoured regions. For example, the “whole brain” can be extracted by taking the largest connected region.

vtkDecimatePro The decimator is a filter that reduces the number of polygons used to represent a surface, while still maintaining a good approximation to the initial shape. Fewer polygons results in faster visualization and rendering, leading to better interactive behavior.

vtkPolyDataNormals This filter recomputes the normal vectors for each polygon in the mesh. These normals are used to do shading in the visualization.

vtkPolyDataWriter This is used to save the resulting surface(s) to a file for later viewing.

vtkPolyDataMapper This turns the polygonal data into graphics primitives that can be displayed.

Automating the Pipeline: PyMed3D

Although it is relatively easy for an informed programmer to put together a pipeline script, it is not a trivial task. In order to facilitate the creation of visualizations by non-experts, we have developed PyMed3D. PyMed3D is a Python program that provides a graphical user interface for working with datasets such as those discussed in earlier sections. PyMed3D lets users set up pre–defined visualization pipelines, allows real–time modifications to the pipeline, and provides convenient I/O capabilities. By providing a graphical user interface and approriate documentation, we hope to make the creation and use of medical, stereographic visualization accessible to non-programmers, for example, biologists.

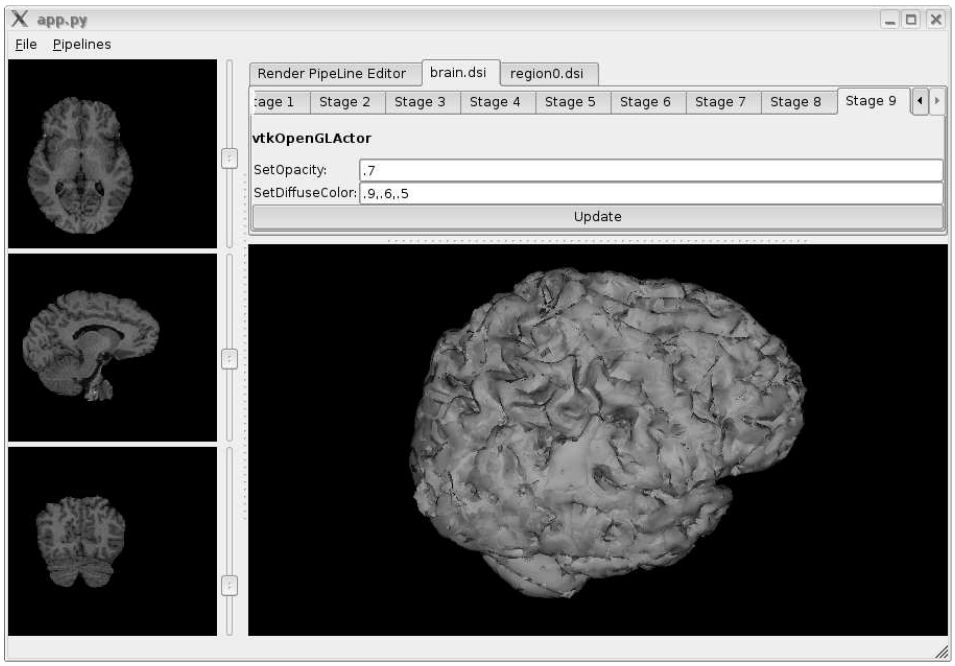

Figure 5: PyMed3D's user interface

PyMed3D allows for importing sets of images. An import wizard will pick the appropriate underlying `vtkImageReader` instance and configure it as completely as possible. The wizard subsequently produces a simple text configuration file that describes the volume in VTK compatible input formats. This approach allows more experienced users to create specialized input descriptions in cases where the import wizard is too generalized. Multiple image sets can be loaded simultaneously with independent pipelines; this allows for the creation of visualizations with multiple parts (e.g., fMRI with multiple "active regions").

The PyMed3D GUI allows for real-time modification of individual pipeline stages. Each dataset read by PyMed3D has its own pipeline editor, which is accessbile through a tabbed interface above the 3D viewport (Figure 5). For example, by adjusting the threshold or contouring values, various detail levels can be achieved. This interactive approach to adjusting parameters should prove more natural for individuals not all that comfortable with modifying lines of a program source file.

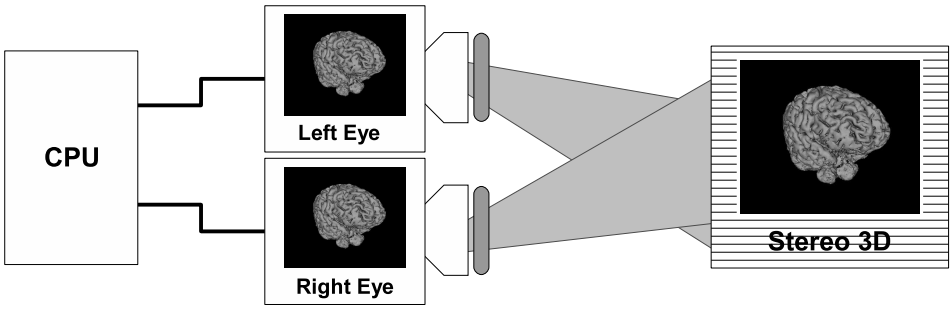

Figure 6: Stereo projection is accomplished by sending the two images from the dualhead graphics card to separate projectors with polarizing filters placed in front of them. The images are then projected on top of each other onto a silvered screen.

Once a satisfactory result has been achieved, pyMed3D users can export the models they are working on in a VTK compatible format. By using the VTK's built in export functions, the results can be used for other VTK visualizations, as well pyMed3D's own viewer application, which supports both regular and stereoscopic display.

Stereo Rendering

Stereo Viewing Setup

One of the main motivations for this visualization project was to make use of stereo presentation for a true 3D effect in a classroom setting. The main requirement for stereo viewing is getting separate images to the left and right eye to produce the 3D illusion. There are many different techniques for doing this[8]. Our setup, called SVEN, comprises two LCD projectors attached to the outputs of a stock dualhead graphics card. The computer desktop is extended across both screens so that images on the left-half of the desktop are shown on one projector and images on the right-half are projected on the other. Polarizing filters are placed in front of the projectors, and the images are superimposed on a silvered screen that preserves the polarization. Viewers wear inexpensive polarized 3D glasses. Figure 6 depicts the setup.

The main advantage of this stereo setup is that it does not require any expensive graphics hardware such as a quad-buffered stereo graphics card or shutter glasses. Instead, the application has to present a fullscreen window spanning the entire desktop with the left and right views placed side-by-side. The disadvantage of this approach is that off-the-shelf packages such as VTK are generally not designed for this type of passive stereo projection.

Embedding VTK in QT

Although the VTK does not include direct support for SVEN stereo, it is generally stereo-aware. The VTK provides stereo modes for certain types of active stereo displays as well as red-blue stereo mode. One nice design feature of the VTK is that the left and right views that are generated for these stereo modes are also available as separate rendering modes. In other words, its easy to tell a VTK window to simply produce what the left (or right) eye would see.

The simplest way to use the VTK with SVEN is to put two images side-by-side in a single window with the left image being the left-eye view and the right image being the right-eye view. Unfortunately, although VTK provides a way to place multiple renderers (images) into a single window, the selection of which stereo mode to use happens at the window level, not at the renderer level. So it is not directly possible to place left- and right-eye views into a single VTK window.

Our solution to this problem involves using the Python bindings for VTK and the QT graphical interface toolkit to create a QT widget that encapsulates a VTK window. VTK includes an example widget, `QVTKRenderWindowInteractor` that allows such embeddings. Using this widget as a base, it is relatively straightforward to place multiple VTK windows inside of a QT window, with each having a different stereo setting. One stumbling block in using the `QVTKRenderWindowInteractor` is that it did not work correctly out of the box with our versions of QT and Python. We had to tweak the code in several obvious ways such as converting certain integer parameters to make use of QTs enumerated values (e.g. `setBackgroundMode(2)` needed to be `setBackgroundMode(QWidget.NoBackground)`). Once these changes were made, we were able to embed VTK windows inside of PyQT applications.

For convenience, we designed a class called `QvtkWidget` that acts simultaneously as a generic QT Widget (a `QWidget`), a VTK window, and a VTK renderer. This is accomplished with a little Python "magic." The class itself is a subclass of `QWidget` (through `QVTKRenderWindowInteractor`). When an instance of `QvtkWidget` is created, it also creates an instance of `VTKRenderer` which is attached to the underlying `VTKRenderWindow` and also saved in an instance variable. All of these entities are tied together using Python's dynamic method dispatch mechanism. A QT method will succeed directly, since a `QvtkWidget` is a `QWidget`. When a non-QT method is called on the `QvtkWidget`, it first tries to find a suitable method in the associated `VTKRenderWindow` and then in the `VTKRenderer`. The programmer using a `QvtkWidget` sees an object that responds to all three kinds of methods and normally does not need to worry about where the various methods are actually implemented. This illustrates Python's utility as a general "glue language" for our application.

Synchronizing Stereo Views

Of course, being able to place two VTK windows side-by-side in an application is just the first step. We also need to set the windows up to show the appropriate stereo views and

stay in synch as the views are manipulated. For convenience, we want the pair of widgets to act like a single VTK window/renderer. To accomplish this, we created a new class `StereoQvtkWidgets`. An instance of this class encapsulates a pair of `QvtkWidgets`, representing the left- and right-eye views.

When a method is called on a `StereoQvtkWidgets` object, the object delegates the method to both its left and right views. So, for example, adding an object to both views in a stereo widget can be carried out in a single call:

```
# create a Stereo widget pair
stereoView = StereoQvtkWidgets()
...
# VTK calls are passed through to both left and right views
stereoView.AddActor(someVTKActor)
stereoView.ResetCamera()
...
```

The last piece of the display puzzle is ensuring that any interaction with the scene (changes in the camera) take place simultaneously in both views. This is done by having the two views share the same VTK Camera object that defines the current view. Any change to the camera will be reflected in both views. The only complication is making sure that both views actually get updated on screen. VTK provides an underlying observer/callback mechanism. We use this machinery to ensure that every time the camera changes, both views are notified of the event. The net result is that the programmer only has to place appropriate actors into the scene, and any mouse manipulations to zoom, rotate, or pan the view produce (nearly) simultaneous updates in both the left and right views.

These stereo viewing techniques are put to use in PyMed3D's viewer application (see Figure 7). This viewer is designed specifically to support fMRI visualization. Users specify a main model with an adjustable opacity. A second set of models (e.g., representing active regions) can be shown one at a time, superimposed on the main model. The application is interactive allowing the user to zoom, rotate, and pan the view using the mouse. The viewer can also be used for less specific, general stereoscopic visualization of VTK compatbile PolyData files.

Stereo Viewing Issues

Using the techniques described in this section, we were able to construct SVEN-style stereo views of VTK models with only a little bit (less than 100 lines) of Python glue code. The resulting interactive visualizations of simple models are remarkably smooth on relatively modest hardware (1.6 ghz P4 with stock graphics card). However, we do still have some remaining issues. When manipulating complex models (hundreds of thousands of polygons), the updating is not as immediate, and the stereo views are out of synch during interaction. This can be a somewhat jarring visual experience. We have also noticed a severe memory leak during interactions on some platforms. This can lead to a program lock-up if a model is manipulated over an extended time frame. These are issues that we still need to address.

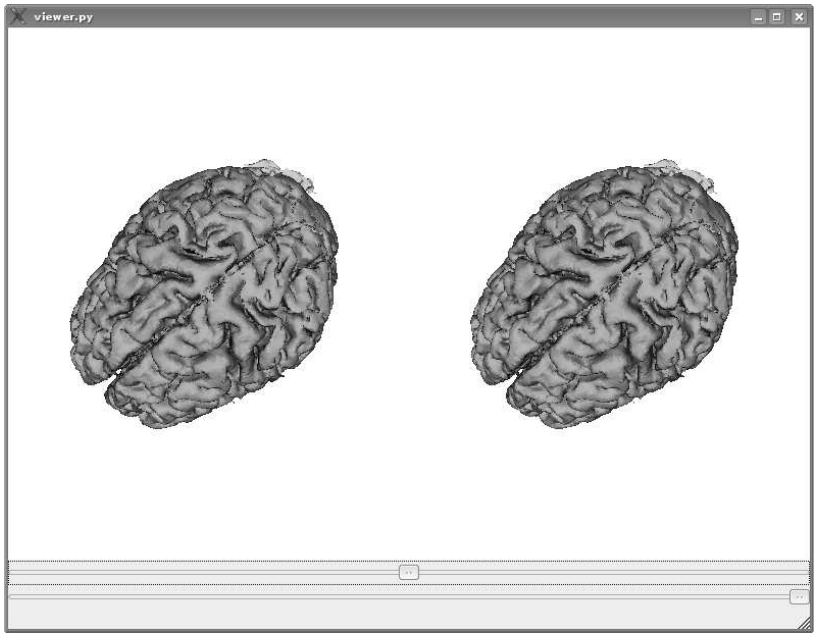

Figure 7: PyMed3D's viewer application showing left- and right-eye views. Slider controls at the bottom are used to adjust the opacity of the main model and to traverse through various extra models (e.g. fMRI regions)

Conclusions and Future Work

Building on VTK and the QT toolkit, we have been able to assemble a complete demonstration application, PyMed3D, that allows for stereographic display of fMRI data. While this is a significant milestone, it is really just a first step toward our longer-term goal of bringing 3D visualizations into the science classroom. We have a number of important directions for future work.

First, we would like to address some of the performance issues mentioned in the previous section. If it is not possible to interact smoothly with complex datasets, a viable alternative might be to generate an animation, such as a rotating view, that is then simply played back in the classroom. We have already begun investigations into appropriate formats for such stereo animations.

Another important task is packaging our work so that others can take advantage of it. That involves work on a number of fronts including: packaging the application so that it can be easily installed by other users, providing suitable documentation so that non-experts can build sucessful visualizations, and distributing our work via the web. We hope that by making our source code available to others, we can get feedback to continue its improvement.

Finally, we hope to get our visualizations classroom tested in neurobiology courses. That feedback will be invaluable in guiding the future development of PyMed3D. Building the system has been an interesting project, but it will be a disappointment if that effort does not ultimately result in a useful educational tool.

Acknowledegments

This work was partially supported by a Maytag Innovation Award for student/faculty research and by the Wartburg College Undergraduate Research Fellowship program. We further would like to thank especially Mark Dow and Greogory Scott from the University of Oregon, who provided us with fMRI data, and John M. Melville, our local neuroscientist at Wartburg College who inspired much of this work and answered a plethora of questions.

References

[1] http://www.bic.mni.mcgill.ca/nifti/ NIFTI (neuroimaging informatics technology initiative).

[2] http://www.riverbankcomputing.co.uk/pyqt/ PyQt: Overview.

[3] http://www.python.org Python programming language – official website.

[4] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit An ObjectOriented Approach To 3D Graphics*. Kitware, Inc. publishers, third edition, 2004.

[5] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit User's Guide*. Kitware, Inc. publishers, 4.4 edition, 2004.

[6] S.M. Smith, M. Jenkinson, M.W. Woolrich, C.F. Beckmann, T.E.J. Behrens, H. JohansenBerg, P.R. Bannister, M. De Luca, I. Drobnjak, D.E. Flitney, R. Niazy, J. Saunders, J. Vickers, Y. Zhang, N. De Stefano, J.M. Brady, , and P.M. Matthews. Advances in functional and structural MR image analysis and implementation as FSL. *NeuroImage*, 33(S1)(208219), 2004.

[7] VTK 4.2.1 documentation. http://www.vtk.org/doc/release/4.2/html/.

[8] John M. Zelle and Charles Figura. Simple lowcost stereographics: VR for everyone. *SIGCSE Bulletin*, 36(1)(348357), March 2004.

APPENDIX

A VTK Surface Extraction

The following is a python script creating a full VTK Surface Extraction pipleine. The `vtkActor` instance at the end of the pipeline, can be added to a `vtkRenderWindow` instance by using the `AddActor()` method.

```
reader =  vtk.vtkPNGReader()
reader.SetFilePrefix("brain")
reader.SetFilePattern("%s%03i.png")
reader.SetDataExtent(0,199,0,263,51,212)
reader.SetDataSpacing(1, 1, 1)
reader.Update()
pipe = reader

shrink = vtk.vtkImageShrink3D()
shrink.SetInput(pipe.GetOutput())
shrink.SetShrinkFactors((2,2,1))
pipe = shrink

thresh=vtk.vtkImageThreshold()
thresh.SetInput(pipe.GetOutput())
thresh.ThresholdByUpper(float(sys.argv[1]))
thresh.SetInValue(255)
thresh.SetOutValue(0)
thresh.ReleaseDataFlagOff()
pipe = thresh

cf = vtk.vtkContourFilter()
cf.SetInput(pipe.GetOutput())
cf.SetValue(0,255);
pipe = cf

con = vtk.vtkPolyDataConnectivityFilter()
con.SetInput(pipe.GetOutput());
con.SetExtractionModeToLargestRegion();
pipe = con

dec = vtk.vtkDecimatePro()
dec.SetInput(pipe.GetOutput())
dec.SetFeatureAngle(1)
dec.PreserveTopologyOn()
pipe = dec

norm = vtk.vtkPolyDataNormals()
norm.SetInput(pipe.GetOutput())
norm.SetFeatureAngle(10)
pipe = norm

mapper = vtk.vtkPolyDataMapper()
mapper.SetInput(reader.GetOutput())
mapper.ScalarVisibilityOff()

actor = vtk.vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetDiffuseColor(1,.6,.6)
actor.GetProperty().SetOpacity(.1)

#actor can be added to vtkRenderWindow
```