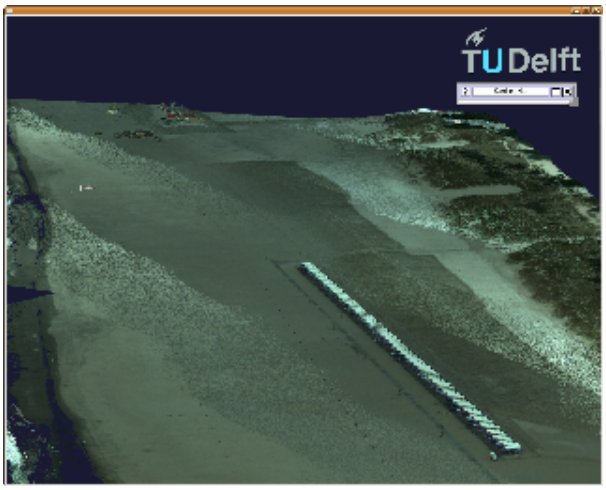
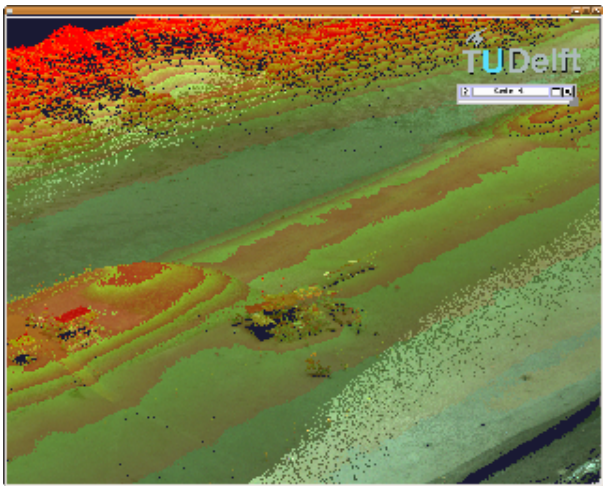
A stereoscopic view on the coast

Introduction

This example shows how to visualize data of the Rhine ROFI. This example was demonstrated at the NCK days 2010.
This was the presentation as presented on the left beamer:





For the example that was shown in stereoscopy you need:

Data

The data will be made available through opendap.deltares.nl. You need the rhinerofi.nc file. It is not yet available.

Tools

- python
- mayavi2
- netcdf4-python or pupynere (this one requires a lot of memory 2GB at least)

Hardware

- Stereoscopic equipment (polarized, lcd shutter or anaglyphs). We're asssuming anaglyph glasses here. You can get those at http://www.3dstore.nl . At the NCK days we used 2 beamers with polarized filters, a screen with silver coating to reflect the polarized light and polarized (linear) glasses.

The script used:

```
#!/usr/bin/env python

# This is an example script to generate a stereoscopic visualization
# We'll import some libraries
import random
import os

from enthought import mayavi # the application
from enthought.mayavi import mlab # a procedural facade to the library which uses the same names as matlab
from enthought.mayavi.modules.api import Outline, GridPlane
from enthought.mayavi.sources.vtk_data_source import VTKDataSource
from enthought.tvtk.api import tvtk


import numpy
from numpy import mgrid, empty, sin, pi, array, meshgrid, arange, prod
```

```
# some path to your local netcdf file
path = 'rhinerofi.nc'
path = '/Users/fedorbaart/Documents/checkouts/baart_f/programs/matlab/gerben/gerbenvankees.nc'

# try and import the dataset, prefer netcdf4, you might want to use pydap also here if you access a dap server.
ds = None
try:
    import netCDF4
    ds = netCDF4.Dataset(path)
except:
    import pupynere
    # use mmap = False if it doesn't fit ...
    ds = pupynere.NetCDFFile(path, mmap=True)

# read the variables (these are just links to the objects for now)
sal = ds.variables['sal'] # salinity
x = ds.variables['x'] # x location (projected)
y = ds.variables['y'] # y location
z = ds.variables['z'] # sigma level
zwl = ds.variables['zwl']  # sea surface elevation
u = ds.variables['u'] # velocity in u direction
v = ds.variables['v'] # v direction
w = ds.variables['w'] # w direction

# we'll slice some data for better performance and less memory.
# If the grid is not changing (as it does with a sigma layer) this can be done with an extractgrid in the
pipeline
# But now we have to update the whole grid and this can be time consuming, so we make use of netcdf slicing to
reduce the
# amount of data we read in
SLICE = True
if SLICE:
    mslice = slice(0,165,3) # from 0 to 165, each 3rd value
    nslice = slice(40,150,3)
    kslice = slice(0,16,2)
else:
    mslice = slice(0,-1,1) # from 0 to end each value (no slice)
    nslice = slice(0,-1,1)
    kslice = slice(0,-1,1)
# read the data into matrices, raise the seawater level a bit (this should be done in the pipeline)
X, Y, Z = x[mslice,nslice], y[mslice,nslice], zwl[:, mslice, nslice] + 5
# we don't want nan's in our coordinates. You could use the mask cell function to hide these cells
if not SLICE:
    X[:,-3:] = [-1750., -1250.,  -750.]

# set the data to a value
if not SLICE:
    Y[:,-3:] = numpy.c_[Y[:,-4], Y[:,-4], Y[:,-4]] # this can be done simpler

# strip of a vector because here the sigma layer is constant over time.
z = z[1,kslice]

# Now we'll create an n by 3 vector of xyz data
# create an empty matrix and fill in:
pts = empty((prod(X.shape + (len(z),)), 3), dtype=float)
# x
pts[:,0] = numpy.tile(X.T.ravel(), z.shape[0])
# y
pts[:,1] = numpy.tile(Y.T.ravel(), z.shape[0])
# z (also multiply by 3000, this should be done in the pipeline)
pts[:,2] = numpy.repeat(z, prod(Z[1,:,:].shape)) * numpy.tile(Z[1,:,:].T.ravel(), z.shape[0]) * 3000

# Now we create the main data object that stores coordinates together with scalar and vector values
sg = tvtk.StructuredGrid(dimensions=X.shape + (len(z),))
# set the coordinates
sg.points = pts
# set the scalars to the salinity (the convention in netcdf is to use zyx for performance reasons)
sg.point_data.scalars= sal[1,kslice, mslice,nslice].swapaxes(1,2).ravel()
sg.point_data.scalars.name = 'scalars'
```

```
# compute the vectors and multiply the w component with the same factor as used for the points
vectors = numpy.c_[u[1,kslice,mslice,nslice].swapaxes(1,2).ravel(),
                   v[1,kslice,mslice,nslice].swapaxes(1,2).ravel(),
                   w[1,kslice,mslice,nslice].swapaxes(1,2).ravel()*3000]
sg.point_data.vectors = vectors
sg.point_data.vectors.name = 'vectors'


# We have the data, now we're creating a pipeline for visualization
# start with the source
d = mlab.pipeline.add_dataset(sg)

# we'll extract the grid, so we can tweak the performance. This is only used of slice is not active. If it is
# we slice at the source, not in the pipeline
eg = mlab.pipeline.extract_grid(sg)
# We'll create an extra extract grid to extract the top layer for the water surface
egsurf = mlab.pipeline.extract_grid(sg)
if not SLICE:
    # slice
    eg.set(x_max=165, x_ratio=3, y_min=40, y_max=150, y_ratio=3, z_ratio=3)
    # slice and only extract the top layer
    egsurf.set(x_max=165, x_ratio=3, y_min=40, y_max=150, z_min = 0, z_max=0, y_ratio=3, z_ratio=3)
else:
    # extract the top layer
    egsurf.set(z_min = 0, z_max=0)

# create the water surface using a green to blue color (depending on salinity) and make it a bit transparent
sfsurf = mlab.pipeline.surface(egsurf, opacity=0.3, colormap='GnBu', representation='wireframe' )

# create the salinity contours
sf = mlab.pipeline.iso_surface(eg, opacity=0.5)
sf.contour.auto_contours = False
sf.contour.auto_update_range = False
# show the most interesting contours
sf.contour.contours = [20, 30.0, 31.7, 32.5]
# show every 20th vector, make it a factor 5000 longer and show as 3D arrows
vf = mlab.pipeline.vectors(eg, mask_points=20, scale_factor=5000, mode='arrow')

# create a streamline
sl = mlab.pipeline.streamline(eg, linetype='tube', opacity=0.8)
# make it stream longer
sl.stream_tracer.maximum_propagation = 50000.0
# make the tubes thicker
sl.tube_filter.radius = 200.0
# set the size of the sphere
sl.seed.widget.radius = 5608.0572051878689
# make the tubes a bit rounder
sl.tube_filter.number_of_sides = 5
# set the location of the sphere
sl.seed.widget.center = array([-12359.64662114,  51292.88341463,  25157.53124445])
sl.actor.visible = True
# hide it for now...
sl.visible = False

# create a set of line streamlines
sp = mlab.pipeline.streamline(eg, opacity=0.8, seedtype='plane')
# make it stream north/southward
sp.seed.widget.normal_to_y_axis = True
sp.stream_tracer.maximum_propagation = 50000.0
sp.actor.visible = True
# hide it for now
sp.visible = False

# show the outline of the grid
ol = mlab.pipeline.outline(eg)

# show the resolution of the grid in all dimensions
gx = GridPlane()
gx.grid_plane.axis = 'x'
eg.add_module(gx)
gy = GridPlane()
```

```
gy.grid_plane.axis = 'y'
eg.add_module(gy)
gz = GridPlane()
gz.grid_plane.axis = 'z'
eg.add_module(gz)

# set some colors and a title
f = mlab.gcf()
f.scene.foreground=(0,0,0)
f.scene.background=(1,1,1)
title = mlab.title('North Sea salinity example')
title.width=0.3

# The image is now done. Now we'll add some functions for interaction

def timestep(i):
    """set the data from timestep(i)"""
    # set a new title
    title.text=str(i)
    title.width=0.03
    title.x_position = 0.1
    title.y_position = 0.9
    # update the grid with new values
    sg.point_data.scalars = sal[i,kslice,mslice,nslice].swapaxes(1,2).ravel()
    sg.point_data.scalars.name = 'scalars'
    # and new locations (because pts is referenced from sg.points, we can just update the source)
    pts[:,2] = numpy.repeat(z, prod(Z[i,:,:].shape)) * numpy.tile(Z[i,:,:].T.ravel(), z.shape[0]) * 7000
    # and new vectors
    vectors = numpy.c_[u[i,kslice,mslice,nslice].swapaxes(1,2).ravel(), v[i,kslice,mslice,nslice].swapaxes(1,2).
ravel(), w[i,kslice,mslice,nslice].swapaxes(1,2).ravel()*300]
    sg.point_data.vectors = vectors
    sg.point_data.vectors.name = 'vectors'
    # flush the pipe by notifying the object is updated
    sg.modified()

# we could do without this
engine = mlab.get_engine()
# now we'll add a method for an animation
# an animation is just an iterator
@mlab.animate(delay=100)
def anim(i=0):
    scene = engine.scenes[0]
    while True: # keep playing
        # improve performance by temporary disabling rendering
        scene.scene.disable_render = True
        # make the animation start at the beginning
        i+=1
        if i >= 77:
            i=1
        # update to timestep i
        timestep(i)
        # start rendering again
        scene.scene.disable_render = False
        value = (yield i) # allows to rewind the animation (not used)
        if value:
            i = value
# now for some more utility functions (all use global variables which is a bit ugly but since they're used
during presentation
# they also allow for less typing...
# switch grid on and off
def gridon():
    """show computational grid"""
    gx.visible = True
    gy.visible = True
    gz.visible = True
def gridoff():
    """turn off computational grid"""
    gx.visible = False
    gy.visible = False
    gz.visible = False
# switches for streamlines
```

```
def slon():
    """enable streamlines and disable vector fields"""
    sl.visible = True
    sp.visible = True
    sp.seed.visible = False
    vf.visible = False
def sloff():
    """enable vector field and disable streamline"""
    sl.visible = False
    sp.visible = False
    sp.seed.visible = False
    vf.visible = True


# set the camera in the correct angle
scene = engine.scenes[0]
mlab.view(40, 50)

# scale colors a bit
module_manager = vf.module_manager
module_manager.vector_lut_manager.use_default_range = False
module_manager.vector_lut_manager.data_range = array([ 0. ,  0.7])
module_manager.scalar_lut_manager.use_default_range = False
module_manager.scalar_lut_manager.data_range = array([ 31.0 ,  34.5])

#turn on stereo
renderer = engine.scenes[0].scene.render_window
renderer.stereo_type = 'anaglyph' # 'crystal_eyes' for quad buffered stereo
renderer.stereo_render = 1

mlab.show()
```