# A simple approach to Mayavi with Python and the Show.py Script

**Antonio Xavier Garcia**

**antonioxgarcia@gmail.com**

**March 10, 2007**

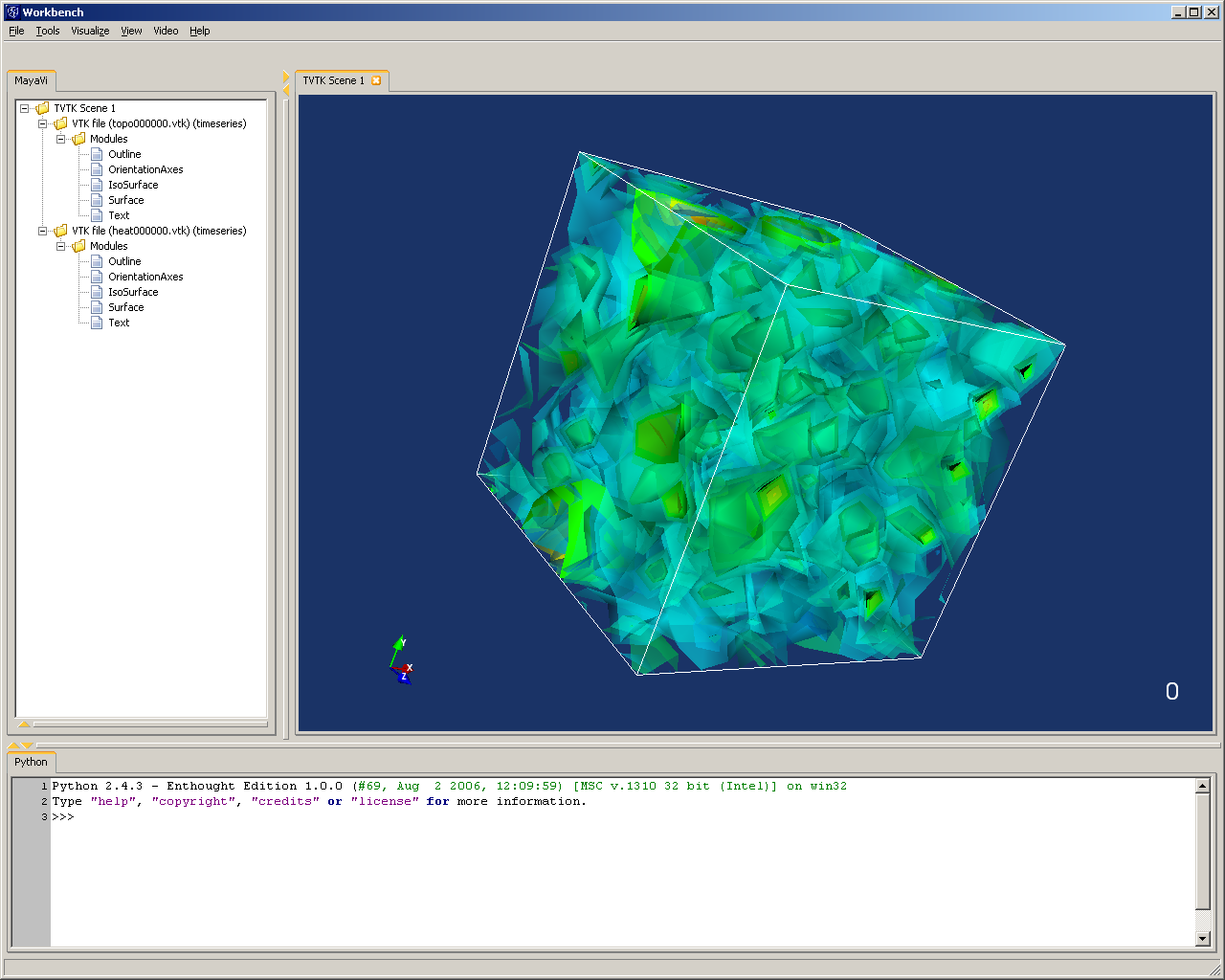
### Introduction

### What is Mayavi?

Mayavi2 is an Envisage plug-in for 2D/3D scientific data visualization and is the predecessor to Mayavi. Mayavi2 uses the Enthought tool-suite in the form of Traits, TVTK and Envisage.

Some of the features include:

* Extensibility
* Ease of visualizing scalar and vector data in 2D and 3D.
* wxPython Based
* Ability to save I a variety of formats.



### Mayavi Scripting

### Setup

Mayavi2 is the successor of Mayavi and is an application for the rendering of scientific data for 2D/3D visualization. It relies on VTK and a python interface as underlying frameworks.

The Visualization Toolkit (VTK) is a popular, open source, high-level visualization library. VTK is implemented in C++ and provides convenient wrappers for Python, Tcl and Java.

In the following sections we will be covering a number of code examples using Python. For these purposes we will use the IDLE (Python GUI) that comes with the Enthought Edition Python 2.4.x install. This will allow us to script out our code and see any exceptions as they occur with each line of code written. If you would like to keep your code simply write out your code to a python file and start it up as you would any python script.

### Writing your first script

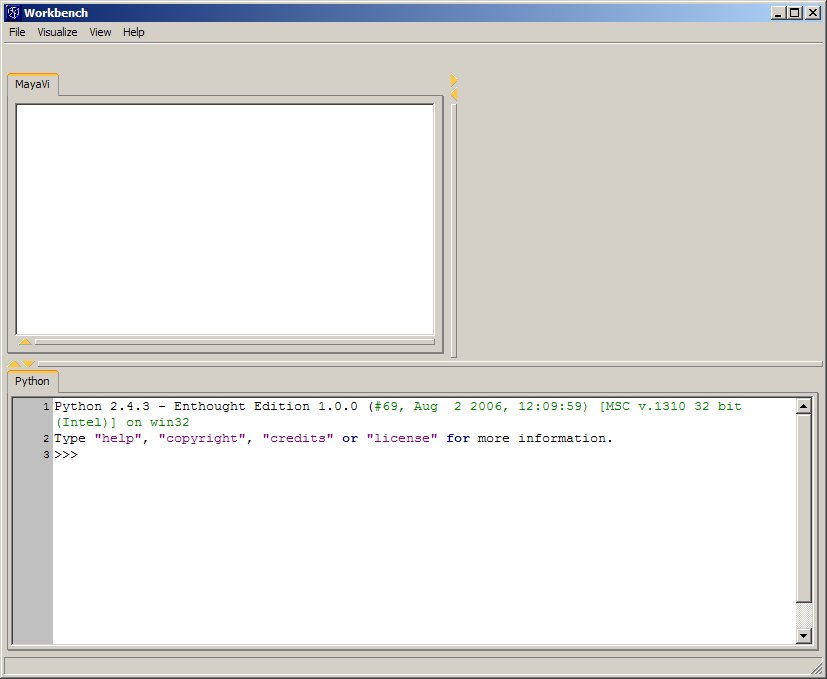
We will begin by writing a simple bit of code that will open up the Mayavi Workbench environment. To do this we only need to create a new instance of a Mayavi class and call the main function to open it. That snippet might be implemented as follows.

**from enthought.mayavi.app import Mayavi**

**m = Mayavi()**

**m.main()**

You will notice that to import the Mayavi libraries we do so from the Enthought namespace. This will ensure that we will not experience a conflict so that the Mayavi class that we import is indeed from Enthought. After running the three lines of code you should be presented with an empty Mayavi workbench as shown below.



#### Expanding on the Mayavi class

While the Mayavi Workbench is a quite power GUI interface there is always the need to expand on it so that custom functionality can be added. To do this we can create a new class that inherits Mayavi and over load its behavior. This is in fact the manner in which the Mayavi2.py functions, proving the power of this methodology. If you are not familiar with this script, it is a sample script created by the developers at Enthought but do not worry you need not know anything about it to continue.

To begin we will make a simple class that inherits Mayavi and opens a hard coded VTK file.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**v = MyMayavi()**

**v.main()**

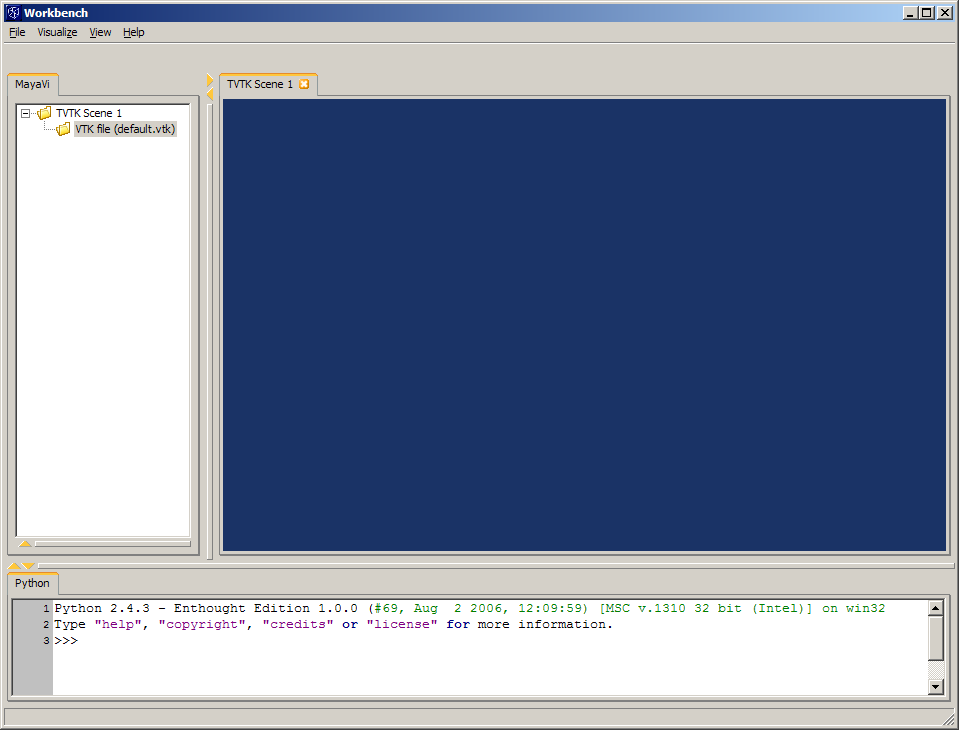
Once again we import Mayavi from the enthought.mayavi.app namespace and then we proceed to declare a new class call MyMayavi that inherits Mayavi. The function that we declare called run is a function that is always executed once the application GUI has started up. It is suggested that all other Mayavi imports are performed here, as such in our example where we import the VTKFileReader.

The VTKFileReader is as its name suggests a file reader that reads in VTK files. It should be noted that this class does not support the newer XML file format of VTK but only the older formats of ASCII and binary. This reader also supports all data type sets and time series. For more information on this class object please refer to the Enthought documentation available online.

After our imports in the run function we will call into the script class of Mayavi by calling self.script. The script class provides a scriptable view of the Mayavi Engine, which we are using to call the new\_scene() function. This function creates a new scene on the Mayavi Workbench so that the VTK file rendering may be displayed.

Now we have come to the part where we actually create a new instance of a VTKFileReader and we will set it to our variable named d. Using this new instance we can initialize it by passing in a VTK file on our system that we want to be read in, in this case we choose the file default.vtk located on our C:\ drive. At this point we have an instance of the VTKFileReader that is properly set up and we need only hand it off to the Mayavi for rendering. This is done by calling the function mayavi.add\_sources and passing in our VTKFileReader.

We now have a class that will start a Mayavi Workbench and open up a VTK file from the hard coded location that we specified. Run the code by creating a new instance of our class and call the main function, as was done in our previous example. This should display a Mayavi Workbench similar to the one in the image below.



You will notice that while the VTK file is open nothing is displayed on the scene. This is expected since we did not instruct Mayavi to do anything further and will cover that next.

### Mayavi Foundations

#### Experimenting with Modules

In our last example we showed that we were able to inherit the Mayavi class and get it to open up a VTK file of our choosing. In this section we will go further and delve into how to get the Mayavi Workbench to render our VTK file using Modules. These modules allow our data to be displayed and manipulated to suit our needs.

In the following examples we will give a brief over view of what are known as basic modules and then proceed further into to what are known as main modules.

#### Basic Modules

The basic modules are so named because they are general and independent of all data sets. This means that they will work with all data sets that we load unlike main modules which can only been used with specific data set types. These modules include the Outline module, the Axes module, the OrientationAxes module and the Text module.

The Outline is probably the most basic module provided by the Enthought libraries. This module simply displays an outline, as the name suggests, for the given data. Let’s see an example of how we can add this module to our MyMayavi class.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#add modules**

**#Outline Module**

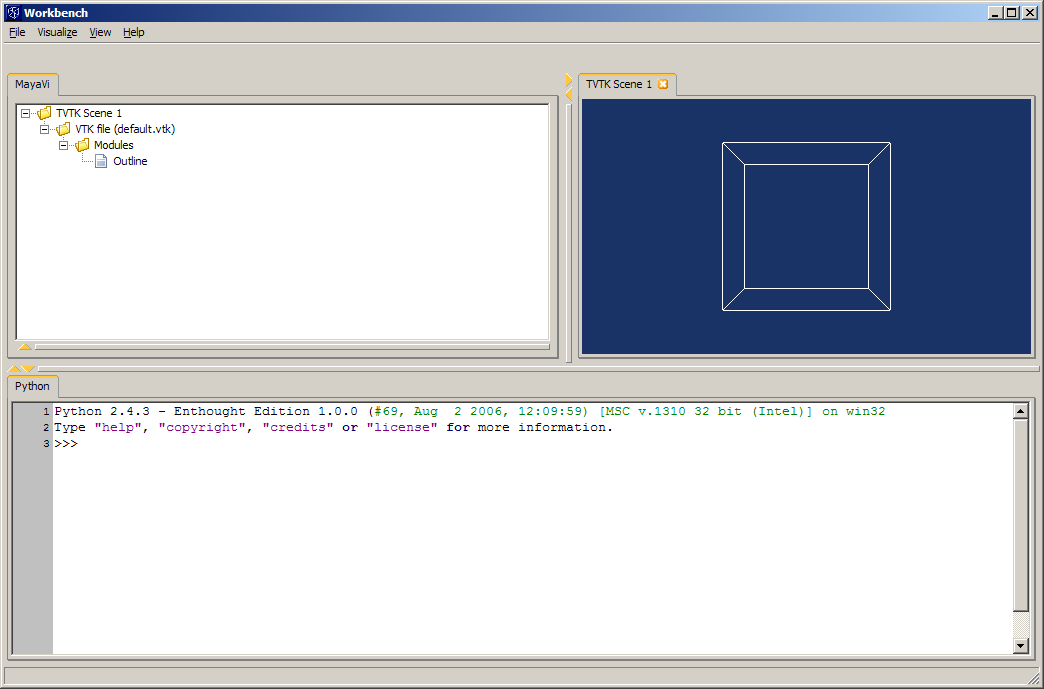
**o = Outline()**

**mayavi.add\_module(o)**

**v = MyMayavi()**

**v.main()**

You will notice that adding a module is a simple matter of creating an instance of the Outline class and then passing it into the mayavi.add\_module function, which will then display the module on the current scene and place it under the current data source in the Mayavi tree.



Given the simplicity of the Outline module we need not do any more than just create an instance of it but for other module that are slightly more complex we can set the properties of those module.

The Axes module will give us an example of one such module, which has properties that can be set. This module again is pretty simple but it gives the ability to label the X-axes, Y-axes and Z-axes of the rendering of our data set.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.axes import Axes**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#Axes Module**

**a = Axes()**

**mayavi.add\_module(a)**

**a.axes.x\_label = "X"**

**a.axes.y\_label = "Y"**

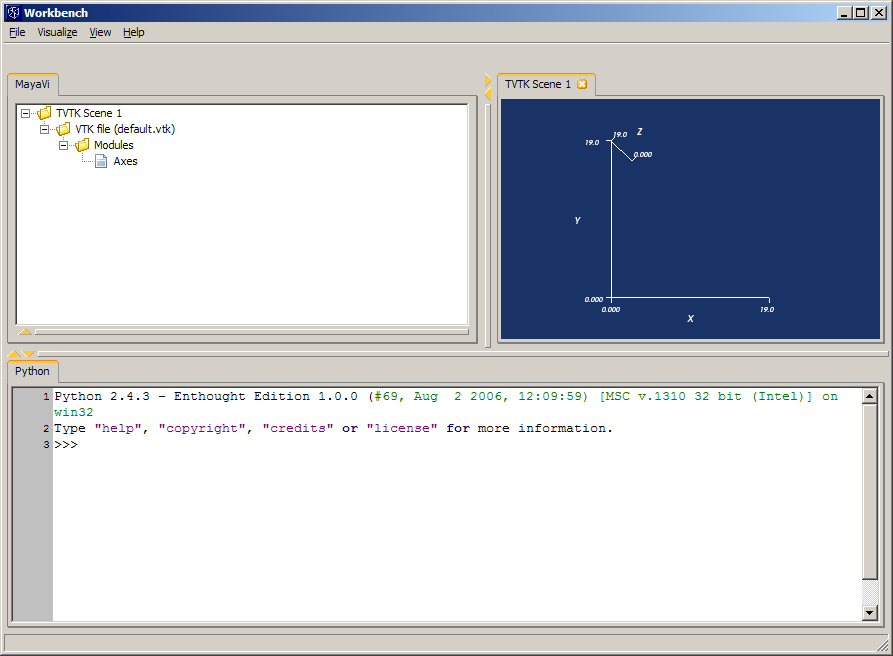
**a.axes.z\_label = "Z"**

**v = MyMayavi()**

**v.main()**

This example shows how to add an Axes module to our MyMayavi class. We do this similarly to how we added the Outline module but we modify it so that the x\_label will read “X”, the y\_label will read “Y” and the z\_label will read “Z.”

Below is our MyMayavi class running. We can see that labels added are now rendered for this module.



The next example is a module called the orientation axes. This module adds a trihedron to the rendering. By default this will be added the bottom right corner of the current scene thought we can modify this by accessing the modules properties.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.orientation\_axes import OrientationAxes**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#OrientationAxes Module**

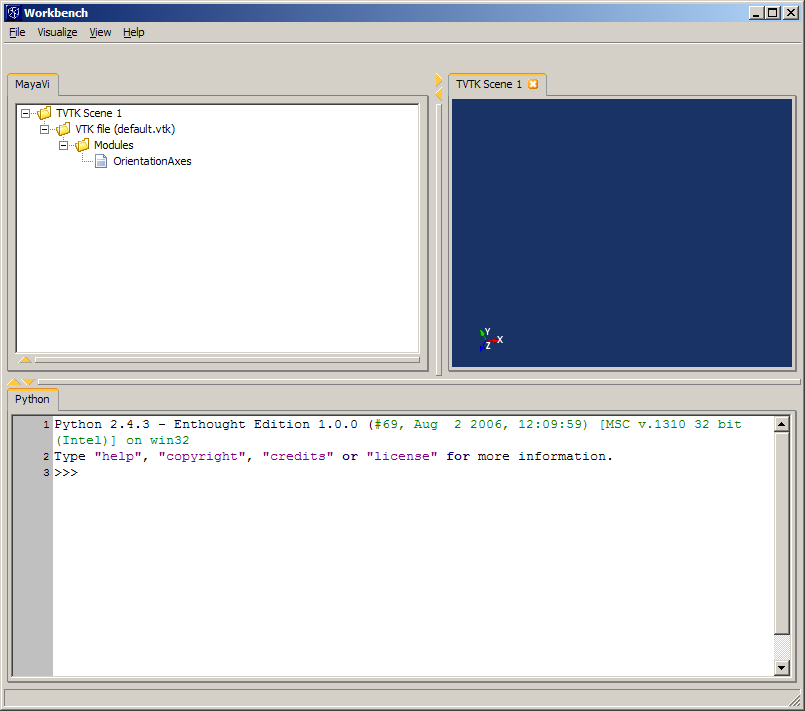
**oa = OrientationAxes()**

**mayavi.add\_module(oa)**

**v = MyMayavi()**

**v.main()**

You will notice that the code for this is minimal and we get a nice trihedron in the scene rendering.



The last of the basic modules is arguably one of the most useful of this type of modules. The Text module allows one to place text anywhere on the rendered scene in any font and/or format of your choosing.

The X and Y positions can be set using the x\_postion and y\_position parameter respectively and the window coordinates go from 0 to 1. Below is our example of how to add a Text module to our MyMayavi class.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.text import Text**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#Text Module**

**t = Text()**

**t.text = "Hello World!"**

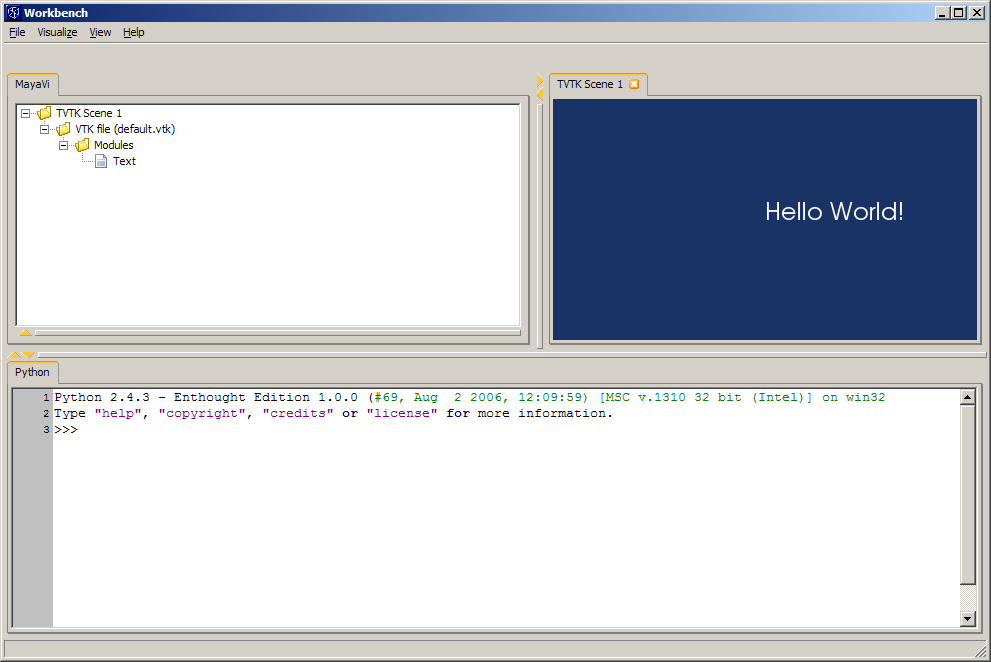
**mayavi.add\_module(t)**

**t.actor.text\_property.color = (1.0, 1.0, 1.0)**

**v = MyMayavi()**

**v.main()**

This example shows how to add the programmer’s quintessential “Hello World!” text to a rendering. You will see that we in our example we set the text color to the value (1.0, 1.0, 1.0) which is representative or the color white. Color is represented by a 3-tuple value of RGB values with each value ranging from 0 to 1.0.



#### Main Modules

We will now proceed to show a few examples of the modules that are known as main modules. Unlike the basic modules these modules do not all work with any type of data set. For example, some of these modules work only for StructuredPoints or StructuredGrids. It is best that you refer to the VTK file format documentation to understand the data that you are dealing with.

Once you have selected the appropriate module that you would like to use, adding the model is very similar to adding a basic module. Lets start by taking a look at the IsoSurface module.

The IsoSurface module displays the 3 dimensional representations of points with equal values in a 3D data distribution. This module can be added to our MyMayavi class by simply creating a new instance of the IsoSurface class and passing it into the add\_module function of mayavi.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**from enthought.mayavi.modules.iso\_surface import IsoSurface**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#Outline Module**

**o = Outline()**

**mayavi.add\_module(o)**

**#IsoSurface**

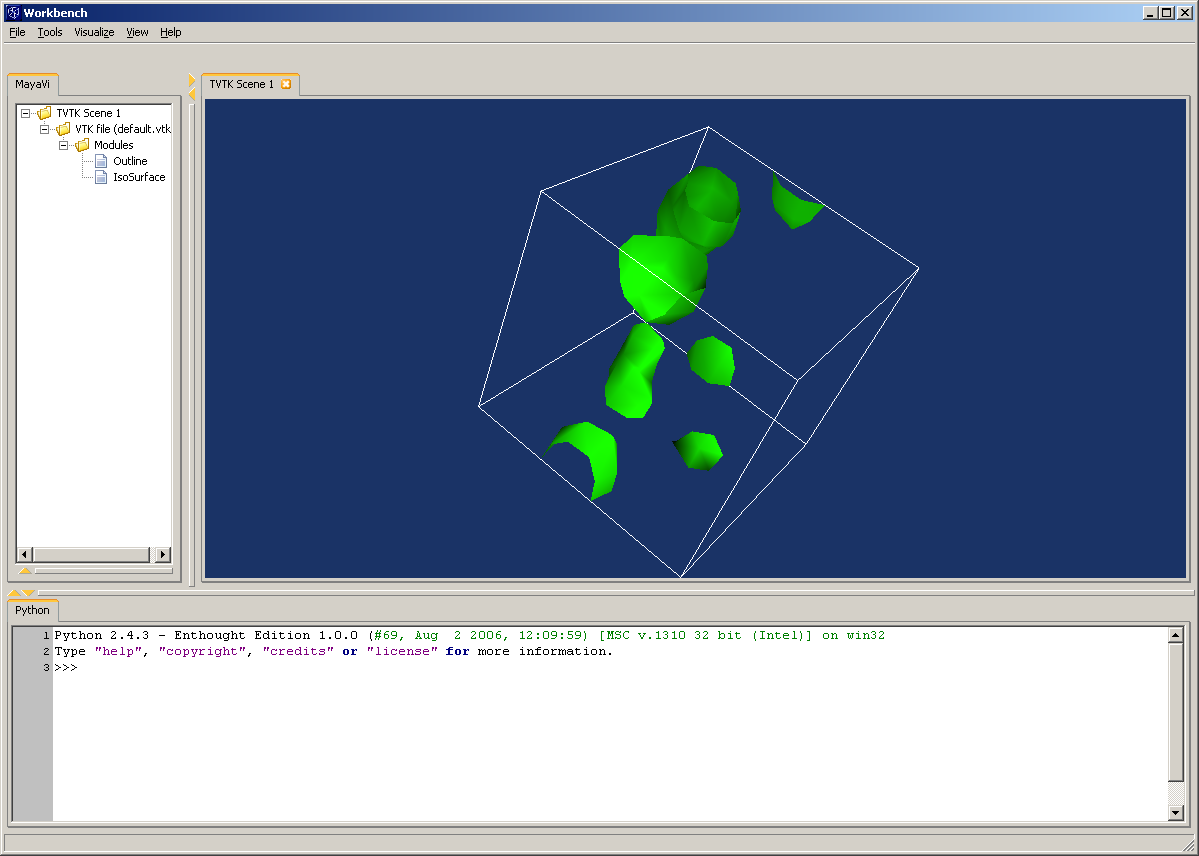
**i = IsoSurface()**

**mayavi.add\_module(i)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**v = MyMayavi()**

**v.main()**



This seems very similar to the usage of basic modules and indeed it is with the difference that main modules tend to be more configurable so that they can be tailored the fit the users specific needs. We will see an example of this in the following example.

The surface module is very similar in what it renders to the IsoSurface module in that it renders several isosurfaces for a given number of values in a specified range. The surface module does this automatically and can be modified via user configuration. In our example we will create an instance of a Surface module and then proceed to enable contours and make the rendering opacity 50 percent.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**from enthought.mayavi.modules.surface import Surface**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#Outline Module**

**o = Outline()**

**mayavi.add\_module(o)**

**#Surface**

**s = Surface()**

**s.enable\_contours = True**

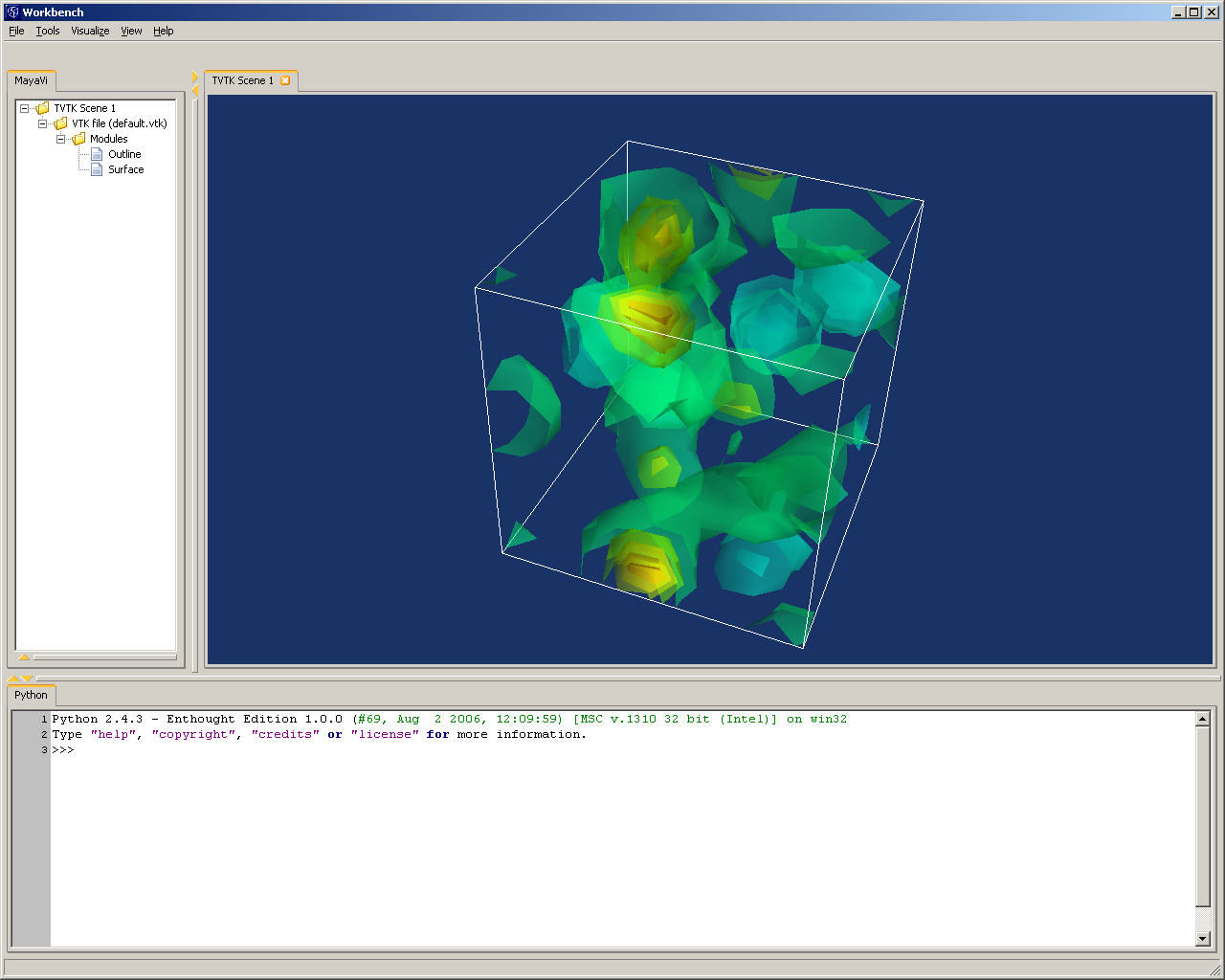
**s.actor.property.opacity = 0.5 #50% opacity**

**mayavi.add\_module(s)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**v = MyMayavi()**

**v.main()**



You will note that when run, the rendering for the Surface module is very similar to isosurface as fore mentioned.

In our next example we will show the usage of a ScalarCutPlane module. This module adds the ability to slice a particular plane of our data set.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**from enthought.mayavi.modules.scalar\_cut\_plane import ScalarCutPlane**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#add modules**

**#Outline Module**

**o = Outline()**

**mayavi.add\_module(o)**

**scp = ScalarCutPlane()**

**scp.implicit\_plane.normal = (1,0,0) #set normal to O at the X axis**

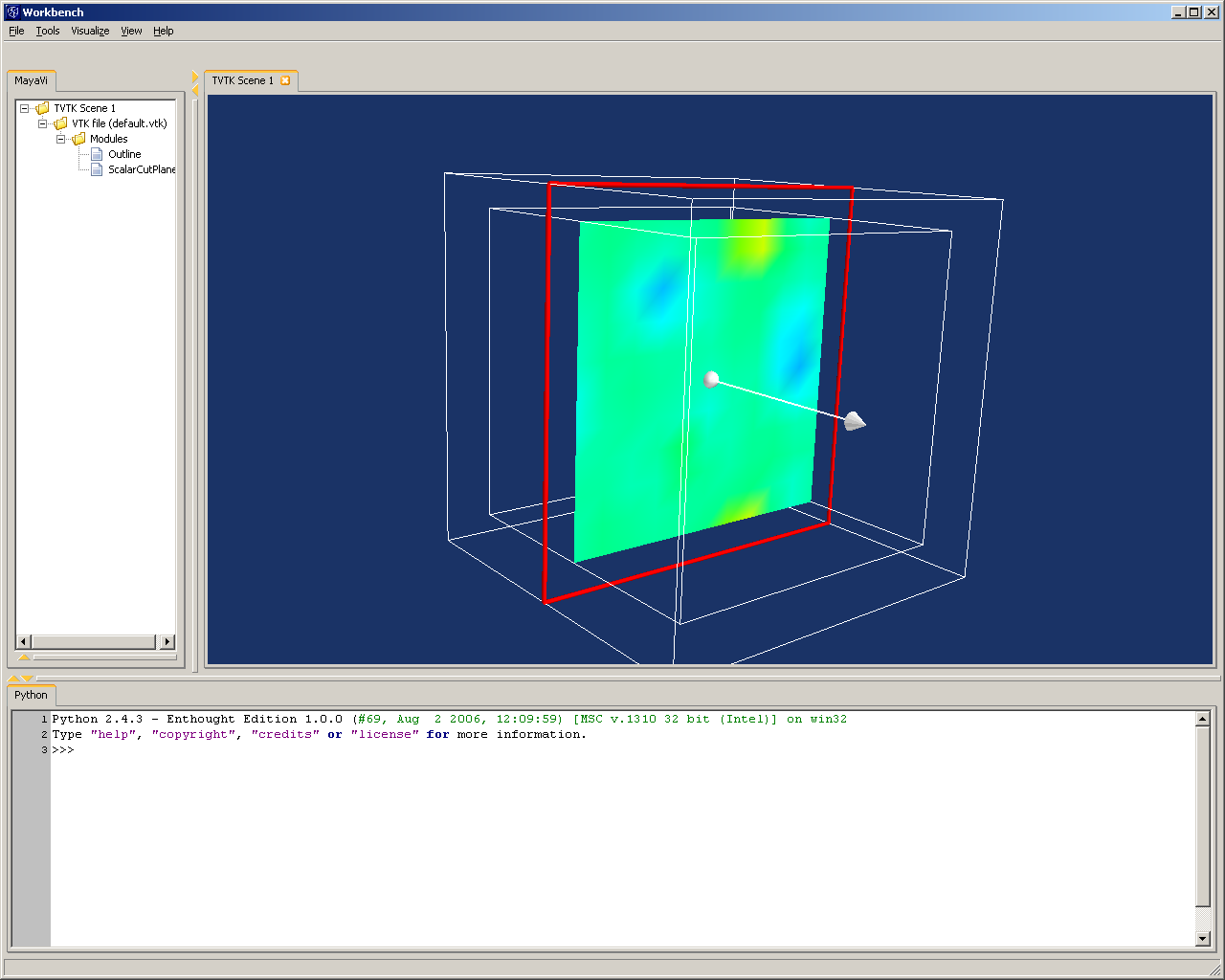
**mayavi.add\_module(scp)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**v = MyMayavi()**

**v.main()**

As you can see that is a pretty simple module to configure. In this example we only set the normal property so that the plane would slice thought the X axes at the 0.



When run you will see this module only renders a slice of our scalar data.

Next we will examine the Vectors module. This module is very similar to the previous modules with its properties and functionality however it differs in the type of data that it is rendering. This module handles the rendering of vectors as appose the scalar data that we have explored up until now.

The properties of the Vector module allow you to configure a number of values pertaining to the arrow shape. This being so there are a few things that should be known before proceeding. First, it is suggested that data be decimated else you may see a nothing but many arrows and hence a loss of data. Secondly, you should choose the shape of your vector; by default this will be the ArrowSource (i.e. your vector data will be represented by arrows). Lastly, it is optional, since you can use the defaults, but you can set properties of the shape of the vector you choose so to customize the representation of your data.

To add a vector module to our MyMayavi class we will first need to make sure that the file that the VTKFileReader is using has a vector data set. Once you have done that adding the module is simply a matter of instantiating an instance of the Vectors object and passing it into the add\_module function as previously performed.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**from enthought.mayavi.modules.vectors import Vector**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default\_vector.vtk")**

**mayavi.add\_source(d)**

**#Outline Module**

**o = Outline()**

**mayavi.add\_module(o)**

**#vector module**

**v = Vectors()**

**mayavi.add\_module(v)**

**v.glyph.mask\_input\_points = True #decimate data**

**v.glyph.mask\_points.on\_ratio = 100 # ratio of 100**

**v.glyph.mask\_points.random\_mode = \**

**True #randomly decimate**

**#choose glyph source**

**#0 - GlyphSource2D**

**#1 - ArrowSource**

**#2 - ConeSource**

**#3 - CylinderSource**

**#4 - SphereSource**

**#5 - CubeSource**

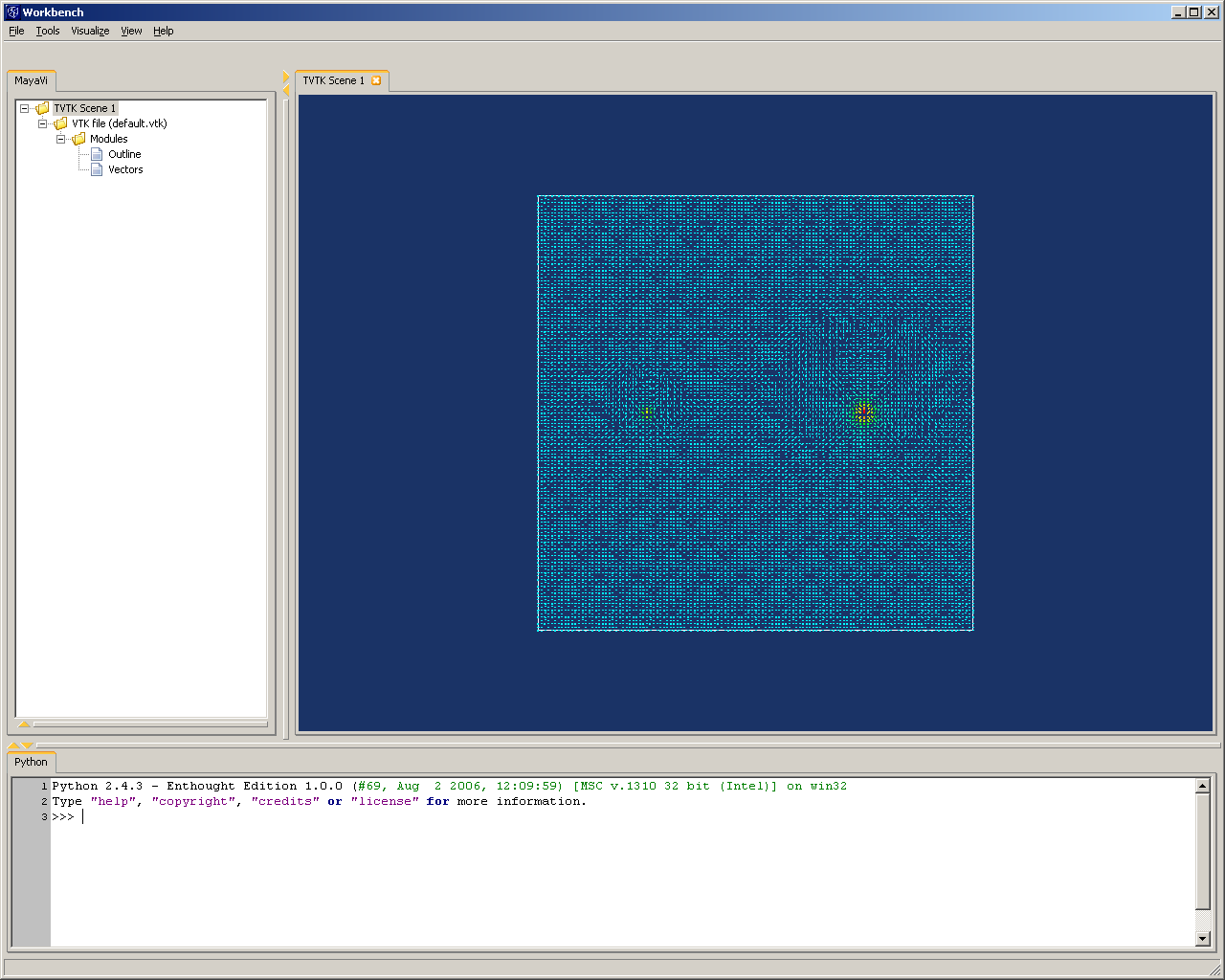
**v.glyph.glyph\_source = \**

**v.glyph.glyph\_list[1] #arrowsource**

**if \_\_name\_\_ == '\_\_main\_\_':**

**v = MyMayavi()**

**v.main()**



You will notice that the vector module works closely with the Glyph module and that is because the Vector and Glyph module as well as the VectorCutPlane module use many of the same properties with only different default properties.

#### Experimenting with Filters

Up until now we have looked into Mayavi modules that can be added for a particular VTK file so that they represent scalar or vector data. We will now briefly explore how to add Mayavi filters. This will be a shorten dive in to what can be a particularly advance and complex topic depending on the data you are attempting to render. For further exploration into Mayavi filters as well as a number of very good example I would suggest checking out the Scipy.org web site (http://www.scipy.org/Cookbook/MayaVi/ScriptingMayavi2#head-c2a3ffab1265a7426fde4b3840751396ae8f6d50)

#### The Threshold Filter

The Threshold filter will allow you to have scalar data within a given range rendered. This can be quite useful if you care to only view a subset of your data set.

When adding a filter you will notice that you use the add\_filter function of the mayavi class and then add your module to the filter itself using the threshold’s add\_module function. This is a bit different than what we has previously seen where we needed to add our mayavi modules using the mayavi.add\_module function but it should make sense since what we are really asking for is to have the module filtered by the filter object. Below is a simple example of the usage of the threshold filter.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**from enthought.mayavi.modules.vectors import Vector**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default\_vector.vtk")**

**mayavi.add\_source(d)**

**#Outline Module**

**o = Outline()**

**mayavi.add\_module(o)**

**#filter**

**i = IsoSurface()**

**th = Threshold()**

**mayavi.add\_filter(th)**

**th.lower\_threshold = 0.2**

**th.upper\_threshold = 0.8**

**th.add\_module(i)**

**i.contour.contours = [0.5]**

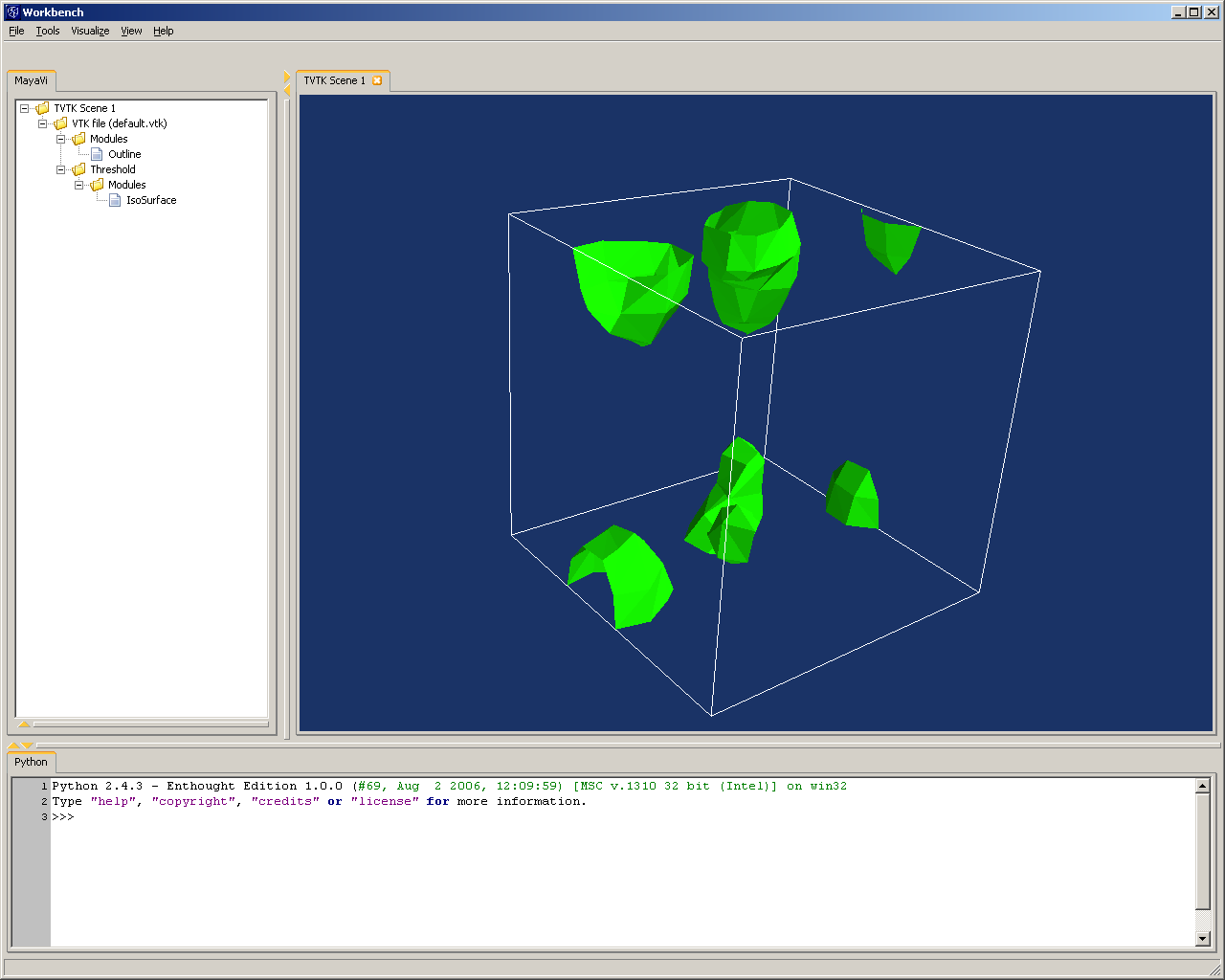
**i.compute\_normals = True**

**if \_\_name\_\_ == '\_\_main\_\_':**

**v = MyMayavi()**

**v.main()**

You can see the following image that the IsoSurface module appears under the threshold in the Mayavi tree and not in the modules level as before.



### Advanced Scripting

#### Mayavi objects and the Mayavi Workbench can be very powerful however on occasion there is the need to access some of the underlying components to expose further functionality. This section will cover a few of the more advanced topic that can prove quite useful when scripting to Mayavi.

#### Saving

#### Mayavi provides a number of ways to save data in a variety of file formats. These include saving the rendered data in a particular VTK scene to a file, saving a VTK scene to the clipboard or saving the whole visualization.

#### We will look first at saving a VTK scene to a file by accessing the Scene object within the mayavi engine. Below is a quick overview of how to save a single scene to the many file formats provided to you by Mayavi.

**from enthought.mayavi.app import Mayavi**

**class MyMayavi(Mayavi):**

**def run(self):**

**from enthought.mayavi.sources.vtk\_file\_reader import VTKFileReader**

**from enthought.mayavi.modules.outline import Outline**

**self.script.new\_scene()**

**d = VTKFileReader()**

**d.initialize("C:\\default.vtk")**

**mayavi.add\_source(d)**

**#Outline Module**

**o = Outline()**

**mayavi.add\_module(o)**

**e = mayavi.engine**

**s = e.current\_scene.scene**

**#save a bitmap file**

**s.save\_bmp("file.bmp")**

**#save image as png format**

**s.save\_png("file.png")**

**#save image as GL2PS(opengl to PostScript)**

**s.save\_gl2ps("file.gl2ps")# file\_name, exp=None**

**#save image as OpenInventor format**

**s.save\_iv("file.vi")**

**#save image as jpeg**

**s.save\_jpg("file.jpg")**

**#save image as Geomview format**

**s.save\_oogl("file.oogl")**

**#save image as postscript**

**s.save\_ps("file.ps")**

**#save image as RenderMan format**

**s.save\_rib("file.rib")# file\_name, bg=0, resolution=None, resfactor=1.0**

**#save image as TIFF format**

**s.save\_tiff("file.tiff")**

**#save image as Virtual Reality Markup Language**

**s.save\_vrml("file.vrml")**

**#save image as Wavefront format**

**s.save\_wavefront("file.obj")**

**if \_\_name\_\_ == '\_\_main\_\_':**

**v = MyMayavi()**

**v.main()**

As you can see accessing the scene is simply a matter of obtaining the property from mayavi.engine.current\_scene.scene. Once you have this object you can simply call the save function that best fits your needs.

Next we will explore how to save the current image not to a file but to the clipboard. To do this we use the same code as above except we use the function save\_to\_clipboard. This can be done like so.

**e = mayavi.engine**

**s = e.current\_scene.scene**

**s.save\_to\_clipboard()**

Note that this is very much similar to saving an image to a file but this will simply put the image on your clipboard. There is a limitation to the type of image you can capture in this manner however since the image that is saved to the clipboard can only be a bitmap.

There is one last saving technique that we will explore which is how to save a visualization file. A visualization file holds the configuration settings that have been set such as which data sources to load, the modules to be rendered for each data source and all the property values for each module. Visualization files saved are expected to use the extension mv2 for Mayavi2. To save this file programmatically we can use the following code snippet from anywhere within our MyMayavi class.

**mayavi.save\_visualization(“myVisualization.mv2”)**

Thought this section is dedicated to saving data we can use a similar function to load up a visualization file programmatically by simply doing the following.

**mayavi.load\_visualization(“myVisualization.mv2”)**

For more information you can find further documentation provided by SciPy.org at http://new.scipy.org/Wiki/Cookbook/MayaVi/Tips.

#### Accessing wxPython in Mayavi

The Mayavi Workbench is based on the wxPython, which is a cross platform wrapper for the systems GUI API. By obtaining access to this API we can customize the underlying objects to perform as we see fit. In this section we will briefly cover how to expose wx objects as well as their usage and how to modify them.

We will start by showing how we can programmatically obtain the wxFrame object from mayavi. To do this we can simply use the following line of code.

**win = mayavi.engine.appliaciton.gui.GetTopWindow()**

This one line will have given us a handle to the main window of the Mayavi Workbench, which opens up a number of possible options that we can take to modify it. For instance if we want to center the window on the screen we can simply call the CenterOnScreen function

**win.CenterOnScreen()**

This is a simple function indeed but quite powerful as well. A few of the other function that may be useful are listed below.

**#Maximize the window**

**win.Maximize()**

**#Get the title of the window**

**win.GetTitle()**

**#Set a new title for the window**

**win.SetTitle(u’Hello World!’)**

**#Get the size of the window**

**mySize = win.GetSize()**

Getting and setting properties of the Mayavi Workbench window can be valuable but there are occasions when we need to add, modify or remove object from the WorkBench itself for usability. Depending on what your particular goal is this may or may not be a trivial exercise and I will try not to delve too far into the intricacies of the wxPython toolkit. This last example will show how we can easily add a new menu item to the Workbench so that we can expose a new custom option to the user.

We will be creating new wx objects so the first thing that we will need to do is import the appropriate libraries. For this example we will only need wx.

**import wx**

First we must access the wxMenuBar object from the window. After that we will need to create a new wxMenu object to add to it and then insert it into the menubar.

**menubar = win.GetMenuBar()**

**menu = wx.Menu(“”)**

**#find the last spot for the menu**

**menuPos = menubar.GetMenuCount() – 1**

**menubar.Insert( menuPos, menu, “Test”)**

This has added a new menu to the menu bar but we still need to add a menu item as well. For this we will need to create a MenuItem object and append it to the menu and then bind an event to it so that it will do something when the user clicks on it.

**#Create Menu Item**

**mnuItem = wx.MenuItem( menu, wx.ID\_ANY, “Click Me”)**

**menu.Append( mnuItem )**

**#bind menu item to the OnClick function that we created**

**window.Bind(wx.EVT\_MENU, OnClick, mnuItem)**

You will notice that the binding takes a type of wx event, a function callback and the item being bound to the event, which in our case is the MenuItem object. Now when the user clicks on the menu item the OnClick function with fire.

This should give you a very simplified view of how you can access the wxPython toolkit that the mayavi WorkBench is built on. There is of course far more material to cover on this topic but that is far beyond the scope of the document. For further documentation on wxPython you should have a look at the wxPython documentation that is provided online (http://www.wxpython.org).

#### Putting it all together: The Show.py Script

In this section we will cover the main functionality provided by Show.py and examine both how it is to be used by the user and how it was implemented in code. To do this each section will be divided up into two subsections: Usage (which will explain how the user interacts with a particular feature) and Programmatic Implementation (which as the name suggests explains how the feature was implemented).

#### Show.py: A Mayavi Plug-in

The Show.py script is an extension of Mayavi (distributed by Enthought, Inc) that pre-configures the Mayavi Workbench with custom features. These include the ability to load VTK file(s), poll loaded the file(s) for updates or allow the user to specify a group of files to be loaded and iterating over them at a 1 second interval.

The script also provides advanced functionalities such as well such as video generation and automatic image capture when the rendering changes. In all this provided a very powerful tool that requires minimum configuration and setup to get started.

#### Overview

#### Usage

Show.py provides a simple command line interface to get up and running. This interface is designed to hide much of Mayavi and exposes only a subset of preconfigured options to the user. The following is the output provided by show.py when help is requested via the show.py –help command.

usage: show.py -f [FILE]... [OPTION]...

show.py -u [FILE]... [OPTION]...

show.py -l [PATTERN]... [OPTION]...

Show.py is an extension of Mayavi (distributed by Enthought, Inc) which loads

VTK files, can poll for updates on loaded file(s) and enables the iterating of

files in a particular directory. Rendered scenes can be captured as images,

saved to a directory and then rendered as a mpeg video files.

options:

--version show program's version number and exit

-h, --help show this help message and exit

-f SINGLEFILENAME, --file=SINGLEFILENAME

Load VTK file(s) onto the Mayavi Workbench. To load

multiple files use ';' as a delimiter (e.g. show.py -f

file1.vtk;file2.vtk)

-u UPDATEFILENAME, --update=UPDATEFILENAME

Load VTK file(s) and listen for updates. To load

multiple files use ';' as a delimiter (e.g. show.py -u

file1.vtk;file2.vtk)

-l PATTERN, --loop=PATTERN

Iterates through VTK files that fit the defined

pattern. To load multiple lists of files use ';' as a

delimiter (e.g. show.py -l ./fileA\*.vtk;./fileB\*.vtk)

-i ISOSURFACE, --isosurface=ISOSURFACE

Turn on IsoSurface module

-s SURFACE, --surface=SURFACE

Turn on Surface module

-t TEXT, --text=TEXT Turn on Text modules and use text as title

-o IMAGEFILEDIR, --out=IMAGEFILEDIR

Turn on image capture of rendering and outputs the

image as a png file to the specified directory.

If you are familiar with Mayavi you will notice that these option are far more limited than what you are used to and that you are not allowed full access to all modules and filters. This is done by design so that the user is only able to access the isosurface and surface modules.

#### Programmatic Implementation

The Show.py script uses a very useful library for Python called OptParse. This library is a powerful command line option parser that handles user input validation, exposure of an automatic help system, displaying versioning information and even allows the user to hook callback functions onto each parameters for customizable validation.

To show how this library can be used we will briefly go over the features that were used to create Show.py.

**from optparse import OptionParser**

**from optparse import OptionValueError**

**from optparse import Option**

**usage = “This is our usage statement”**

**version = “Cool Application v1.0”**

**parser = OptionParser(usage, None, Option, version)**

The previous code snippet shows how to create a new instance of the OptionParser class with a predefined usage statement and the ‘--version' and ‘--help' options enabled. Next we will add a simple command line option.

**parser.add\_option(“-f”, “--file",**

**action=”store”, type=”string”,**

**dest=”filename”,**

**help=”Help information”)**

This line of code tell parser that you are adding a few command line option that can be used by calling program.py –f somefile. A few things in this statement may seem self explanatory such as the type and help but there are quite a few nice features here. For instance the type specified here will actually perform a type check before storing in the destination variable ‘filename’. The destination parameter passed here is another nice feature because now we can access the value by simply calling it from ‘parser.filename’.

Lets add another option for file output but this time we will attach a callback function so that we can perform a check to identify weather the file already exists. We start by defining a function that takes Option, opt, value and parser as the arguments. The arguments are defined as follows.

Option – the instance that is calling the callback.

Opt – The command line option specified and that called the callback.

Value – The value that was specified on the command line for the opt.

Parser – The OptParser instance itself.

**def validatefile(option, opt, value, parser):**

**if not os.path.exists(value):**

**raise OptionValueError, "'%s' does not exist" % (f)**

**else:**

**if not os.path.isfile(f) :**

**raise OptionValueError, "'%s' is not a file" % (f)**

**setattr(parser.values, option.dest, value)**

You will notice that you can inspect the value that user specified and if you were so inclined you can even modify it before passing it to the setattr function that sets the value to the destination.

Now when we add the option we specify that the action value is a callback and add an extra parameter of callback that is set to the name of the function we just created.

**parser.add\_option("-o", "--out",**

**action="callback", type="string",**

**dest="outputfile",**

**callback=validatefile,**

**help="Help Information”)**

We now have a program that will take in to options an input file and an output file and will validate the output file.

Further information on OptParser can be found at python.org (http://docs.python.org/lib/module-optparse.html).

#### Opening file(s) to be rendered

#### Usage

The simplest usage of Show.py is to open a single file for rendering and this can be accomplished by using the command line option “-f” or “--file" as seen below.

>show.py –f myfile.vtk

This will open a single file with the default outline module, orientation axes module and text module activated. If you would like to open multiple files then you can do this with the use of a delimiter as follow.

>show.py –f myfile1.vtk;myfile2.vtk

#### Programmatic Implementation

The opening of a single file in Show.py is quite simple since this uses the power of the VTKFileReader from Enthought. This class is explained in depth above in the section, Expanding on the Mayavi class.

#### Opening file(s) to be rendered and polled

#### Usage

Polling for updates in the files that are rendered is a very nice feature provided by Show.py. This option allows a user to specify file(s) and when the application finds that the file(s) have been modified it will update reload the file so to rerender it on the scene. To do this the “-u” or “--update" option can be used.

>show.py –u myfile.vtk

Similarly to the way in which rendering a single file function the user can poll for multiple files. Use the following to poll for multiple files.

>show.py –u myfile1.vtk;myfile2.vtk

#### Programmatic Implementation

The opening of a file(s) in Show.py for polling is a bit more complex than opening a single file but it is based on the same concepts and uses the same VTKFileReader class as mentioned in Opening file(s) to be rendered section.

Show.py opens the file(s) exactly as it would when it is not polling however it also saves off the file status of the initial file opened as a baseline for later inspection. The file status is obtained using the stat function from the os class and a simple example of its usage can be seen below.

**import os**

**status = os.stat(filename)**

Once that status is obtained Show.py needs a mechanism for monitoring the input file(s) for changes. For this purpose the Watcher class from the poll\_file.py (written by Prabhu Ramachandran of Enthought) script is incorporated. This class is a simple timer that inherits from the wx.Timer class but is overridden to call a callback at the expiration of each interval.

**import wx**

**class Watcher(wx.Timer):**

**def \_\_init\_\_(self, interval, callable, \*args, \*\*kw\_args):**

**wx.Timer.\_\_init\_\_(self)**

**self.callable = callable**

**self.args = args**

**self.kw\_args = kw\_args**

**self.Start(interval)**

**def Notify(self):**

**self.callable(\*self.args, \*\*self.kw\_args)**

For the purpose of polling functionality the instance of the Watcher class is issued an interval of 1 second and given the poll function as its callback. The function poll, as seen below, will loop over every file that was passed in to be rendered and check their corresponding status values. If a status is found to not match then the function proceeds to force the VTKFileReader to update itself by first calling d.reader.modified() and then d.update().

The d.reader.modified function call updates the last modified time for the internal reader object of the VTKFileReader while the d.update call actually forces the update.

**def poll(self):**

**#do poll logic**

**idx = 0**

**Lupdate = []**

**#iterate over files and check their statuses**

**for f in self.filelists[0]:**

**stat = os.stat(f)**

**if self.filestatus[idx][-2] != stat[-2]:**

**Lupdate.append(idx)**

**self.filestatus[idx] = stat**

**idx = idx+1**

**if( len(Lupdate) > 0 ):**

**#take snapshot if update is required**

**self.snapshot()**

**self.sceneTextCount.text = \**

**str(int(self.sceneTextCount.text)+1)**

**#update the files that have changed**

**for update in Lupdate:**

**print "file %s changed" % (self.filelists[0][update])**

**d = self.data[update]**

**d.reader.modified()**

**d.update()**

**d.data\_changed = True**

**Lupdate = None**

With these mechanisms in place the Show.py class is able to maintain up to date renderings for within a second of a file change.

#### Iterating though a list of file(s)

#### Usage

Another nice feature that is provided by Show.py is the ability to pass in a list of files and then proceed to iterate over the list at a one second interval. To do this, use the “-l” or “-loop” option.

>show.py –l myfile\*.vtk

This option also provides the ability to pass in multiple lists of files so files from each list are rendered at a time. Again this can be done by the use of a delimiter between file patters as shown below.

>show.py –l fileA\*.vtk;fileB\*.vtk

#### Programmatic Implementation

Programmatically, the iteration over a list of files is very similar to how polling files for updates functions with one exception. This exceptions being that instead of setting the callback of the Watcher class to the poll function that was previously mentioned in the Opening file(s) to be rendered and polled section, the loop function is used.

The loop function is called every second and performs a check to determine whether there is another file in the list (or lists of file depending on whether multiple list where passed in). If another file is found than it is popped off the list and this file is used to reinitialize the corresponding VTKFileReader object. This re-initialization updates the scene rendering and maintains usage of the existing modules that were previously assigned.

**def loop(self):**

**#loop file logic**

**idx = 0**

**atLeastOneFileChanged = False**

**for filelist in self.filelists:**

**if len(filelist) > 0:**

**self.data[idx].initialize(filelist.pop())**

**atLeastOneFileChanged = True**

**idx = idx+1**

**if( atLeastOneFileChanged ):**

**#update scene index with the text module**

**strNextVal = str(int(self.sceneTextCount.text)+1)**

**self.sceneTextCount.text = " "**

**self.sceneTextCount.text = strNextVal**

**self.snapshot()**

**else:**

**self.watcher = None #Kill the watcher**

#### Activating Mayavi modules

#### Usage

The Show.py script uses five modules: Outline, OrientationAxes, Text, IsoSurface and Surface. The first three are always set with in the rendered scene while the last two are used only if explicitly passed in, with the text module used twice if the user specifies a title to be used.

For the IsoSurface and Surface modules, these have the option to take in the list of files that will have the module applied to them or the user can simply pass in an asterisk to have the module applied to all files. This is so that the user can specify many files to be rendered and have only a subset use a particular module.

Here are a few examples of how to properly use the IsoSurface and Surface modules. The first two examples demonstrate two different ways in which the user can add both modules to every file in the list of files passed in.

>show.py –f myfile1.vtk;myfile2.vtk –i \* -s \*

>show.py –f “myfile1.vtk;myfile2.vtk” –i “myfile1.vtk;myfile2.vtk” -s “myfile1.vtk;myfile2.vtk”

The next example demonstrates adding the IsoSurface module to only one of the two files passed in while adding the Surface module to both files.

>show.py –f myfile1.vtk;myfile2.vtk –i myfile1.vtk -s \*

Users have one other module at their command, the text module. When the user uses the “-t” or “--text” option the text entered will be displayed as a title on the scene, in the top center of the rendering. An example of this can be seen below.

>show.py –f myfile1.vtk -t “Hello World”

#### Programmatic Implementation

We will not delve much into this topic here as the implementation of modules was coved in-depth in the Experimenting with Modules section of this document. You can refer back to that section as reference but actual code used in Show.py to add these modules is worth look.

**def addmodules(self, idx):**

**#add Outline module**

**o = Outline()**

**mayavi.add\_module(o)**

**#add OrientationAxes module**

**oa = OrientationAxes()**

**mayavi.add\_module(oa)**

**#add IsoSurface module if requested**

**if( not self.isosurface is None ):**

**if( self.isosurface == [] or (len(self.isosurface) > \**

**idx and self.isosurface[idx])):**

**i = IsoSurface()**

**mayavi.add\_module(i)**

**#add Surface module if requested**

**if( not self.surface is None ):**

**if( self.surface == [] or (len(self.surface) > idx \**

**and self.surface[idx])):**

**s = Surface()**

**s.enable\_contours = True**

**s.actor.property.opacity = 0.5**

**mayavi.add\_module(s)**

**#add Text module if requested**

**if( len(self.title) > 0 ):**

**t1 = Text()**

**t1.text = self.title**

**t1.actor.scaled\_text = False**

**t1.actor.text\_property.font\_size = 24**

**mayavi.add\_module(t1)**

**t1.width = 1.0\*t1.actor.mapper.get\_width( \**

**t1.scene.renderer)/t1.scene.renderer.size[0]**

**height = 1.0\*t1.actor.mapper.get\_height( \**

**t1.scene.renderer)/t1.scene.renderer.size[1]**

**t1.x\_position = 0.5 - t1.width/2**

**t1.y\_position = 1-height**

**#add default Text module for indicating scene index**

**self.sceneTextCount = Text()**

**self.sceneTextCount.text = "0"**

**mayavi.add\_module(self.sceneTextCount)**

**self.sceneTextCount.actor.scaled\_text = False**

**self.sceneTextCount.actor.text\_property.font\_size = 24**

**self.sceneTextCount.x\_position = .95**

**self.sceneTextCount.y\_position = .05**

#### Setting up image outputs

#### Usage

As one of Show.py’s command line arguments the user can specify an output directory where images can be stored. The image capture functionality works together with the polling and with iterating through lists of files.

When specifying an output directory with the polling option set an image is captured each time the application finds that the file’s input has been modified and are re-rendered. The following is an example of opening show.py for polling and outputting images to a directory called imageDir.

>show.py –u myfile.vtk –i \* -s \* -o ./imageDir

When specifying an output directory with the loop option set an image is captured each time the application renders its next set of files. The following is an example of opening show.py for looping over a list of files and outputting the images to a directory called imageDir.

>show.py –l myfile\*.vtk –i \* -s \* -o ./imageDir

#### Programmatic Implementation

Show.py uses the snapshot function to capture the image file of the current scene. The snapshot function first checks if the output directory is specified and if so will attempt to take the image in png format.

**def snapshot(self):**

**#take snapshot**

**if len(self.outputdir) > 0: #if a path was entered**

**e = mayavi.engine**

**s = mayavi.engine.current\_scene**

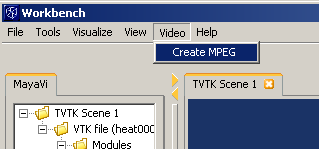
**s.scene.save\_png("%s/output%05d.png" %(self.outputdir, self.imagecount))**

**self.imagecount = self.imagecount + 1**

#### Rendering video

#### Usage

When Show.py is given an output directory where it is to place image file of each change in the rendering a menu item is enabled that allow the user to select to create a video of the images that have been output. The user needs only to select View🡪Create MPEG.



By selecting ‘Create MPEG’, Show.py will compile an mpeg file and output it to the image capture directory.

#### Programmatic Implementation

Getting Show.py to generate a video is quite an easy process since the bulk of the work is performed by an open source utility call mencoder.exe. The mencoder.exe application is a companion program to the Mplayer open source media player.

Through this application we can simply take a list of PNG files and pass them in via command line to compile our video. This is performed in Show.py by the use of the os.system function call, which invokes a command line operation. In this case our code looks as follows.

**os.system("mencoder -ovc lavc -lavcopts " \**

**"vcodec=mpeg1video:vbitrate=2000 " \**

**"-vf scale=320:240 -mf type=png:fps=30 " \**

**"-nosound -of mpeg " \**

**"-o %s/output.mpg mf://\*.png" % (self.outputdir))**

Mencoder is a very rich application unto itself and if you would like more information on this I would suggest the Mplayer website (http://www.mplayerhq.hu) or you can read up on it on one of the many tutorial site available such as the one provided by the U.S. Army Research Laboratory (http://www.arl.hpc.mil/SciVis/mencoder.html).

### Summary

As shown, Mayavi is quite extensible through the use of a MVC based design and has been built upon a number of open technologies, which Mayavi allows great breadth in access to. This makes working with the framework quite enjoyable in its easy of navigation. Furthermore the Show.py script should prove quite convincing that almost every aspect of this framework is programmatically customizable making creating a custom application quick and easy.

It is my hope that the information and examples shown here will give you a firm foundation to get started at hacking away at Mayavi and learning more about this technology.

### References

[1] Prabhu Ramachandran “MayaVi-2: The next generation” Europython 2005 presentation (07/04/06), [https://svn.enthought.com/enthought/attachment/wiki/MayaVi/mayavi2-epc05.pdf?format=raw]

[2] Prabhu Ramachandran “MayaVi-2: The next generation” m2-paper-epc2005.pdf (6/19/2005) URL [https://svn.enthought.com/enthought/attachment/wiki/MayaVi/m2-paper-epc2005.pdf?format=raw]

[3] MayaVi – Enthought – Trac (2007) MayaVi2: the next generation [WWW document]. URL https://svn.enthought.com/enthought/wiki/MayaVi

[4] Python – The Python Programming Language [WWW document]. URL http://python.org/

[5] U.S. Army Research Laboratory, Major Shared Resource Center [WWW document] URL http://www.arl.hpc.mil/SciVis/mencoder.html

[6] Mplayer – The Movie Player [WWW document]

URL http://www.mplayerhq.hu

[7] wxPython - wxPython [WWW document]

URL http://www.wxpython.org/

[8] Scipy.org – Cookbook [WWW document]

URL http://new.scipy.org/Wiki/Cookbook

### Appendix I: Mayavi Architecture Overview

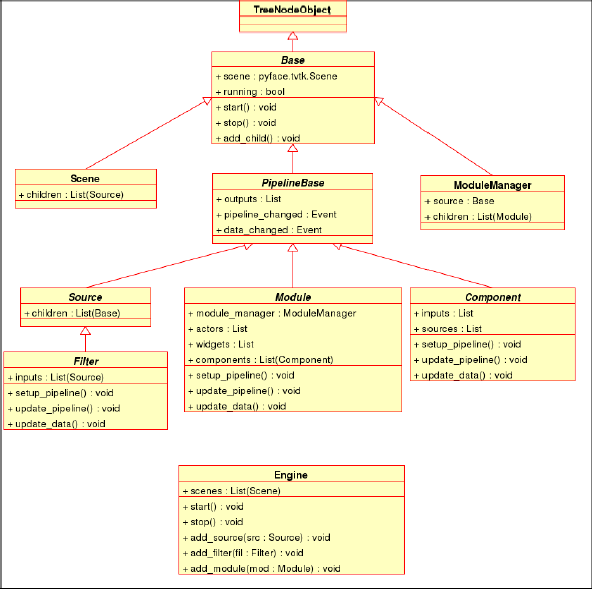


Image from: MayaVi-2: The next generation by Prabu Ramachandra

https://svn.enthought.com/enthought/attachment/wiki/MayaVi/m2-paper-epc2005.pdf?format=raw