tvtk Documentation

Release 3.1.0

tvtk

December 03, 2008

CONTENTS

Contents:

CHAPTER

ONE

An Introduction to Traited VTK (tvtk)

Author Prabhu Ramachandran

Contact prabhu_r@users.sourceforge.net

Contents

1.1 Introduction

The tvtk module (also called TVTK) provides a traits enabled version of VTK. TVTK objects wrap around VTK objects but additionally support traits, and provide a convenient Pythonic API. TVTK is implemented mostly in pure Python (except for a small extension module). Here is a list of current features.

- All VTK classes are wrapped.
- Classes are generated at install time on the installed platform.
- Support for traits.
- Elementary pickle support.
- Pythonic feel.
- Handles numpy arrays/Python lists transparently.
- Support for a pipeline browser, *ivtk* and a high-level *mlab* like module.
- Envisage plugins for a tvtk scene and the pipeline browser.
- tvtk is free software with a BSD style license.

1.2 Requirements

The following is a list of requirements for you to be able to run tvtk.

- Python-2.3 or greater.
- Python should be built with zlib support and must support ZIP file imports.
- VTK-5.x, VTK-4.4 or VTK-4.2. It is unlikely to work with VTK-3.x.
- Traits, version 2.
- numpy – Any recent version should be fine.
- To use *ivtk.py*, *mlab* you need to have *pyface* installed.
- To use the plugins you need Envisage installed.

1.3 Installation

TVTK is meant to be installed as part of the *enthought* package. The *setup_tvtk.py* file sets up TVTK. So installing the *enthought* package should also install tvtk for you. For an inplace build of the Enthought tool suite please read: inplace build of the Enthought tool suite.

This document only covers building and using TVTK from inside a checkout of the the enthought SVN repository. The tvtk module lives inside *enthought.tvtk*. To build the tvtk module and use them from inside the *enthought* sources do the following from the base of the enthought source tree (we assume here that this is in */home/svn/enthought/src/lib/enthought*):

```
$ pwd
/home/svn/enthought/src/lib/enthought
$ cd tvtk
$ python setup_tvtk.py build_ext --inplace
```

The code generation will take a bit of time. On a PentiumIII machine at 450Mhz, generating the code takes about a minute. If the code generation was successful you should see a ZIP file called *tvtk_classes.zip* in the tvtk directory along with an extension module.

This completes the installation. The build can be tested by running the tests in the *tests/* directory. This tests the built code:

```
$ pwd
/home/svn/enthought/tvtk
$ cd tests
$ python test_tvtk.py
[...]
```

If the tests run fine, the build is good. There are other tests in the *tests* directory that can also be run.

Documentation is available in the 'doc/' directory. The 'examples/' directory contains a few simple examples demonstrating tvtk in action.

1.4 Basic Usage

An example of how tvtk can be used follows:

```
>>> from enthought.tvtk.api import tvtk
>>> cs = tvtk.ConeSource()
>>> cs.resolution = 36
>>> m = tvtk.PolyDataMapper()
>>> m.input = cs.output
>>> a = tvtk.Actor()
>>> a.mapper = m
>>> p = a.property
>>> p.representation = 'w'
>>> print p.representation
'wireframe'
```

Or equivalently:

```
>>> from enthought.tvtk.api import tvtk
>>> cs = tvtk.ConeSource(resolution=36)
>>> m = tvtk.PolyDataMapper(input=cs.output)
>>> p = tvtk.Property(representation='w')
>>> a = tvtk.Actor(mapper=m, property=p)
```

Note that the properties of the object can be set during the instantiation.

To import tvtk please use:

```
from enthought.tvtk.api import tvtk
```

While this is perhaps a little inconvenient, note that tvtk provides access to all the VTK classes. This is the same way that the *vtk* package also behaves.

If you are familiar with VTK-Python it is clear from the above example that tvtk "feels" like VTK but is more Pythonic. The most important differences are.

1. tvtk class names are essentially similar to VTK classes except there is no annoying 'vtk' at the front. The only difficulty is with classes that start with a digit. For example 'vtk3DSImporter' becomes '3DSImporter'. This is illegal in Python and therefore the class name used is 'ThreeDSImporter'. So, if the first character is a digit, it is replaced by an equivalent non-digit string. There are very few classes like this so this is not a big deal.
2. tvtk method names are *enthought* style names and not CamelCase. That is, if a VTK method is called *AddItem*, the equivalent tvtk name is *add_item*. This is done for the sake of consistency with names used in the *enthought* package.
3. Many VTK methods are replaced by handy properties. In the above example, we used *m.input = cs.output* and *p.representation = 'w'* instead of what would have been *m.SetInput(cs.GetOutput())* and *p.SetRepresentationToWireframe()* etc. Some of these properties are really traits.
4. Unlike VTK objects, one can set the properties of a tvtk object when the object is initialized by passing the properties (traits) of the object as keyword arguments at the time of class instantiation. For example *cs = tvtk.ConeSource(radius=0.1, height=0.5)*.

If you are used to VTK, this might take a little getting used to. However, these changes are consistent across all of tvtk. If they aren't, its a bug. Please let me know if you see inconsistencies.

If the underlying VTK object returns another VTK object, this is suitably wrapped as a tvtk object. Similarly, all relevant parameters for a tvtk method should be tvtk objects, these are transparently converted to VTK objects.

1.5 Advanced Usage

There are several important new features that tvtk provides in addition to the above. A tvtk object basically wraps around a VTK-Python object and provides a trait enabled API for the VTK object. Before we discuss these new features it is important to understand the notion of what we mean by the "basic state" or "state" of a tvtk object. This is defined and subsequently the new features are discussed in some detail.

1.5.1 Definition of the "basic state" of a tvtk object

In tvtk the set of all properties of the VTK object that are represented as traits and have for their value a simple Python type (int/float/string) or a special value (like a tuple specifying color) define the state.

In terms of the implementation of tvtk, any property of a VTK object that can be set by using methods having the form *<Property>On* or *<Property>Off*, *Set<Property>To<Value>* and *Set/Get<Property>* (where the return type is an int/float/string/tuple) are represented as traits. These properties are said to represent the "basic state" of the tvtk object.

Note that the complete state of the underlying C++ object is impossible to represent in the Python world since this usually involves various pointers to other C++ objects.

1.5.2 The wrapped VTK object

The user should not ordinarily know this (or rely on this!) but it sometimes helps to know how to access the underlying VTK object that the tvtk object has wrapped. The recommended way to do this is by using the *to_vtk* function. For example:

```
>>> pd = tvtk.PolyData()
>>> pd_vtk = tvtk.to_vtk(pd)
```

The inverse process of creating a tvtk object from a VTK object is to use the *to_tvtk* function like so:

```
>>> pd1 = tvtk.to_tvtk(pd_vtk)
>>> print pd1 == pd
True
```

Notice that *pd1 == pd*. TVTK maintains an internal cache of existing tvtk objects and when the *to_tvtk* method is given a VTK object it returns the cached object for the particular VTK object. This is particularly useful in situations like this:

```
>>> cs = tvtk.ConeSource()
>>> o = cs.output
>>> m = tvtk.PolyDataMapper()
>>> m.input = o
>>> # ...
>>> print m.input == o
True
```

It must be noted that if a tvtk object's goes out of scope in Python, it is garbage collected. However, its underlying VTK object may still exist inside the VTK pipeline. When one accesses this object, a new tvtk wrapper object is created. The following illustrates this:

```
>>> cs = tvtk.ConeSource()
>>> o = cs.output
>>> m = tvtk.PolyDataMapper()
>>> m.input = o
>>> print hash(o)
1109012188
>>> print hash(m.input)
1109012188
>>> del o
>>> print hash(m.input)
1119694156
```

Thus, after *o* is garbage collected *m.input* no longer refers to the original tvtk object and a new one is created. This is very similar to VTK's behaviour. The reason that this is difficult to do correctly is that VTK objects cannot be weakly referenced. In the future, when VTK supports weak references (possibly in version 5.0), this can be fixed.

1.5.3 tvtk and traits

All tvtk objects are derived from *traits.HasStrictTraits*. As discussed above, all the basic state related methods are represented as traits in tvtk. This is why we are able to do:

```
>>> p = a.property
>>> p.representation = 'w'
>>> print p.representation
'wireframe'
>>> # OR do this:
>>> p = tvtk.Property(opacity=0.5, color=(1,0,0), representation='w')
```

Also note that it is possible to set many properties of a tvtk object in one go using the *set* method. For example:

```
>>> p = tvtk.Property()
>>> p.set(opacity=0.5, color=(1,0,0), representation='w')
```

Any tvtk object will automatically provide the basic functionality of a traited class. Thus, one can also pop up a standard GUI editor for any tvtk object trivially. For example if one is using *pycrust* or is using *gui_thread* (this module should be available if SciPy is installed) or *ipython -wthread* one could easily do this:

```
>>> p = tvtk.Property()
>>> p.edit_traits()
>>> # OR
>>> p.configure_traits() # This should work even without gui_thread
```

A GUI editor should pop-up at this point. Note, that changes made to the trait will automagically be propagated to the underlying VTK object. Most importantly, the reverse is also true. That is, if some other object changes the basic state of the wrapped VTK object, then the trait will be automagically updated. For example:

```
>>> p = tvtk.Property()
>>> print p.representation
'surface'
>>> p_vtk = tvtk.to_tvtk(p)
>>> p_vtk.SetRepresentationToWireframe()
>>> print p.representation
'wireframe'
```

This also means that if you change properties of the object on the interpreter and at the same time are using a GUI editor, if the object changes, the GUI editor will update automatically.

It is important to note that tvtk objects have strict traits. It is therefore an error to set an attribute that is not already defined in the class. This is illustrated in the following example:

```
>>> cs = tvtk.ConeSource()
>>> cs.foo = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TraitError: Cannot set the undefined 'foo' attribute of a 'ConeSource' object.
```

1.5.4 Sub-classing tvtk classes

You may subclass tvtk classes but please keep in mind the following:

1. All tvtk classes derive from *HasStrictTraits*.
2. You must make sure to call the super class's __init__ correctly.
3. If you have to override the __del__ method, you *must* make sure you call the super class's __del__ method.

1.5.5 Pickling tvtk objects

tvtk objects support a simple form of pickling. The state of the tvtk object maybe pickled. The references to other VTK objects held by the object are *NOT* picklable. For example:

```
>>> import cPickle
>>> p = tvtk.Property()
>>> p.representation = 'w'
>>> s = cPickle.dumps(p)
>>> del p
>>> p = cPickle.load(s)
>>> print p.representation
'wireframe'
```

Once again, only the state of the object is pickled. Internal references are not. So the construction of the VTK pipeline will not be pickled. For example if we pickled the actor from the example given in the Basic Usage section, then the *ConeSource*, *PolyDataMapper* etc. will not be pickled. Some of the tvtk classes like *Matrix4x4* also have special code that enables better pickling.

It is also possible to set the state of a live object. Normally, *pickle.load* will create a new object. However, by using *__setstate__* directly it is possible to just update the state of the object. For example:

```
>>> p = tvtk.Property()
>>> p.interpolation = 'flat'
>>> d = p.__getstate__()
>>> del p
>>> p = Prop()
>>> p.__setstate__(d)
```

Here, after *__setstate__* is called, the object's state alone will be updated. No new object is created.

1.5.6 Docstrings

All tvtk methods and traits are documented using the VTK docstrings. The class names and method names are changed suitably as mentioned in the Basic Usage section. These docstrings are generated automatically and there might be small mistakes in them. All methods of a tvtk object also provide method signature information in the docstring.

1.5.7 User defined code

All the tvtk wrapper classes are generated automatically. Sometimes one would like to customize a particular class and use that instead of the default. The easiest way to do this would be to copy out the relevant class from the *tvtk_classes.zip* file, modify it suitably (without changing the name of the file or class name of course) and then add this file into the *tvtk/custom* directory. Any file here will override the default inside the ZIP file.

1.5.8 Collections

Any object derived from *Collection* (i.e. *vtkCollection*) will behave like a proper Python sequence. Here is an example:

```
>>> ac = tvtk.ActorCollection()
>>> print len(ac)
0
>>> ac.append(tvtk.Actor())
>>> print len(ac)
1
>>> for i in ac:
...     print i
...
[...]
```

```
>>> ac[-1] = tvtk.Actor()
>>> del ac[0]
>>> print len(ac)
0
```

Currently, only subclasses of *Collection* behave this way.

1.5.9 Array handling

All the *DataArray* subclasses behave like Pythonic arrays and support the iteration protocol in addition to *__getitem__*, *__setitem__*, *__repr__*, *append*, *extend* etc. Further, it is possible to set the value of the array using either a numpy array or a Python list (using the *from_array* method). One can also get the data stored in the array into a numpy array (using the *to_array* method). Similarly, the *Points* and *IdList* classes also support these features. The *CellArray* class only provides the *from_array* and *to_array* methods and does not provide a sequence like protocol. This is because of the peculiarity of the wrapped *vtkCellArray* class.

One extremely useful feature is that almost any tvtk method/property that accepts a *DataArray*, *Points*, *IdList* or *CellArray* instance, will transparently accept a numpy array or a Python list. Here is a simple example demonstrating these:

```
>>> ########################################
>>> from enthought.tvtk.api import tvtk
>>> import numpy
>>> data = numpy.array([[0,0,0,10], [1,0,0,20],
...                     [0,1,0,20], [0,0,1,30]], 'f')
>>> triangles = numpy.array([[0,1,3], [0,3,2],
...                          [1,2,3], [0,2,1]])
>>> points = data[:,:3]
>>> temperature = data[:,-1]
>>> mesh = tvtk.PolyData()
>>> mesh.points = points
>>> mesh.polys = triangles
>>> mesh.point_data.scalars = temperature

>>> ########################################
>>> # Array's are Pythonic.
>>> import operator
>>> reduce(operator.add, mesh.point_data.scalars, 0.0)
80.0
>>> print mesh.point_data.scalars
[10.0, 20.0, 20.0, 30.0]
>>> print mesh.points
[(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)]

>>> ########################################
>>> # Demo of from_array/to_array
>>> pts = tvtk.Points()
>>> pts.from_array(points)
>>> print pts.to_array()
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

As can be seen, no *DataArray*, *Points* or *CellArray* instances need to be created. Note that Python tuples are *not* converted implicitly. The conversion from the passed arrays to the VTK arrays is handled transparently and very

efficiently. The only exception to this is the *IdList* class where the conversion is inefficient. However, the *IdList* class is not used commonly.

The *CellArray* is used to specify the connectivity list for polygonal data and has some peculiarities. The *CellArray* needs to be initialized using the cell connectivity list. This can be specified in one of several ways.

1. A Python list of 1D lists. Each 1D list can contain one cell connectivity list. This is very slow and is to be used only when efficiency is of no consequence.
2. A 2D numpy array with the cell connectivity list.
3. A Python list of 2D numpy arrays. Each numpy array can have a different shape. This makes it easy to generate a cell array having cells of different kinds.

This conversion is most efficient if the passed numpy arrays have a typecode of *enthought.tvtk.array_handler.ID_TYPE_CODE*. Otherwise a typecast is necessary and this involves an extra copy. The input data is *always copied* during the conversion. Here is an example illustrating these different approaches:

```
>>> a = [[0], [1, 2], [3, 4, 5], [6, 7, 8, 9]]
>>> cells = tvtk.CellArray()
>>> cells.from_array(a)
>>> a = numpy.array([[0,1,2], [3,4,5], [6,7,8]], int)
>>> cells.from_array(a)
>>> l_a = [a[:,:1], a[:2,:2], a]
>>> cells.from_array(a)
```

An alternative way to use an arbitrary connectivity list having different numbers of points per cell is to use the following approach:

```
>>> ids = numpy.array([3, 0,1,3,
...                    3, 0,3,2,
...                    3, 1,2,3,
...                    3, 0,2,1])
>>> # The list is of form [npts,p0,p1,...p(npts-1), ...]
>>> n_cell = 4
>>> cells = tvtk.CellArray()
>>> cells.set_cells(n_cell, ids)
>>> print cells.data
[3.0, ..., 1.0], length = 16
```

This is done very efficiently and does not copy the input data. More details on this are provided in the next sub-section.

Also note that *DataArray* objects can still be passed to these methods as before. For example we could have just as easily done this:

```
>>> points = tvtk.Points()
>>> points.from_array(data[:,:3])
>>> temperature = tvtk.FloatArray()
>>> temperature.from_array(data[:,-1])
>>> cells = tvtk.CellArray()
>>> cells.set_cells(n_cell, ids)
>>> mesh = tvtk.PolyData()
>>> mesh.points = points
>>> mesh.polys = cells
>>> mesh.point_data.scalars = temperature
```

Important considerations

To clarify the ensuing discussion we make a distinction between two different forms of array handling.

a. **Explicit conversion – These happen when the user is creating a** tvtk array object and initializes this from a numpy array or Python list. Like so:

```
>>> f = tvtk.FloatArray()
>>> a = numpy.array([1,2,3], int)
>>> f.from_array(a)
```

b. **Implicit conversion – These happen when the user passes an array** or list to a tvtk method that expects a *DataArray*, *Points*, *IdList* or *CellArray* instance.

There are a few issues to keep in mind when using tvtk's array handling features. When possible, tvtk uses a view of the passed numpy array and does not make a copy of the data stored in it. This means that changes to the VTK data array or to the numpy array are visible in the other. For example:

```
>>> f = tvtk.FloatArray()
>>> a = numpy.array([1,2,3], 'f')
>>> f.from_array(a)
>>> a[0] = 10.0
>>> print f
[10.0, 2.0, 3.0]
>>> f[0] = -1.0
>>> print a
[-1.  2.  3.]
```

It is important to note that it is perfectly safe to delete the reference to the numpy array since this array is actually cached safely to eliminate nasty problems. This memory is freed when the VTK array is garbage collected. Saving a reference to the numpy array also ensures that the numpy array cannot be resized (this could have disastrous effects).

However, there are exceptions to this behaviour of using "views" of the numpy array. The *DataArray* class and its subclasses and the *Points* class only make copies of the given data in the following situations.

1. A Python list is given as the data.
2. A non-contiguous numpy array is given.
3. The method requiring the conversion of the array to a VTK array expects a *vtkBitArray* instance.
4. The types of the expected VTK array and the passed numpy array are not equivalent to each other. For example if the array passed has typecode 'i' but the tvtk method expects a *FloatArray*.

The cases 3 and 4 occur very rarely in implicit conversions because most methods accept *DataArray* instances rather than specific subclasses. However, these cases are likely to occur in explicit conversions.

CellArray always makes a copy of the data on assignment. For example:

```
>>> ca = tvtk.CellArray()
>>> triangles = numpy.array([[0,1,3], [0,3,2],
...                          [1,2,3], [0,2,1]])
>>> ca.from_array(triangles)
```

This always makes a copy. However, if one uses the *set_cells* method a copy is made in the same circumstances as specified above for *DataArray* and *Points* classes. If no copy is made, the cell data is a "view" of the numpy array. Thus, the following example does not make a copy:

```
>>> ids = numpy.array([3, 0,1,3,
...                    3, 0,3,2,
...                    3, 1,2,3,
...                    3, 0,2,1], int)
>>> ca.set_cells(4, ids)
```

Changing the values of the ids or changing the number of cells is *not* recommended and will lead to undefined behaviour. It should also be noted that it is best to pass cell connectivity data in arrays having typecode *enthought.tvtk.array_handler.ID_TYPE_CODE* (this is actually computed dynamically depending on your VTK build and platform).

The *IdList* also *always* makes a copy of the data passed to it.

Another issue to keep in mind is that VTK's data arrays always re-allocate memory if they are resized. This is illustrated in the following example:

```
>>> d = tvtk.DoubleArray()
>>> a = numpy.array([1,2,3], 'd')
>>> d.from_array(a)
>>> a[0] = 10
>>> d.append(4.0)
>>> a[0] = 1
>>> print a
[ 1.   2.   3.]
>>> print d
[10.0, 2.0, 3.0, 4.0]
>>> # Notice that d[0] == 10.0
```

In this case, *a* is not resized but *d* is. Here, *d* actually makes a copy of *a* and any further changes to *d* or *a* will not be reflected in the other. This case also illustrates a small problem. *d* will hold a reference to *a* internally even though it uses none of *a*'s memory. Fortunately, when *d* is garbage collected the memory occupied by *a* will be freed. Thus the problem is not serious but probably worth keeping in mind.

Summary of issues

To summarize the considerations of the previous sub-section, the following are to be noted.

1. Most often *DataArray* and *Points* objects do not make copies of the numpy data. The exceptions are listed above. This means changes to either the tvtk object or the array are reflected in the other.
2. It is safe to delete references to the array object converted. You cannot resize the numpy array though.
3. *CellArray* always copies data on assignment. However, when using *set_cells*, the behaviour is similar to what happens for *DataArray* objects. Note that it is not advisable to change the connectivity ids and number of cells when this is done. Also note that for the *CellArray* it is best to pass data in the form of numpy arrays having a typecode of *enthought.tvtk.array_handler.ID_TYPE_CODE*). Otherwise one incurs an extra copy due to a typecast.
4. *IdList* always makes a copy of the data. This class is very rarely used.
5. tvtk array objects *always* copy the data when resized. This could lead to increased memory usage in some circumstances. However, this is *not* a memory leak.

The upshot of these features is that array conversion can be extremely efficient in terms of speed and memory considerations.

1.6 Other utility modules

The *tvtk* package ships with several other utility modules. These are briefly described in the following sections.

1.6.1 *pipeline.browser*

The *PipelineBrowser* class presents the view of the VTK pipeline as a tree. Double-clicking any node will let you edit the properties of the object with a trait sheet editor. The *TreeEditor* from the traits package is used to represent the view. This pipeline browser is similar to but more sophisticated than MayaVi's (1.x) pipeline browser. The browser will most often automatically update itself as you change the VTK pipeline. When it does not you can right click on any node and click refresh.

The algorithm to generate the objects in the tree can be changed. The user may subclass *TreeGenerator* and use that instead. Please read the code and docstrings for more details on the implementation.

1.6.2 *tools.ivtk*

A utility module that makes VTK/TVTK easier to use from the Python interpreter. The module uses the *tvtk.scene* module to provide a wxPython widget. ivtk basically provides this scene along with an optional Python interpreter (via PyCrust) and an optional pipeline browser view.

For a stand-alone application one may simply run the module. To use this under IPython (with -wthread) use the *viewer()* helper function. For example:

```
>>> from enthought.tvtk.tools import ivtk
>>> from enthought.tvtk.api import tvtk
>>> # Create your actor ...
>>> a = tvtk.Actor()
>>> # Now create the viewer.
>>> v = ivtk.viewer()
>>> v.scene.add_actors(a)   # or v.scene.add_actor(a)
```

ivtk provides several useful classes that you may use from either PyFace or wxPython – *IVTK*, *IVTKWithCrust*, *IVTKWithBrowser* and *IVTKWithCrustAndBrowser*. Again read the code and docstrings to learn more. An example using ivtk is also available in *examples/ivtk_example.py*.

1.6.3 *tools.mlab*

A module that provides Matlab-like 3d visualization functionality. The general idea is shamelessly stolen from the high-level API provided by Octaviz. Some of the test cases and demos are also translated from there!

The implementation provided here is object oriented and each visualization capability is implemented as a class that has traits. So each of these may be configured. Each visualization class derives (ultimately) from MLabBase which is responsible for adding/removing its actors into the render window. The classes all require that the RenderWindow be a *enthought.tvtk.scene.Scene* instance (this constraint can be relaxed if necessary later on).

This module offers the following broad class of functionality:

Figure This basically manages all of the objects rendered. Just like figure in any Matlab like environment. A convenience function called *figure* may be used to create a nice Figure instance.

Glyphs This and its subclasses let one place glyphs at points specified as inputs. The subclasses are: *Arrows*, *Cones*, *Cubes*, *Cylinders*, *Spheres*, and *Points*.

Line3 Draws lines between the points specified at initialization time.

Outline Draws an outline for the contained objects.

Title Draws a title for the entire figure.

LUTBase Manages a lookup table and a scalar bar (legend) for it. This is subclassed by all classes that need a LUT.

SurfRegular MayaVi1's *imv.surf* like functionality that plots surfaces given x (1D), y(1D) and z (or a callable) arrays.

SurfRegularC Also plots contour lines.

TriMesh Given triangle connectivity and points, plots a mesh of them.

FancyTriMesh Plots the mesh using tubes and spheres so its fancier.

Mesh Given x, y generated from *numpy.mgrid*, and a z to go with it. Along with optional scalars. This class builds the triangle connectivity (assuming that x, y are from *numpy.mgrid*) and builds a mesh and shows it.

FancyMesh Like mesh but shows the mesh using tubes and spheres.

Surf This generates a surface mesh just like Mesh but renders the mesh as a surface.

Contour3 Shows contour for a mesh.

ImShow Allows one to view large numpy arrays as image data using an image actor. This is just like MayaVi1's *mayavi.tools.imv.viewi*.

To see nice examples of all of these look at the *test_** functions at the end of the mlab.py file. Here is a quick example that uses some of these test functions:

```
>>> from enthought.tvtk.tools import mlab
>>> f = mlab.figure()
>>> mlab.test_surf(f) # Create a spherical harmonic.
>>> f.pop() # Remove it.
>>> mlab.test_molecule(f) # Show a caffeine molecule.
>>> f.renwin.reset_zoom() # Scale the view.
>>> f.pop() # Remove this.
>>> mlab.test_lines(f) # Show pretty lines.
>>> f.clear() # Remove all the stuff on screen.
```

Here is the *test_surf* function just to show you how easy it is to use *mlab*:

```
>>> # Create the spherical harmonic.
>>> from numpy import pi, cos, sin, mgrid
>>> dphi, dtheta = pi/250.0, pi/250.0
>>> [phi,theta] = mgrid[0:pi+dphi*1.5:dphi,0:2*pi+dtheta*1.5:dtheta]
>>> m0, m1, m2, m3, m4, m5, m6, m7 = 4, 3, 2, 3, 6, 2, 6, 4
>>> r = sin(m0*phi)**m1 + cos(m2*phi)**m3 + sin(m4*theta)**m5 + cos(m6*theta)**m7
>>> x = r*sin(phi)*cos(theta)
>>> y = r*cos(phi)
>>> z = r*sin(phi)*sin(theta)
>>> # Now show the surface.
>>> from enthought.tvtk.tools import mlab
>>> fig = mlab.figure()
>>> s = Surf(x, y, z, z)
>>> fig.add(s)
```

As you may notice, this example has also been translated from the Octaviz site.

1.6.4 *plugins*

TVTK ships with two Envisage plugins. One for a TVTK scene and another for the pipeline browser.

The *scene* plugin allows one to create a new TVTK scene on the work area. Any number of these may be created. It provides useful menu's to set the view of the scene (like the ivtk menus). It also allows the user to save the view to an image. The plugin also provides a few preferences to set the background color of the window etc.

The *browser* plugin places a pipeline browser on the left of the Envisage window. This browser is hooked up to listen to scene additions to the work area. Each time a scene is added it will show up as a top-level node in the browser.

These plugins may be used from any Envisage plugin. To see a fully functional example of this please look at the *examples/plugin/* directory. The example demonstrates how to use the plugins in Envisage and make an application.

CHAPTER

TWO

Development guide for TVTK

Author Prabhu Ramachandran

Contact prabhu_r@users.sourceforge.net

Contents

2.1 Introduction

This document provides some details on how TVTK works and how it is implemented. Basic documentation on the installation, features and usage of tvtk is provided in the README.txt file that should be in the same directory as this file.

2.2 How tvtk works

All tvtk classes derive from *TVTKBase*. This is defined in *tvtk_base.py*. *TVTKBase* provides basic functionality common to all tvtk classes. Most importantly it provides for the following:

- Allows one to wrap an existing VTK object or create a new one. *TVTKBase.__init__* accepts an optional VTK object, that can be used to wrap an existing VTK object. Read the method docstring for more details.
- Sets up the event handlers that allow us to listen to any changes made to the underlying VTK object.

- Basic support for pickling the object.
- The *update_traits* automatically updates the traits of the tvtk class such that they reflect the state of the underlying VTK object.
- Common code to change the underlying VTK object when the trait is changed.

As mentioned above, tvtk classes can either wrap an existing VTK object or create a new one and manage the new object.

2.2.1 Automatic trait updation

One very important feature of tvtk objects is that any VTK related traits of object are dynamically updated if the underlying VTK object is changed. This is achieved by adding an observer to the VTK object that effectively calls TVTKBase.update_traits when the 'ModifiedEvent' is invoked. Here is an example of a VTK observer callback:

```
>>> import vtk
>>> p = vtk.vtkProperty()
>>> def cb(obj, evt):
...     print obj.__class__.__name__, evt
...
>>> p.AddObserver("ModifiedEvent", cb)
1
>>> p.SetRepresentationToWireframe()
vtkOpenGLProperty ModifiedEvent
```

As can be seen, when the property is modified, the callback is called. The trouble with this approach is that *p* now holds a reference to *cb*. Thus if *cb* were the *update_traits* method of the tvtk class that manages the VTK object, we run into a non-garbage-collectable reference cycle issue and a huge memory leak. To get around this we use the functionality provided by *messenger.py*. Effectively, *TVTKBase.__init__* tells messenger to call *self.update_traits* when it is called back from the VTK object. Messenger itself uses *weakrefs* so the reference cycle problem does not exist. The VTK object basically calls *messenger.send* and *messenger* takes care of the rest. This allows the tvtk class to be safely used and also allows for automatic trait updation.

Each tvtk class has an attribute called *_updateable_traits_*. This is a tuple of tuples. Each tuple contains a pair of strings like so:

```
>>> p = tvtk.Property()
>>> p._updateable_traits_[:4]
(('opacity', 'GetOpacity'),
 ('frontface_culling', 'GetFrontfaceCulling'),
 ('point_size', 'GetPointSize'),
 ('specular_color', 'GetSpecularColor'))
```

The first string is the name of the trait and the second the name of the function used to obtain the value of the trait from the VTK object. *TVTKBase.update_traits* uses this tuple and for each trait it calls the relevant VTK 'get' function and updates the value of the trait.

While this whole process seems to be inefficient, it actually works quite well in practice and does not appear to be slow at all.

2.2.2 Messaging

The messaging functionality is defined in *tvtk/messenger.py*. Unit tests are in the 'tests/' directory. As described in the previous section, tvtk objects use the messenger functionality to ensure that the traits are always up to date.

messenger.py is well documented and tested. So please read the docstrings for more details. Here is some basic information from the documentation.

messenger.py implements a simple, robust, safe, *Messenger* class that allows one to register callbacks for a signal/slot (or event/handler) kind of messaging system. One can basically register a callback function/method to be called when an object sends a particular event. The Messenger class is Borg. So it is easy to instantiate and use. This module is also reload-safe, so if the module is reloaded the callback information is not lost. Method callbacks do not have a reference counting problem since weak references are used. The main functionality is provided by three functions, *connect*, *disconnect* and *send*.

connect basically allows one to connect an object, *obj*, that emits an event, *event* to a callback function, *callback*.

When an object calls *send*, it passes itself, the event it wishes to emit along with any optional arguments and keyword arguments to send. All registered *callback* for the particular event are called with the the object that called it (*obj*), the event (*event*) and the optional arguments.

disconnect allows one to disconnect either a particular callback, or all callbacks for a particular event or the entire object.

The messenger class thus provides a simple observer pattern implementation that makes it easy to do inter-object communication. For further details please read the code and docstrings. Also look at the test suite in *tests/test_messenger.py*.

2.2.3 Pickling

TVTKBase provides the basic functionality for pickling. Essentially, the dictionary (*self.__dict__*) of the tvtk object is pickled. The *_vtk_obj* attribute is not pickled since VTK objects are not picklable. Other irrelevant attributes are also removed. *__setstate__* works even on a live object by checking to see if *self._vtk_obj* exists. Some of the tvtk classes override these methods to implement them better. For example look at the code for the *Matrix4x4* class.

2.2.4 Array handling

TVTK lets the user pass Numeric arrays/Python lists in place of the *DataArray*, *CellArray*, *Points* etc. classes. This is done by finding the call signature of the wrapped VTK method, checking to see if it has an array and then intelligently generating the appropriate VTK array data using the signature information. This generated VTK data array is then passed to the wrapped function under the covers. This functionality also works if the method has multiple call signatures (overloaded VTK method) and one of them involves arrays. The main functions to do this conversion magic are in the *array_handler.py* module. The *wrapper_gen.py* module generates the appropriate wrapper code depending on the call signature. So if a method has no VTK objects in its signature, no wrapping is necessary. If it only has non-array VTK objects, then array handling is unnecessary but one must dereference the VTK object from the passed tvtk wrapper object. If the call signature involves VTK array objects, then any Numeric arrays or Python lists are converted to the correct VTK data array and passed to the wrapped VTK object.

For the efficient conversion to *CellArray* objects, an extension module is required. The sources for this extension module are in *src/array_ext.**. The Pyrex source file can be used to generate the *src/array_ext.c* file like so:

```
$ cd src
$ pyrexc array_ext.pyx
```

The sources ship with the generated source file so that folks without Pyrex can build it.

special_gen.py defines all the additional methods for the *DataArray* and other classes that allow these objects to support iteration, *__repr__*, *__getitem__* etc.

2.2.5 The magical *tvtk* instance

When one does the following:

```
>>> from enthought.tvtk.api import tvtk
```

The imported *tvtk* is really an instance of a class and not a module! This is so because, the tvtk classes are almost 800 in number and they wrap around the *entire* VTK API. VTK itself allows one to import all the defined classes in one go. That is, if one does *import vtk*, one can instantiate all VTK related classes. This is very useful functionality. However, importing 800 wrapper classes in one go is *extremely* slow. Thus, *tvtk* provides a convenient solution to get around the problem. While *tvtk* is a class instance, it behaves just like you'd expect of a module.

- It supports tab-completion on class names.
- Allows one to access all the tvtk classes.
- Imports extremely fast. Any slowness in the import is really because tvtk needs to import VTK. Thus importing tvtk should not be much slower than importing vtk itself.
- Imports the tvtk classes *only* on demand.

All the tvtk related code is generated and saved into the *tvtk_classes.zip* file. The *tvtk* instance is basically an instance of the TVTK class. This is defined in *tvtk_helper.py*. This file is inside the ZIP file. For each wrapped VTK class, TVTK defines a property having the name of the wrapper tvtk class. The *get* method of the property basically returns the class after importing it dynamically from the ZIP file. This is done in the *get_class* function. Each class thus imported is cached (in *_cache*) and if a cached copy exists the class is not re-imported. Note that in the implementation, the class and its parent classes need to be imported. These parent classes are also cached.

enthought/tvtk/__init__.py adds the *tvtk_classes.zip* file into *sys.path* and instantiates the TVTK class to provide the *tvtk* instance.

Note that any files inside the *tvtk_classes.zip* are automatically generated.

2.2.6 Miscellaneous details

tvtk_base.py defines some important and frequently used traits that are used in tvtk. It is used by all the tvtk classes. It provides a couple of useful functions (*get_tvtk_name* and *camel2enthought*) to perform name mangling so one can change the VTK method name to an enthought style name. The module also provides the function *deref_vtk*, that lets one obtain the underlying VTK object from the tvtk object.

2.3 Code Generation

All the tvtk classes are dynamically generated. *code_gen.py* is the main driver module that is responsible for this. Code generation works involves the following steps.

- Listing and sorting all the VTK classes into a class hierarchy. This is done by *class_tree.py*. *ClassTree* basically organizes all the VTK classes into a tree structure. The tree is organized into "levels". The classes with no bases are at the lowest level (the root) and the higher levels represent classes that have one or more bases. Each class in the tree is represented as a *Node*. Each node has references to its children and parents. Thus one can walk the entire class hierarchy given a single node. This is fairly generic code and will work with any class hierarchy. Its main function is to allow us to generate the classes in the right order and also allows us to easily find the parents and children of a class in a class hierarchy. More details are in the file *class_tree.py*. UTSL.

- Parsing the methods of each VTK class and categorizing them. This step also involves finding the default values of the state of the VTK objects, the valid range of values etc. This entire functionality is provided by *vtk_parser.py*. VTK parser module is completely self documenting and the public interface is fairly simple. The *VTKMethodParser* by default uses an instance of the *ClassTree* class to do some of its work (the use of the *ClassTree* can be turned off though).
- Formatting the output code correctly (indentation). This is handled by the *indenter.Indent* class. The *Indent* class lets us format a code-string suitably so that the code is formatted at the current indentation level.
- Massaging VTK documentation suitably. This is done by *indenter.VTKDocMassager*.
- Generating the tvtk code. Done mainly by *code_gen.py*.
- Generating a ZIP file of the code, also done by *code_gen.py*.

The *code_gen* module defines a *TVTKGenerator* class that drives the code generation. The module uses the *wrapper_gen* module to generate the wrapper classes.

The *wrapper_gen* defines a *WrapperGenerator* class. An instance of this class creates an instance of the *VTKMethodParser* (and uses its internal *ClassTree* instance). *TVTKGenerator* uses this class tree.

For each class in the VTK hierarchy (starting from classes at the root of the tree), the class is parsed, and a suitable tvtk wrapper class (with the name of the class and file name of the module suitably modified) is generated using the functionality provided by *WrapperGenerator*. A separate *tvtk_helper* module (described in the The magical tvtk instance section) is also simultaneously updated with the class for which the wrapper is generated. All of these classes are dumped into a temporary directory. These classes are then put into a ZIP file.

The *tvtk_helper.py* code is generated by the *HelperGenerator* class. The other tvtk classes are generated by the *WrapperGenerator* class. 'WrapperGenerator' also makes use of the *SpecialGenerator* to write special code for some of the VTK classes.

TVTKGenerator uses the other code generators and drives the entire code generation process, builds a ZIP file and optionally cleans up the temporary directory where the wrapper code was written.

Please run the following:

```
$ python code_gen.py -h
```

To see a list of options to the code generator. Some of the options are extremely useful during development. For example do this to generate the code for a few classes alone:

```
$ python code_gen.py -n -z -o /tmp/tvtk_tmp vtkCollection \
  vtkProperty vtkConeSource
```

The *-n* option ensures that '/tmp/tvtk_tmp' is not removed. *-z* inhibits ZIP file generation.

- *Search Page*