3D visualization with TVTK and Mayavi

Prabhu Ramachandran
Gaël Varoquaux

Department of Aerospace Engineering
IIT Bombay
and
Enthought Inc.

20, August 2008

Outline

Outline

Introduction

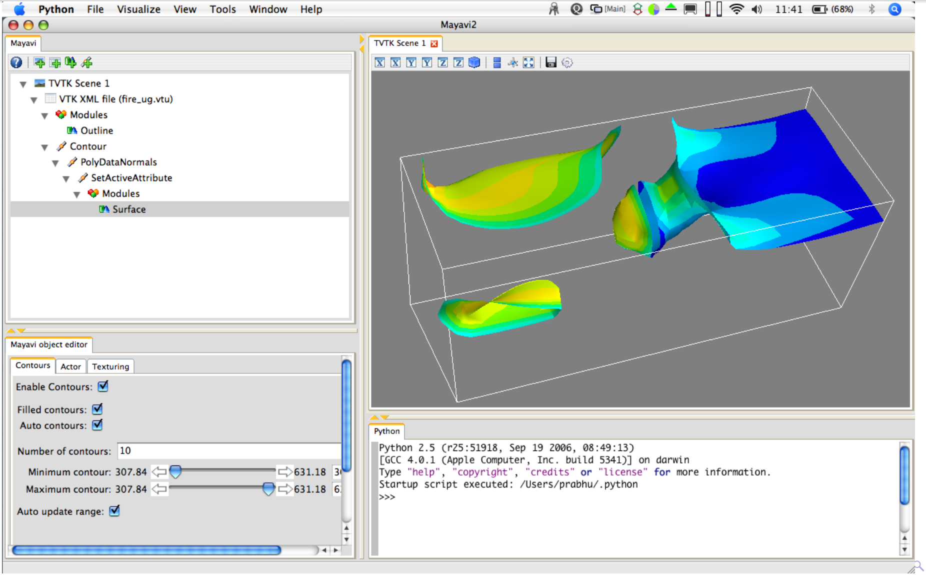
Mayavi

A free, cross-platform, general purpose 3D visualization tool

Features

- Mayavi provides
  - An application for 3D visualization
  - An easy interface: `mlab`
  - Ability to embed mayavi in your objects/views
  - Envisage plugins
  - A more general purpose OO library
  - A numpy/Python friendly API

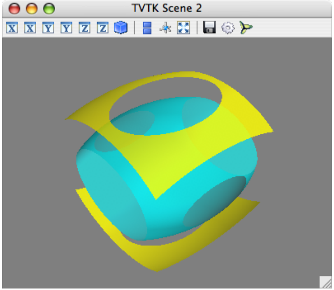
The Mayavi application



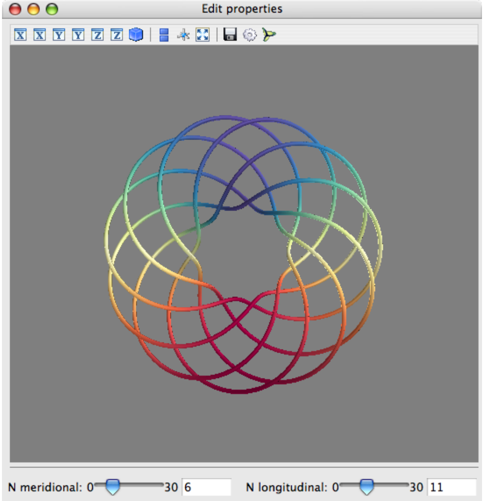
mlab

Example code

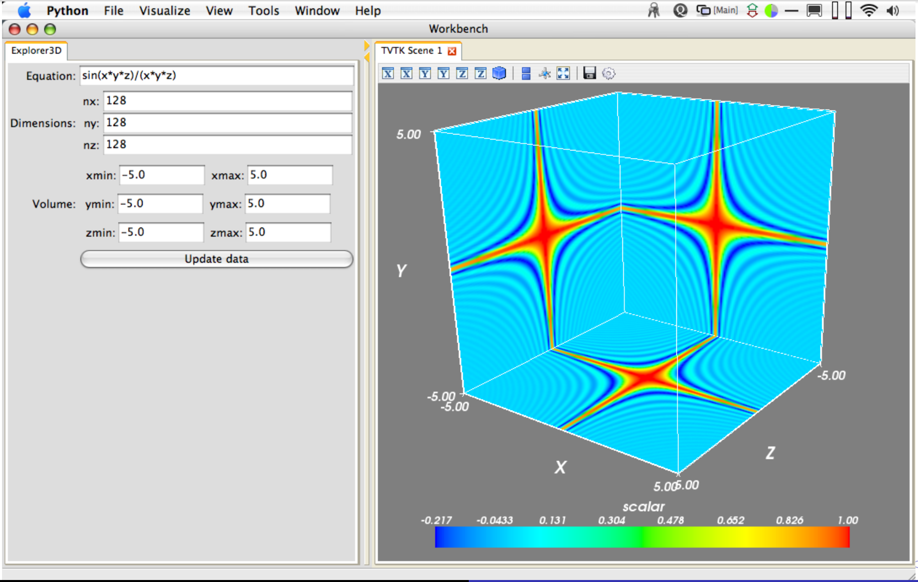
```
dims = [64, 64, 64]
xmin, xmax, ymin, ymax, zmin, zmax = \
                        [-5,5,-5,5,-5,5]
x, y, z = numpy.ogrid[xmin:xmax:dims[0]*1j,
                      ymin:ymax:dims[1]*1j,
                      zmin:zmax:dims[2]*1j]
x, y, z = [t.astype('f') for t in (x, y, z)]

scalars = x*x*0.5 + y*y + z*z*2.0
# Contour the data.
from enthought.mayavi import mlab
mlab.contour3d(scalars, contours=4,
               transparent=True)
```



mlab in your dialogs



Mayavi in your envisage apps



Outline

1. Introduction
   - Overview
   - Installation
2. mlab
   - Introduction
   - Animating data
3. VTK and TVTK
   - Introduction to VTK and TVTK
   - TVTK datasets from numpy arrays
4. Advanced features
   - Embedding mayavi/mlab in traits UIs
   - The mayavi library

Installation

- Requirements:
  - numpy
  - wxPython-2.8.x or PyQt 4.x
  - VTK-5.x
  - IPython
  - Ideally ETS-3.0.0 if not ETS-2.8.0 will work
- Easiest option: install latest EPD
- Debian packages??
- `easy_install Mayavi[app]`
- Get help on `enthought-dev@mail.enthought.com`

Outline

Outline

- Exposure to major mlab functionality
- Introduction to the visualization pipeline
- Advanced features
  - Datasets in mlab
  - Filtering data
  - Animating data
- Demo of the mayavi2 app via mlab

Outline

Animating data

- Animate data without recreating the pipeline
- Use the `mlab_source` attribute to modify data

Example code

```
x, y = numpy.mgrid[0:3:1,0:3:1]
s = mlab.surf(x, y, numpy.asarray(x*0.1, 'd'))

# Animate the data.
ms = s.mlab_source # Get the source
for i in range(10):
    # Modify the 'scalars'
    ms.scalars = numpy.asarray(x*0.1*(i+1), 'd')
```

Animating data

- Typical `mlab_source` traits
  - `x, y, z`: x, y, z positions
  - `points`: generated from x, y, z
  - `scalars`: scalar data
  - `u, v, w`: components of vector data
  - `vectors`: vector data
- Important methods:
  - `set`: set multiple traits efficiently
  - `reset`: use when the arrays change shape; slow
  - `update`: call when you change points/scalars/vectors in-place
- Check out the examples: `mlab.test_*_anim`

Exercise

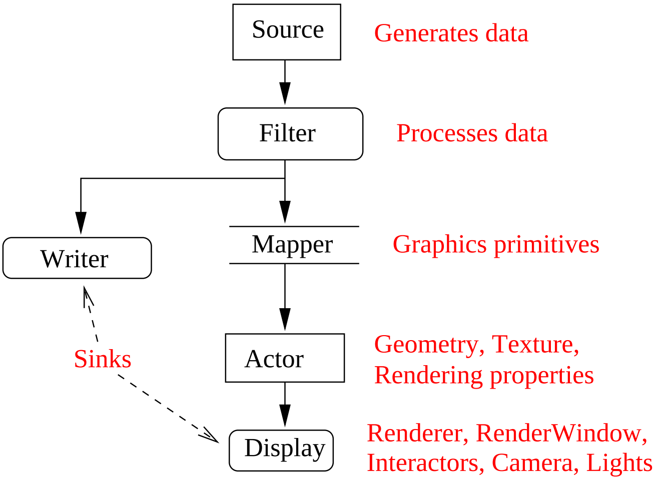
- Show iso-contours of the function $\sin(xyz)/xyz$
- x, y, z in region $[-5, 5]$ with 32 points along each axis

Outline

Introduction

- Open source, BSD style license
- High level library
- 3D graphics, imaging and visualization
- Core implemented in C++ for speed
- Uses OpenGL for rendering
- Wrappers for Python, Tcl and Java
- Cross platform: *nix, Windows, and Mac OSX
- Around 40 developers worldwide
- Very powerful with lots of features/functionality: > 900 classes
- Pipeline architecture
- Not trivial to learn (VTK book helps)
- Reasonable learning curve

VTK / TVTK pipeline



Issues with VTK

- API is not Pythonic for complex scripts
- Native array interface
- Using NumPy arrays with VTK: non-trivial and inelegant
- Native iterator interface
- Can't be pickled
- GUI editors need to be "hand-made" ($>$ 800 classes!)

TVTK

- "Traitified" and Pythonic wrapper atop VTK
- Elementary pickle support
- `Get/SetAttribute()` replaced with an `attribute` trait
- Handles numpy arrays/Python lists transparently
- Utility modules: pipeline browser, `ivtk`, `mlab`
- Envisage plugins for `tvtk` scene and pipeline browser

The differences

VTK	TVTK
`import vtk`	`from enthought.tvtk.api import tvtk`
`vtk.vtkConeSource`	`tvtk.ConeSource`
no constructor args	traits set on creation
`cone.GetHeight()`	`cone.height`
`cone.SetRepresentation()`	`cone.representation='w'`

- `vtk3DWidget` → `ThreeDWidget`
- Method names: consistent with ETS (`lower_case_with_underscores`)
- VTK class properties (Set/Get pairs or Getters): traits

Array example

Any method accepting `DataArray`, `Points`, `IdList` or `CellArray` instances can be passed a numpy array or a Python list!

```
>>> from enthought.tvtk.api import tvtk
>>> from numpy import array
>>> points = array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], 'f')
>>> triangles = array([[0,1,3], [0,3,2], [1,2,3], [0,2,1]])
>>> mesh = tvtk.PolyData()
>>> mesh.points = points
>>> mesh.polys = triangles
>>> temperature = array([10, 20 ,20, 30], 'f')
>>> mesh.point_data.scalars = temperature
>>> import operator # Array's are Pythonic.
>>> reduce(operator.add, mesh.point_data.scalars, 0.0)
80.0
>>> pts = tvtk.Points() # Demo of from_array/to_array
>>> pts.from_array(points)
>>> print pts.to_array()
```

More details

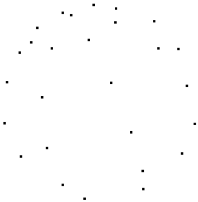
- TVTK is fully documented
- More details here: `http://code.enthought.com/projects/mayavi/documentation.php`

Outline

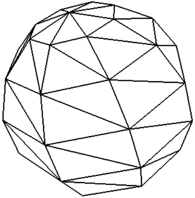
Datasets: Why the fuss?

- 2D line plots are easy
- Visualizing 3D data: requires a little more information
- Need to specify a topology (i.e. how are the points connected)

An example of the difficulty



Points (0D)



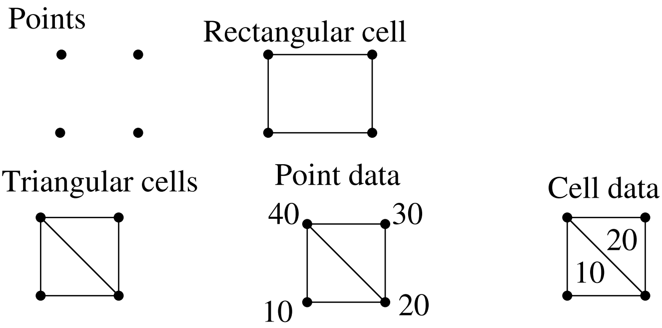
Wireframe (1D)

Surface (2D)

Interior of sphere: Volume (3D)

The general idea

1. Specify the points of the space
2. Specify the connectivity between the points (topology)
3. Connectivity specify “cells” partitioning the space
4. Specify “attribute” data at the points or cells



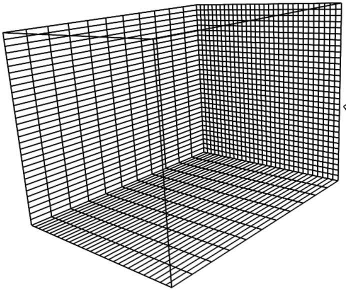
Types of datasets

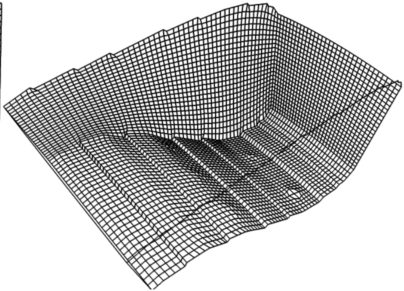
- Implicit topology (structured):
  - Image data (structured points): constant spacing, orthogonal
  - Rectilinear grids: non-uniform spacing, orthogonal
  - Structured grids: explicit points
- Explicit topology (unstructured):
  - Polygonal data (surfaces)
  - Unstructured grids

Implicit versus explicit topology

Structured grids

- Implicit topology associated with points:
  - The *X* co-ordinate increases first, *Y* next and *Z* last
- Easiest example: a rectangular mesh
- Non-rectangular mesh certainly possible (structured grid)

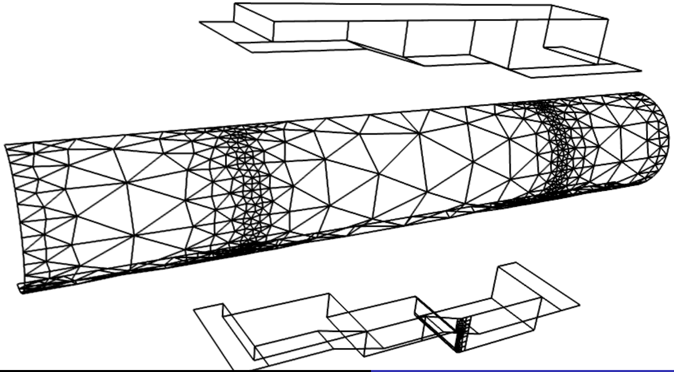




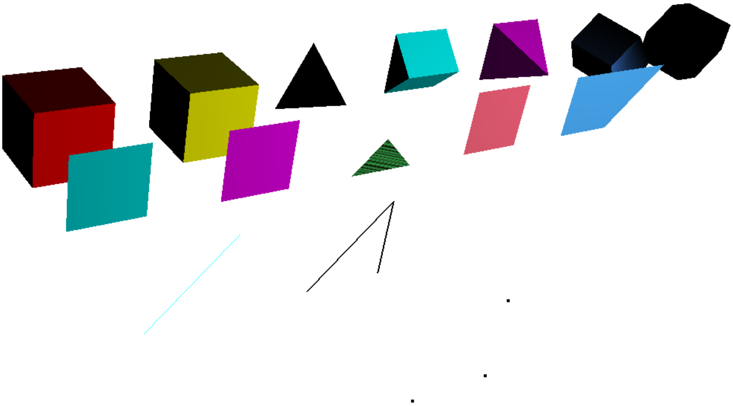
Implicit versus explicit topology

Unstructured grids

- Explicit topology specification
- Specified via connectivity lists
- Different number of neighbors, different types of cells



Different types of cells



Dataset attributes

- Associated with each point/cell one may specify an attribute
  - Scalars
  - Vectors
  - Tensors
- Cell and point data attributes
- Multiple attributes per dataset

Overview

- Creating datasets with TVTK and Numpy: by example
- Very handy when working with Numpy
- No need to create VTK data files
- Can visualize them easily with mlab/mayavi
- See `examples/mayavi/datasets.py`

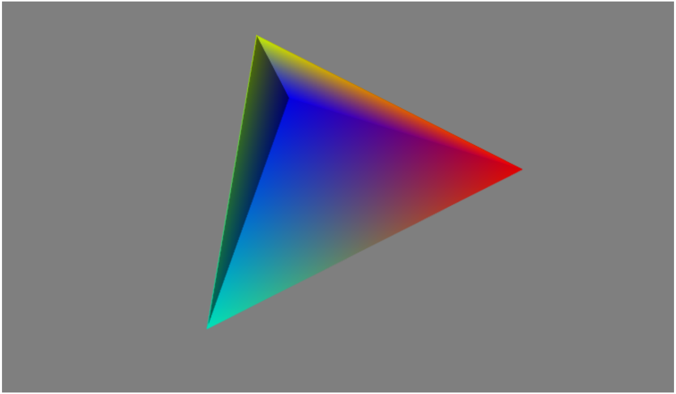
PolyData

- Create the dataset: `tvtk.PolyData()`
- Set the points: `points`
- Set the connectivity: `polys`
- Set the point/cell attributes (in same order)

PolyData

```
from enthought.tvtk.api import tvtk
# The points in 3D.
points = array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], 'f')
# Connectivity via indices to the points.
triangles = array([[0,1,3], [0,3,2], [1,2,3], [0,2,1]])
# Creating the data object.
mesh = tvtk.PolyData()
mesh.points = points # the points
mesh.polys = triangles # triangles for connectivity.
# For lines/verts use: mesh.lines = lines; mesh.verts = vertices
# Now create some point data.
temperature = array([10, 20 ,20, 30], 'f')
mesh.point_data.scalars = temperature
mesh.point_data.scalars.name = 'temperature'
# Some vectors.
velocity = array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], 'f')
mesh.point_data.vectors = velocity
mesh.point_data.vectors.name = 'velocity'
# Thats it!
```

PolyData



ImageData

- Orthogonal cube of data: uniform spacing
- Create the dataset: `tvtk.ImageData()`
- Implicit points and topology:
  - `origin`: origin of region
  - `spacing`: spacing of points
  - `dimensions`: think size of array
- Set the point/cell attributes (in same order)
- Implicit topology implies that the data must be ordered
- The *X* co-ordinate increases first, *Y* next and *Z* last

Image Data/Structured Points: 2D

```
# The scalar values.
from numpy import arange, sqrt
from scipy import special
x = (arange(50.0)-25)/2.0
y = (arange(50.0)-25)/2.0
r = sqrt(x[:,None]**2+y**2)
z = 5.0*special.j0(r)  # Bessel function of order 0
# -------------------------------------------------------
# Can't specify explicit points, the points are implicit.
# The volume is specified using an origin, spacing and dimensions
img = tvtk.ImageData(origin=(-12.5,-12.5,0),
                     spacing=(0.5,0.5,1),
                     dimensions=(50,50,1))
# Transpose the array data due to VTK's implicit ordering. VTK
# assumes an implicit ordering of the points: X co-ordinate
# increases first, Y next and Z last. We flatten it so the
# number of components is 1.
img.point_data.scalars = z.T.flatten()
img.point_data.scalars.name = 'scalar'
```

ImageData 2D

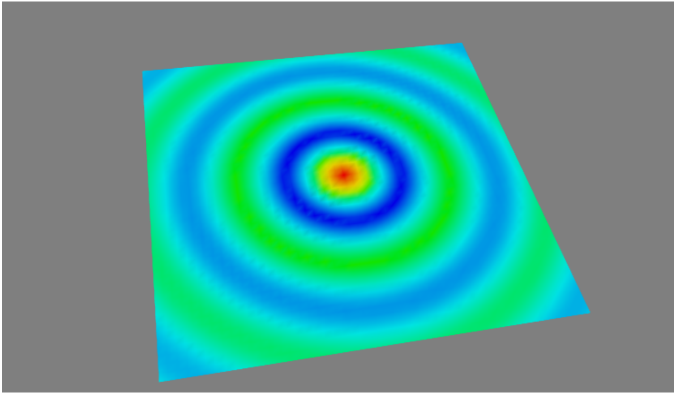
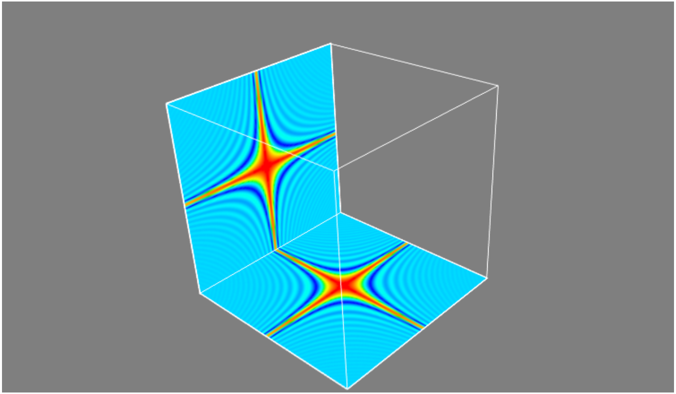


Image data: 3D

```
from numpy import array, ogrid, sin, ravel
dims = array((128, 128, 128))
vol = array((-5., 5, -5, 5, -5, 5))
origin = vol[::2]
spacing = (vol[1::2] - origin)/(dims -1)
xmin, xmax, ymin, ymax, zmin, zmax = vol
x, y, z = ogrid[xmin:xmax:dims[0]*1j,ymin:ymax:dims[1]*1j,
                zmin:zmax:dims[2]*1j]
x, y, z = [t.astype('f') for t in (x, y, z)]
scalars = sin(x*y*z)/(x*y*z)
# ----------------------------------------------------------------
img = tvtk.ImageData(origin=origin, spacing=spacing,
                     dimensions=dims)
# The copy makes the data contiguous and the transpose
# makes it suitable for display via tvtk.
s = scalars.transpose().copy()
img.point_data.scalars = ravel(s)
img.point_data.scalars.name = 'scalars'
```

ImageData 3D



RectilinearGrid

- Orthogonal cube of data: non-uniform spacing
- Create the dataset: `tvtk.RectilinearGrid()`
- Explicit points: `x_coordinates`, `y_coordinates`, `z_coordinates`
- Implicit topology: X first, Y next and Z last
- Set the point/cell attributes (in same order)

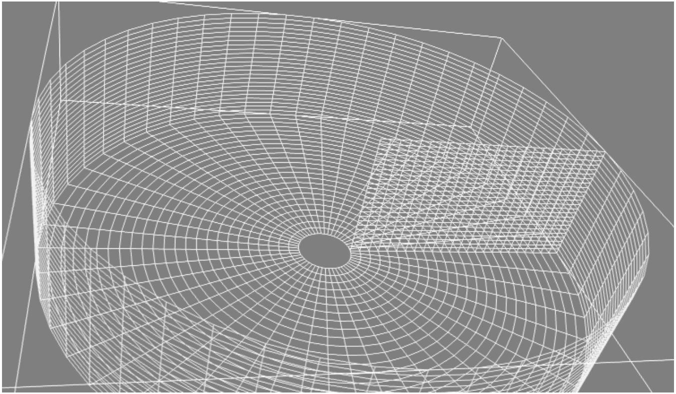
StructuredGrid

- Create the dataset: `tvtk.StructuredGrid()`
- Explicit points: `points`
- Implicit topology: X first, Y next and Z last
- Set the point/cell attributes (in same order)

Structured Grid

```
r = numpy.linspace(1, 10, 25)
theta = numpy.linspace(0, 2*numpy.pi, 51)
z = numpy.linspace(0, 5, 25)
# Crreate an annulus.
x_plane = (cos(theta)*r[:,None]).ravel()
y_plane = (sin(theta)*r[:,None]).ravel()
pts = empty([len(x_plane)*len(height),3])
for i, z_val in enumerate(z):
    start = i*len(x_plane)
    plane_points = pts[start:start+len(x_plane)]
    plane_points[:,0] = x_plane
    plane_points[:,1] = y_plane
    plane_points[:,2] = z_val
sgrid = tvtk.StructuredGrid(dimensions=(51, 25, 25))
sgrid.points = pts
s = numpy.sqrt(pts[:,0]**2 + pts[:,1]**2 + pts[:,2]**2)
sgrid.point_data.scalars = numpy.ravel(s.copy())
sgrid.point_data.scalars.name = 'scalars'
```

StructuredGrid



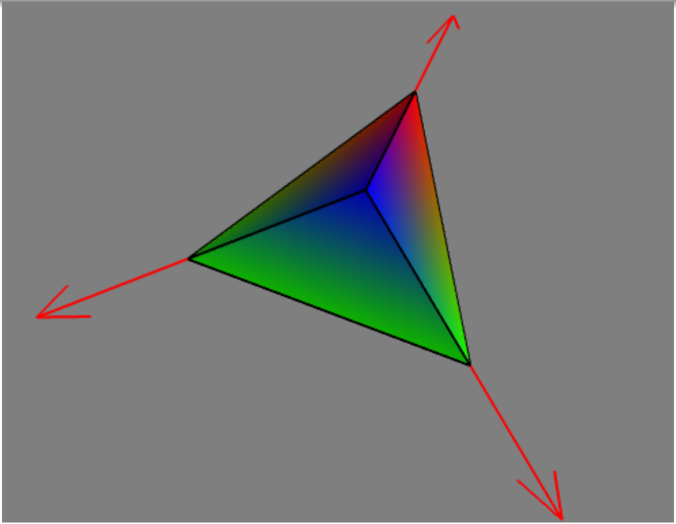
UnstructuredGrid

- Create the dataset: `tvtk.UnstructuredGrid()`
- Explicit points: `points`
- Explicit topology:
  - Specify cell connectivity
  - Specify the cell types to use
  - `set_cells(cell_type, cell_connectivity)`
  - `set_cells(cell_types, offsets, connect)`
- Set the point/cell attributes (in same order)
- See `examples/mayavi/unstructured_grid.py`

Unstructured Grid

```
from numpy import array
points = array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], 'f')
tets = array([[0, 1, 2, 3]])
tet_type = tvtk.Tetra().cell_type # VTK_TETRA == 10
#-----------------------------------------------------------
ug = tvtk.UnstructuredGrid()
ug.points = points
# This sets up the cells.
ug.set_cells(tet_type, tets)
# Attribute data.
temperature = array([10, 20 ,20, 30], 'f')
ug.point_data.scalars = temperature
ug.point_data.scalars.name = 'temperature'
# Some vectors.
velocity = array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], 'f')
ug.point_data.vectors = velocity
ug.point_data.vectors.name = 'velocity'
```

UnstructuredGrid



Outline

1. Introduction
   - Overview
   - Installation
2. `mlab`
   - Introduction
   - Animating data
3. VTK and TVTK
   - Introduction to VTK and TVTK
   - TVTK datasets from numpy arrays
4. Advanced features
   - Embedding mayavi/mlab in traits UIs
   - The mayavi library

General approach

- Create class deriving from `HasTraits` ...
- Add an Instance of `MlabSceneModel` trait: lets call it `scene`
- mlab is available as `self.scene.mlab`
- Use a `SceneEditor` as an editor for the `MlabSceneModel` trait
- Do the needful in trait handlers
- Thats it!

An example

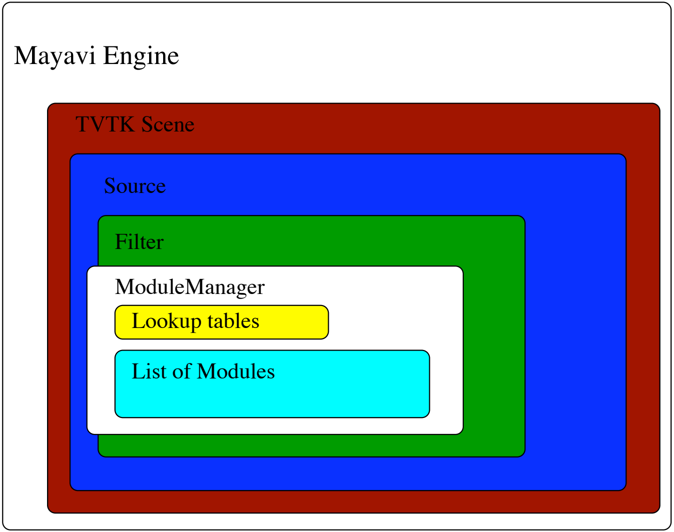
```
from enthought.tvtk.pyface.scene_editor import SceneEditor
from enthought.mayavi.tools.mlab_scene_model import MlabSceneMode
from enthought.mayavi.core.ui.mayavi_scene import MayaviScene
class ActorViewer(HasTraits):
    scene = Instance(MlabSceneModel, ())
    view = View(Item(name='scene',
                     editor=SceneEditor(scene_class=MayaviScene),
                     show_label=False, resizable=True, width=500,
                     height=500), resizable=True)
    def __init__(self, **traits):
        HasTraits.__init__(self, **traits)
        self.generate_data()
    def generate_data(self):
        X, Y = mgrid[-2:2:100j, -2:2:100j]
        R = 10*sqrt(X**2 + Y**2)
        Z = sin(R)/R
        self.scene.mlab.surf(X, Y, Z, colormap='gist_earth')

a = ActorViewer(); a.configure_traits()
```

Exercise

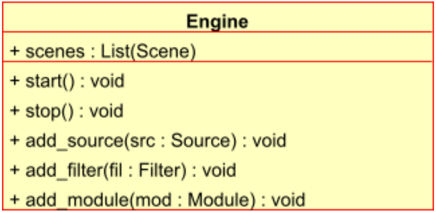
- Take the previous example $\sin(xyz)/xyz$ and allow a user to specify a function that is evaluated
- Hints:
  - Use an `Expression` trait
  - Eval the expression given in a namespace with your x, y, z arrays along with numpy/scipy.
  - Use `mlab_source` to update the scalars

Outline

The big picture

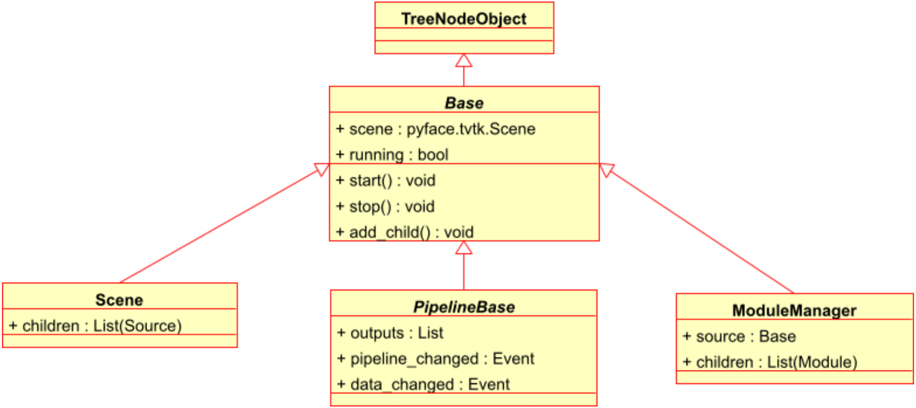


Containership relationship

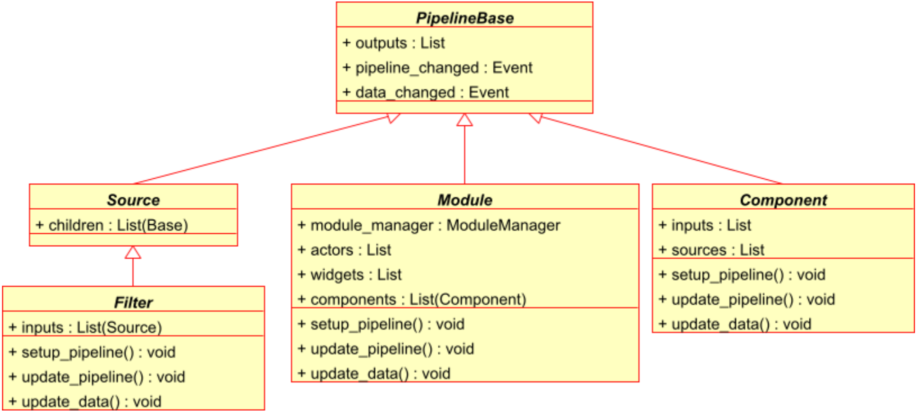


- `Engine` contains: list of `Scene`
- `Scene` contains: list of `Source`
- `Source` contains: list of `Filter` and/or `ModuleManager`
- `ModuleManager` contains: list of `Module`
- `Module` contains: list of `Component`

Class hierarchy



Class hierarchy



Interactively scripting Mayavi2

- Drag and drop
- The `mayavi` instance

```
>>> mayavi.new_scene() # Create a new scene
>>> mayavi.save_visualization('foo.mv2')
```

- `mayavi.engine`:

```
>>> e = mayavi.engine # Get the MayaVi engine.
>>> e.scenes[0] # first scene in mayavi.
>>> e.scenes[0].children[0]
>>> # first scene's first source (vtkfile)
```

Scripting ...

- `mayavi`: instance of `enthought.mayavi.script.Script`
- Traits: `application`, `engine`
- Methods (act on current object/scene):
  - `open(fname)`
  - `new_scene()`
  - `add_source(source)`
  - `add_filter(filter)`
  - `add_module(m2_module)`
  - `save/load_visualization(fname)`

Stand alone scripts

- Several approaches to doing this
- Recommended way:
  - Save simple interactive script to `script.py` file
  - Run it like `mayavi2 -x script.py`
  - Can use built-in editor
  - Advantages: easy to write, can edit from mayavi and rerun
  - Disadvantages: not a stand-alone Python script
- `@standalone` decorator to make it a standalone script:

```
from enthought.mayavi.scripts.mayavi2
import standalone
```

Creating a simple new filter

- New filter specs: Take an iso-surface + optionally compute normals
- Easy to do if we can reuse code, use the `Optional`, `Collection` filters

```
from enthought.mayavi.filters.api import (PolyDataNormals,
                              Contour, Optional, Collection)
c = Contour()
# `name` is used for the notebook tabs.
n = PolyDataNormals(name='Normals')
o = Optional(filter=n, label_text='Compute normals')
filter = Collection(filters=[c, o], name='MyIsoSurface')
# Now filter may be added to mayavi
mayavi.add_filter(filter)
```

Creating a simple new module

- Using the `GenericModule` we can easily create a custom module out of existing ones.
- Here we recreate the `ScalarCutPlane` module

A ScalarCutPlaneModule

```
from enthought.mayavi.filters.api import Optional, WarpScalar, ...
from enthought.mayavi.components.contour import Contour
# import Contour, Actor from components.
from enthought.mayavi.modules.api import GenericModule
cp = CutPlane()
w = WarpScalar()
warper = Optional(filter=w, label_text='Enable warping',
                  enabled=False)
c = Contour()
ctr = Optional(filter=c, label_text='Enable contours',
               enabled=False)
p = PolyDataNormals(name='Normals')
normals = Optional(filter=p, label_text='Compute normals',
                   enabled=False)
a = Actor()
components = [cp, warper, ctr, normals, a]
m = GenericModule(name='ScalarCutPlane', components=components,
                  contour=c, actor=a)
```

Additional features

- Offscreen rendering
- Animations
- Envisage plugins
- Customizing mayavi

Offscreen rendering

- Offscreen rendering: `mayavi2 -x script.py -o`
- Requires a sufficiently recent VTK release (5.2 or CVS)
- No UI is shown
- May ignore any window shown
- Normal mayavi and mlab scripts ought to work

Animation scripts

```
from os.path import join, exists
from os import mkdir

def make_movie(directory='/tmp/movie', n_step=36):
    if not exists(directory):
        mkdir(directory)
    scene = mayavi.engine.current_scene.scene
    camera = scene.camera
    da = 360.0/n_step
    for i in range(n_step):
        camera.azimuth(da)
        scene.render()
        scene.save(join(directory, 'anim%02d.png'%i))

if __name__ == '__main__':
    make_movie()
```

Run using `mayavi2 -x script.py`

Envisage plugins

- Allow us to build extensible apps
- Mayavi provides two plugins:
  - MayaviPlugin: contributes a window level `Engine` and `Script` service offer and preferences
  - MayaviUIPlugin: UI related contributions: menus, tree view, object editor, perspectives

Customization

- Global: `site_mayavi.py`; placed on `sys.path`
- Local: `~/.mayavi2/user_mayavi.py`: this directory is placed in `sys.path`
- Two things can be done in this file:
  - Register new sources/modules/filters with mayavi's registry: `enthought.mayavi.core.registry:registry`
  - `get_plugins()` returns a list of envisage plugins to add
- See `examples/mayavi/user_mayavi.py`