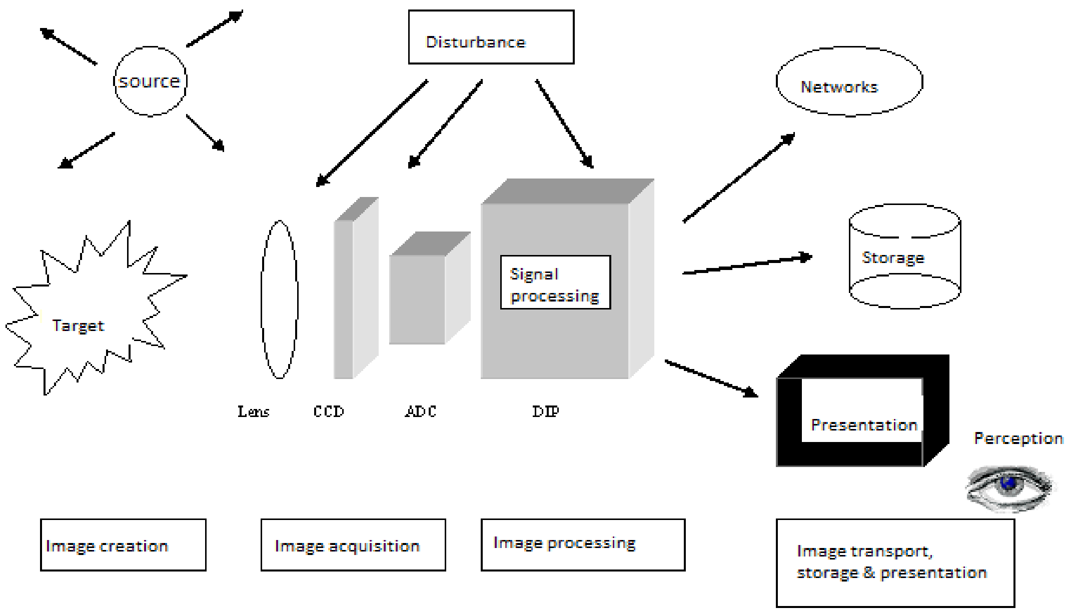
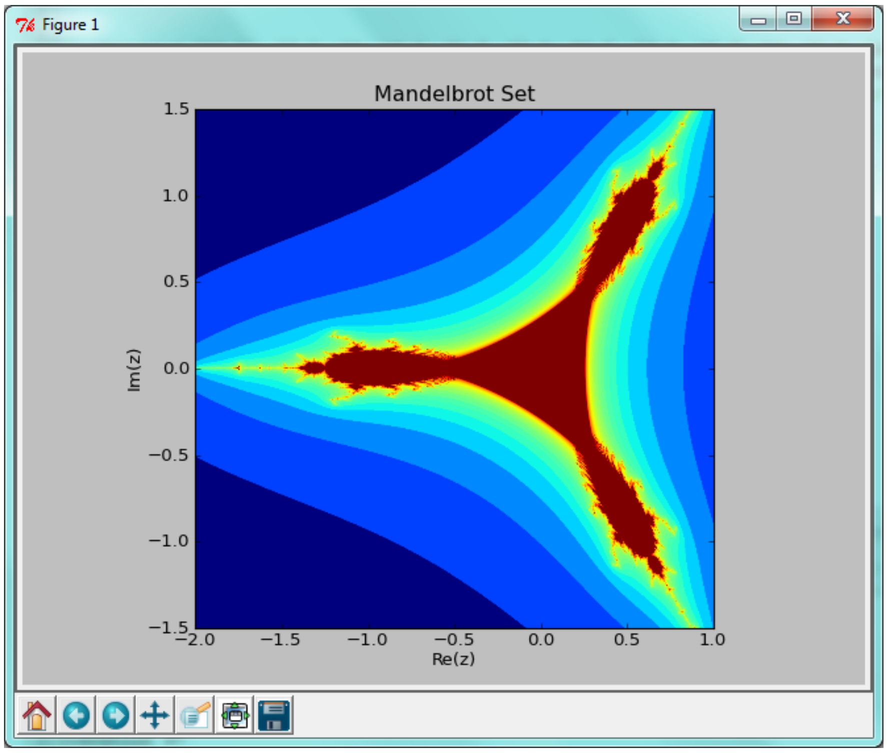
Visualization
Capita Selecta (VCS2)





Program for sept 2016- Jan 2017
Sjaak Verwaaijen
Fontys ICT

Contents

Introduction

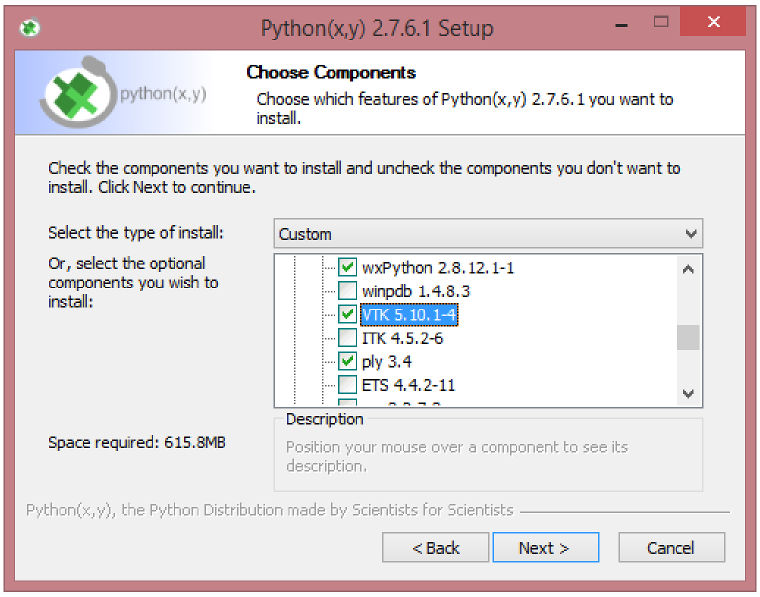
In this self-study module, visualization techniques are discussed and trained via video lessons, and via practical exercises in this book. You can learn to make a 3D visualization in the framework VTK in Python.

Of course if you have difficulties with the assignments to can contact the teacher.

You can use the Scientific-oriented Python Distribution based on Qt and Spyder https://code.google.com/p/pythonxy/

Download http://www.mirrorservice.org/sites/pythonxy.com/Python(x,y)-2.7.6.1.exe

Check VTK 5.10.1-4. We use VTK as a visualization tool (Visualization ToolKit http://www.vtk.org).



Python(x,y) 2.7.6.1 Setup

Choose Components

Choose which features of Python(x,y) 2.7.6.1 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Custom

Or, select the optional components you wish to install:

- [✓] wxPython 2.8.12.1-1
- [ ] winpdb 1.4.8.3
- [✓] VTK 5.10.1-4
- [ ] ITK 4.5.2-6
- [✓] ply 3.4
- [ ] ETS 4.4.2-11

Space required: 615.8MB

Description

Position your mouse over a component to see its description.

Python(x,y), the Python Distribution made by Scientists for Scientists

Note: Visual Studio 2015 also has the ability to use standard Python.

Visualization

We define *visualization* as the communication of information using graphical representations. It is any technique for creating images, diagrams, or animations to communicate a message.

http://www.webdesignerdepot.com/2009/06/50-great-examples-of-data-visualization/

Pictures have been used as a mechanism for communication since before the formalization of written language. A single picture can contain a wealth of information, and can be processed much more quickly than a comparable page of words. Pictures can also be independent of local language, as a graph or a map may be understood by a group of people with no common tongue.

It is an interesting exercise to consider the number and types of data and information visualization that we encounter in our normal activities. Some of these might include:

- a table in a newspaper, representing data being discussed in an article;
- a train and subway map with times used for determining train arrivals and departures;

- a map of the region, to help select a route to a new destination;
- a weather chart showing the movement of a storm front that might influence your weekend activities;
- a graph of stock market activities that might indicate an upswing (or downturn) in the economy;
- a plot comparing the effectiveness of your pain killer to that of the leading brand;
- a 3D reconstruction of your injured knee, as generated from a CT scan;
- an instruction manual for putting together a bicycle, with views specific to each part as it is added;
- a highway sign indicating a curve, merging-of lanes, or an intersection.

Visualization is used on a daily basis in many areas of employment as well, such as:

- the result of a financial and stock market analysis;
- a mechanical and civil engineering rotary bridge design and systems analysis;
- a breast cancer MRI for diagnosis and therapy;
- a cornet path data and trend analysis;
- the analysis of human population smoking behaviors;
- the study of actuarial data for confirming and guiding quantitative analysis;
- the simulation of a complex process;
- the analysis of a simulation of a physical system;
- marketing posters and advertising.

In each case, the visualization provides an alternative to, or a supplement for, textual or verbal information. It is clear that visualization provides a far richer description of the information than the word-based counterpart. There are many reasons why visualization is important. Perhaps the most obvious reason is that we are visual beings who use sight as one of our key senses for information understanding. But visualization is also a necessary tool to make sense of the flood of information in today's world. Consider how much information we have to take in every day as part of our routine activities. E-mails arrive on our computes, credit card statements arrive from a bank every month, and last-minute holiday offers, stock market index variations, and advertising leaflets fill the mailbox. Therefore, we need effective methods that allow us to go through this information and, for example, help us make decisions. To be short, visualization →helps us think, Reduces load on working memory, offloads cognition and uses the power of human perception.

But which choice should you make for the visualization of data; 2D, 3D or VR? You know the difference between 3D and VR. VR is 3D++ that is, with user immersion, multi sensorial VE with interaction devices and stereoscopic images.

But what are the pros and cons of 3D versus 2D? Research show an improvement in answer times using 3D with no detriment of accuracy. Other work indicated that displaying data in 3D instead of 2D can make it easier for users to understand the data. Also is pointed out that realistic 3D displays could support cognitive spatial abilities and memory tasks, namely remembering the place of an object, better than with 2D.

On the other hand several problems arise such as intensive computation, more complex implementations than 2D interfaces, and user adaptation and disorientation.

In VR, the user can always access external information without leaving the environment and the context of the representation. Also, the user’s immersion in the data allows him to take advantage of stereoscopic vision that enables him to disambiguate complex abstract representations.

It is generally considered that only stereoscopy allows one to fully exploit the characteristics of the 3D representations. It helps the viewer to judge the relative size of objects and the distances between them. Furthermore an interactive method enables the user to process the image data 30 times faster than manually. As a result, it is suggested that human interaction may significantly increase overall productivity. Conclusion is that VR is most helpful to visualization tasks.

For inspiration: http://www.gapminder.org/

Grading

You make all the assignments of this tutorial and demonstrate and discuss them with the teacher for grading.

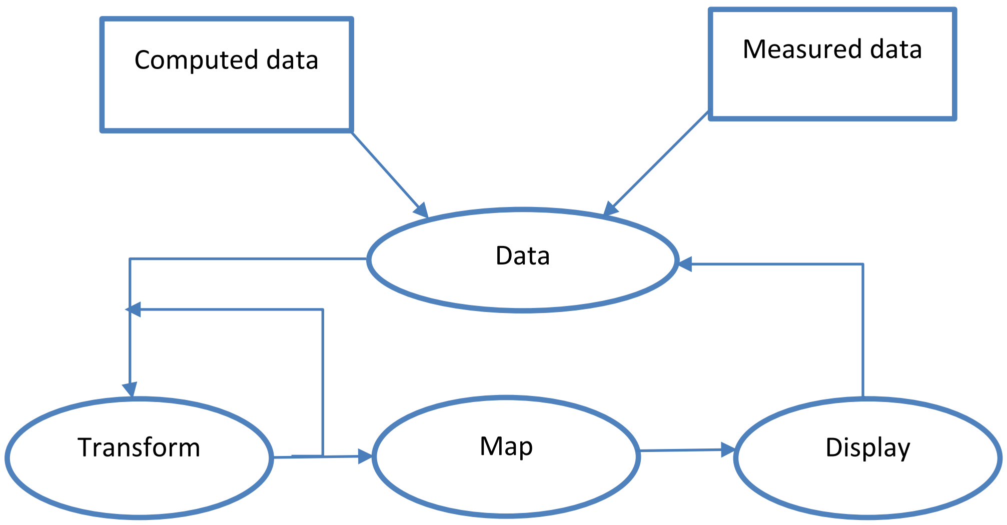
If you are ready for an exam, you can make an appoinment (1 hour), via the outlook agenda of the teacher

Mark:

- 7 All the assignments have been done correctly.
- <7 Some assignments haven't done correctly.
- >7 Assignments have been done correctly and in addition extra things have been done.

Part 1. An Introduction to the VisualizationToolkit

In this module you will attempt to address all the topics associated with data processing and visualization.

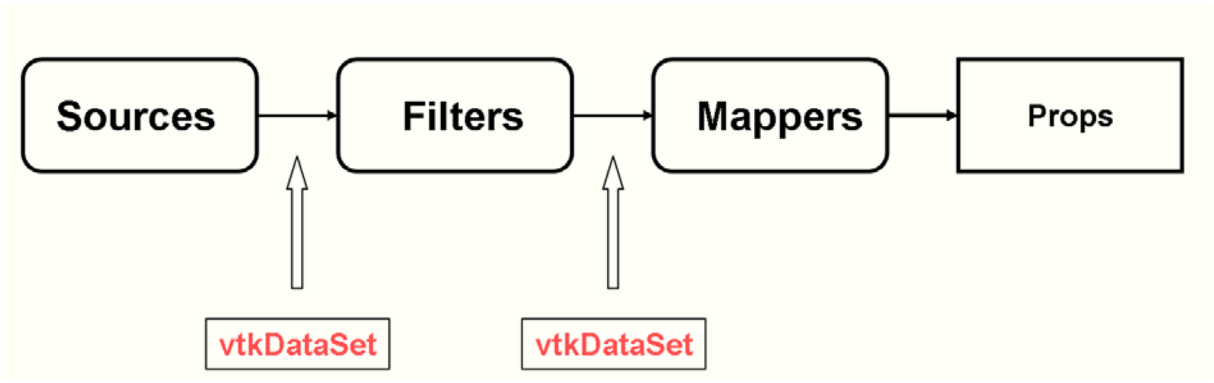


In the figure you see the visualization pipeline. As this figure illustrates we see that visualization process focuses on data. In the first step data is acquired from some source. Next the data is transformed by various methods, and then mapped to a form appropriate for presentation to the user. Finally, the data is rendered or displayed, completing the process.

This pipeline is used by all kind of visualization toolkits and also by VTK.

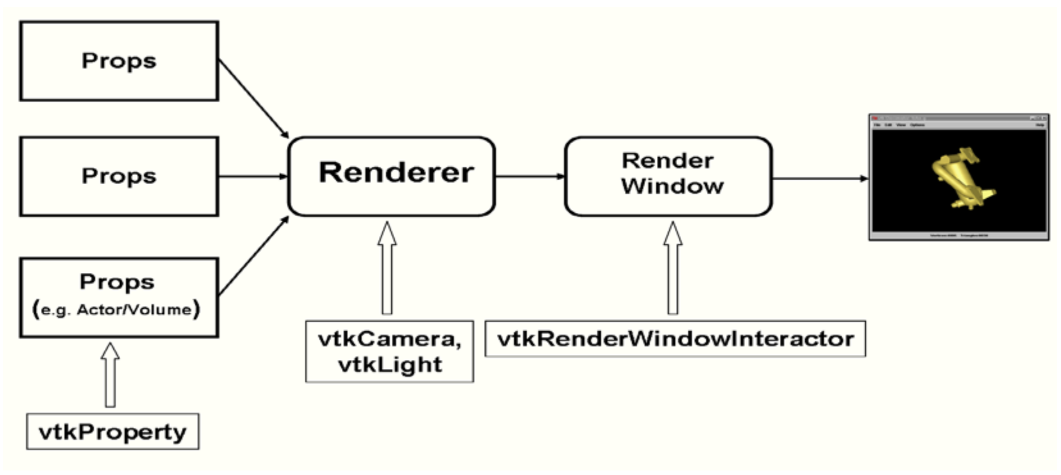
The Visualization Toolkit (VTK) has become one of the de facto standard tools in modern programming for image analysis. It has rich functionality for both image/surface processing and visualization. It's design pioneered the use of the combination of scripting and compiled languages in a single piece of software through the use of wrapping techniques for making available the underlying C++ classes in VTK as new commands in scripting languages such as Tcl and Python. The keys to successfully using VTK are (i) understanding the structure of the object-oriented hierarchy and (ii) understanding its pipeline architecture.

VTK is a large, complicated, powerful but often surprisingly easy to use toolkit for 3D image processing and visualization. It is an object-oriented library. Many operations in VTK are performed using a pipeline architecture where multiple elements are attached together to perform a complex task. A typical pipeline takes the form shown in the figures.



1. Sources – these classes produce data. For example, *vtkJPEGReader* reads a jpeg image from a file and generates an image output.
2. Filters – these operate on some data to produce a modified version. For example, *vtkImageGaussianSmooth* acts on an image to perform Gaussian smoothing and to produce a new smoothed image.
3. Mappers – these define the interface between data (e.g. images) and graphics primitives or software rendering techniques. A special kind of "mapper" like class are the writers which output the data to files (e.g. vtkJPEGWriter). Multiple mappers may share the same input, but render it in different ways.

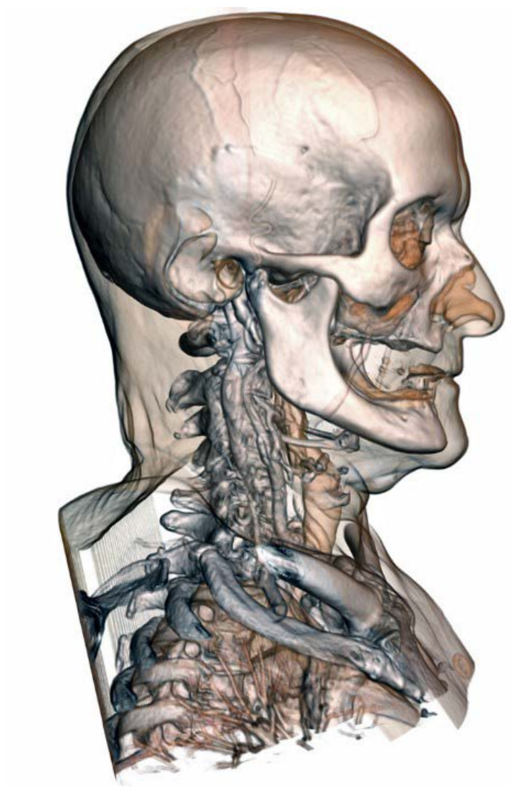
The second part of the pipeline consists of the elements that make up the virtual 3D world, namely:



1. Props/Actors – these take as input the output of a mapper and 'know' how to generate the visible representation of data. The type of rendering produced is governed by an auxiliary data structure known as a property (e.g. color, showing a surface as a wire-frame vs a full surface etc.). Props take as their input the output of a mapper. Mappers should not be shared among props. Volumes – these are special kinds of props that are used to display volume rendered images.
2. Renderer – Renderers are the classes that generate a 2D image from a 3D scene. They have attached actors as well as lights and cameras. More formally; "renderer is an object that controls the rendering process for objects. Rendering is the process of converting geometry, a specification for lights, and a camera view into an image. vtkRenderer also performs coordinate transformation between world coordinates, view coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device). Certain advanced rendering features such as two-sided lighting can also be controlled."
3. Render Window – the Render Window is the piece of screen real estate in which the virtual camera image is displayed. An important auxiliary item attached to a render window is an interactor which handles mouse/keyboard input to the window.

Exercise 1.1 Introduction to VTK

There is a lot of information about VTK on the Internet. Of course at http://www.vtk.org



VTK has an abstract graphics layer above the local host's graphic software interface (usually OpenGL). This insures cross-platform portability and creates a device independent graphics layer. In the graphics model class names were adapted from the movie-making industry. Lights, cameras, actors, and props are all used to create a scene.

Render Window

In order to visualize your data, you will need to open a window and create a renderer. **vtkRenderWindow** is the class you will use to create a window and **vtkRenderer** is the class you will use to create the renderer which will draw in the window. The vtkRenderWindow is a container class for vtkRenderer objects. Within a VTK application you can have multiple windows and multiple renderers tiled within each window.

Try this code:

vtkRenderWindow

- *Manages window(s) on display device*

```
import vtk
renWin=vtk.vtkRenderWindow()
renWin.SetSize(500,500)
ren1=vtk.vtkRenderer()
ren1.SetBackground(0.0,0.0,0.0)
renWin.AddRenderer(ren1)
renWin.Render()
```

vtkRenderer

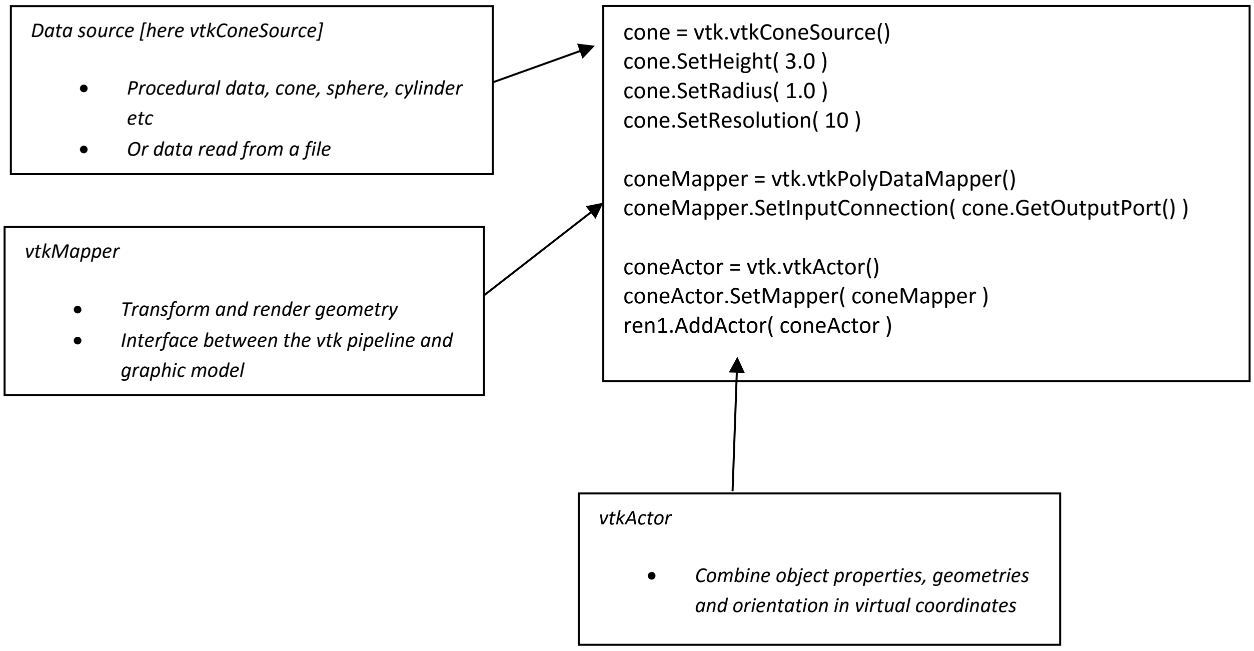
- *Coordinates lights cameras and actors to create and image*

- Change the background color in grey

Mappers, Properties, and Actors

Now that you know how to create a window, you can create an actor for our scene. **vtkActor** is the class you will use to represent 3D geometric data in the scene. To start you will need some data. You will use an instance of **vtkConeSource** to generate polygonal data for a cone. You will then use an instance of **vtkPolyDataMapper** to take the polygonal data from the vtkConeSource and create a rendering for the renderer. In the image you see an example to create a simple cone and a window and renderer.

Try this code:



Data source [here vtkConeSource]

- *Procedural data, cone, sphere, cylinder etc*
- *Or data read from a file*

vtkMapper

- *Transform and render geometry*
- *Interface between the vtk pipeline and graphic model*

vtkActor

- *Combine object properties, geometries and orientation in virtual coordinates*

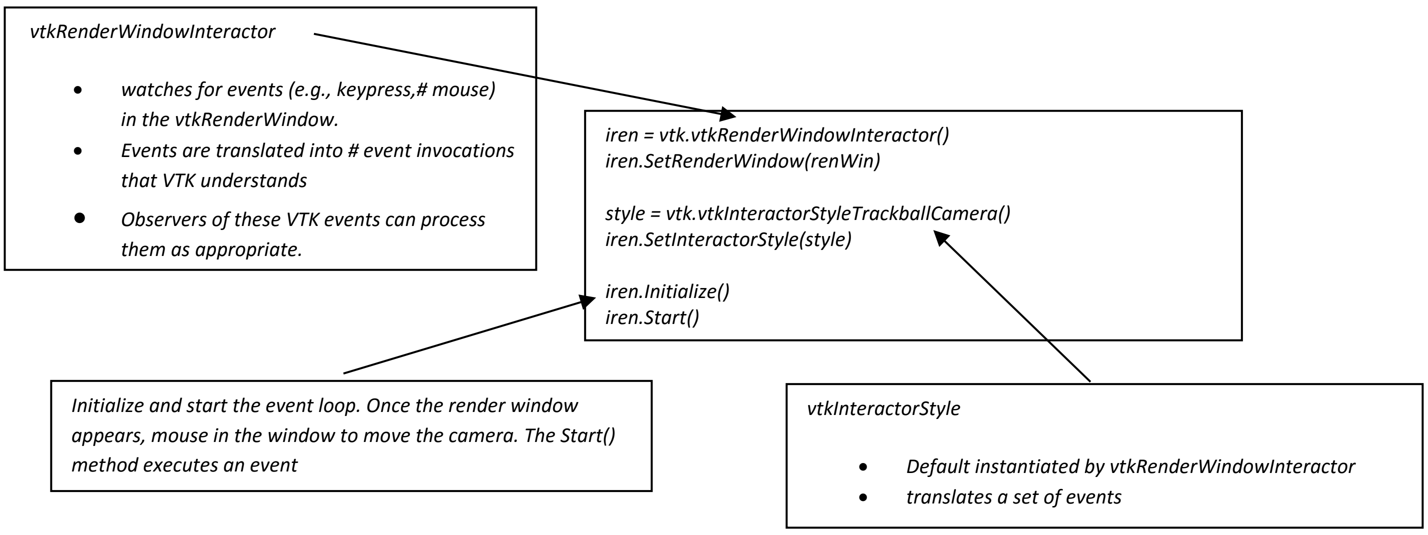
```
cone = vtk.vtkConeSource()
cone.SetHeight( 3.0 )
cone.SetRadius( 1.0 )
cone.SetResolution( 10 )

coneMapper = vtk.vtkPolyDataMapper()
coneMapper.SetInputConnection( cone.GetOutputPort() )

coneActor = vtk.vtkActor()
coneActor.SetMapper( coneMapper )
ren1.AddActor( coneActor )
```

Try experimenting:

- The resolution of the cone
- Change the size of render window
- Give the cone a color.
- Loop over 360 degrees and render de cone each time.

Interaction

In order to interact with the scene using the mouse you can use an instance of **vtkRenderWindowInteractor**. The interactor can be placed into joystick or trackball mode using the “j” or “t” keys. The left mouse button controls rotation, the middle mouse button controls panning, and the right mouse button controls zooming. The “r” key can be used to reset the camera. The “q” key is used to quit. Create a cone within an interactive window and renderer.



vtkRenderWindowInteractor

- *watches for events (e.g., keypress,# mouse) in the vtkRenderWindow.*
- *Events are translated into # event invocations that VTK understands*
- *Observers of these VTK events can process them as appropriate.*

```
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

style = vtk.vtkInteractorStyleTrackballCamera()
iren.SetInteractorStyle(style)

iren.Initialize()
iren.Start()
```

Initialize and start the event loop. Once the render window appears, mouse in the window to move the camera. The Start() method executes an event

vtkInteractorStyle

- *Default instantiated by vtkRenderWindowInteractor*
- *translates a set of events*

- Add another cone to the window (different color, size, position, rotation)
- Rotate the left mouse button and zoom in or out with the right.

Exercise 1.2 “Hello World” or something like that

In de textbox you see VTKcode to render a text on the screen. The user can also interact with the text by using the mouse. Run this code and compare it with the code of Exercise 2.1

```
# This program demonstrates how VTK can be used to render a text
# The user can also interact with the text by using the mouse

# load VTK
import vtk

# Create a Text source and set the text
text = vtk.vtkTextSource()
text.SetText("Hello World")

# Create a mapper and set the Text source as input
textMapper = vtk.vtkPolyDataMapper()
textMapper.SetInputConnection(text.GetOutputPort())

# Create an actor and set the mapper as input
textActor = vtk.vtkActor()
textActor.SetMapper(textMapper)
```

```
# Create a renderer
ren = vtk.vtkRenderer()

# Assign the actor to the renderer
ren.AddActor(textActor)

# Create a rendering window
renWin = vtk.vtkRenderWindow()

# Add the renderer to the window
renWin.AddRenderer(ren)

# Set the name of the window (this is optional)
renWin.SetWindowName("Hello World!")

# Make sure that we can interact with the application
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

# Initialze and start the application
iren.Initialize()
iren.Start()
```

Try this out:

- It is usually possible to change the color of objects. However it is not always done in a consistent way. Make the text red by using the method *SetForegroundColor()*
- By changing the first line to *from vtk import *,* you can remove all **vtk.** in the program. This usually makes the program a bit easier to read

Exercise 1.3 More objects in the scene and shading

Make a program with a vtkcylinder (height 4, radius 4) and a cone (height 12, radius 3). Position the cylinder in the origin and the cone at x=5. Give the objects a different color.

The overall appearance of the objects can be improved by adding some shading. By shading the objects the visual impression is improved. In this exercise you will apply some simple smooth shading by using the Phong illumination model.

The following way of doing shading can be applied to almost any object. Just change the property of the appropriate object. The quality of the shading will increase with the number of facets of the cone, so it will be better to increase the resolution:

cone.SetResolution(120)

Now you can proceed to change the shading parameters, which are set in a vtkProperty. Start by creating the object:

prop=vtkProperty()

Set the diffuse property of the object. This is how much of the diffuse light shall be reflected by the material.

prop.SetDiffuse(0.7)

Do the same for the specular part.

prop.SetSpecular(0.4)

You also need to specify the specular power, which controls the size of the highlight. A larger power will yield a smaller highlight and vice versa.

```
prop.SetSpecularPower(20)
```

Finally set the property for the object.

```
coneActor.SetProperty(prop)
```

Note that if you made any previous changes to the property of the coneActor they will be lost as you create a new property and set the property of the coneActor to be this new property. A better way (in this case) would be to change the existing properties. One way would be to read the existing property instead of creating it.

```
prop=coneActor.GetProperty()
```

Then set the properties.

```
prop.SetDiffuse(0.7)
prop.SetSpecular(0.4)
prop.SetSpecularPower(20)
coneActor.SetProperty(prop)
```

An even faster way, but perhaps less readable is to set the properties directly without even having a vtkProperty object.

```
coneActor.GetProperty().SetDiffuse(0.7)
coneActor.GetProperty().SetSpecular(0.4)
coneActor.GetProperty().SetSpecularPower(20)
```

The previous two examples will yield the same result and no previous properties are lost except for the new ones that are being set.

Try out the ambient light as well.

In the course V3D1 you got experience with 3D graphics concepts like a camera, viewing directions and lights. In vtk , if lights and cameras are not directly created, the renderer automatically instantiates them. If you wish to access a camera that already exists, you can ask the renderer for the active camera. For example: ren1.GetActiveCamera().Azimuth(150)

Try out the vtkCamera

- SetActiveCamera
- SetClippingRange
- SetFocalPoint
- SetPosition
- ComputerViewPlaneNormal
- SetViewUp
- Zoom
- Dolly

Try out the vtkLight

- AddLight
- SetColor
- SetFocalPoint
- Setposition
- PositionalOn

Part 2 Data from files

Today digital systems generate, store, and process vast amounts of "Big Data". This data is often heterogeneous with many interconnections and dependencies (relations).

Visualization starts with the data that is to be displayed. First step is to examine the characteristics of the data. Data comes from many sources: it can be gathered from sensors or surveys, or can be generated by simulations and computations. Data can be raw(untreated), or it can be derived from raw data via some process, such as smoothing, noise removal, scaling, or interpolation. Since our aim is to visualize data, clearly we need to know the character of the data.

First, visualization data is discrete. This is because we use digital computers to acquire, analyze, and represent our data, and typically measure or sample information at a finite number of points. A typically data set used in visualization consists of a list of n records. (r1,r2,.....rn). Each record ri consists of m(one or more) observations or variables. An observation may be a single number/symbol/string or a more complex structure.

A variable may be classified as either as independent or dependent. An independent variable is one whose value is not controlled or affected by another variable, such as the time. A dependent variable is one whose value is affected by a variation in one or more associated independent variables. Temperature for a region would be considered a dependent variable, as its value could be affected by variables such as date, time or location. We can think of data as being generated by some process or function. For example y=x^2. if we are using a conventional digital computer, we must discretize this equation to operate on the data it represents. For example, to plot this equation we would sample the function in some interval say [-1,1] (the domain of the independent variable) and then compute the value y (the range of the dependent variable) of the function at a series of discrete points in this interval.

Because of the discrete character of the data we do not know anything about the regions in between data values. This poses a serious problem, because an important visualization activity is determine data values at arbitrary positions. There is an obvious solution to this problem: interpolation. We presume a relationship between neighboring data values. Often this is a linear function, but we can use quadratic, cubic, spline or other interpolation functions.

A second important characteristic of visualization data is that its structure may be regular or irregular. Regular data has an inherent relationship between datapoints. For example, if we sample on an evenly spaced set of points, we do not need to store all the points coordinates, only the beginning position of the interval, the spacing between points and the total number of points. The point positions are then known implicitly, which can be taken of advantage of to save computer memory. Data that is not regular is irregular data. The advantage of irregular data is that we can represent information more densely where it changes quickly and less densely where the change is not great. Thus irregular data allows us to create adaptive representational forms, which can be beneficial given limited computing resources.

Finally, data has a dimension, from 0D points, to 1D curves, 2D surfaces, 3D volumes and even higher dimensions. The dimension of that data is important because it implies appropriate methods for visualization and data representation. For example, in 1D we naturally use x-y plots, bar charts or pie charts and store the data as a 1D list of values. For 2D data we might store the data in a matrix and visualize it with a deformed surface plot.

VTK and Data

VTK can handle all these different aspects of data. The Visualization Toolkit provides a number of source and writer objects to read and write popular data file formats. The Visualization Toolkit also provides some of its own file formats.
In VTK readers are source objects, and writers are mappers. What this means from a practical point of view is that readers will ingest data from a file, create a data object, and then pass the object down the pipeline for processing. Similarly, writers ingest a data object and then write the data object to a file. Thus, the readers and writers will interface to the data well if VTK supports the format, and you only need to read or write a single data object. If the data file format is not supported by the system, you will need to interface the data via the programming interface. If you wish to interface to a collection of objects, you will probably want to see whether an exporter or importer object exist to support. Typically importers and exporters are used to save or restore an entire scene (i.e. lights, cameras, actors, data, transformations, etc). When an importer is executed, it reads one or more files and may create several objects. For example, in VTK the vtk3DSImporter imports a 3D Studio file and creates a rendering window, renderer, lights, cameras and actors.

Exercise 2.1 obj file

Make a program that shows how an .obj file can be read and how the polygons can be rendered. The source will not be a predefined object like Cone, instead the vtkOBJReader class will be used.
Steps:

- Download an obj file from the intranet
- Create an .obj reader and set the file name.
- Create a Mapper and an Actor. Note that this part is very similar to what did in the example Cone.
- Then proceed as "usual", adding the actor to the renderer and creating a window etc.
- Try out the ‘Translate’ and ‘Rotate’ transforms on the model.

Exercise 2.2 vtk file

VTK has also an own format with extension vtk. See http://www.vtk.org/VTK/img/file-formats.pdf

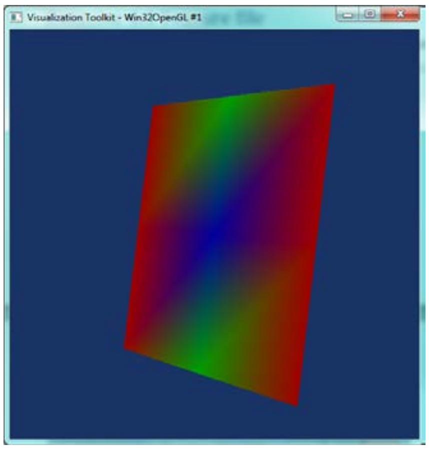
Use the VTK reader called *vtkStructuredPointsReader() or the vtkPolyDataReader()* to read the brain.vtk and skin.vtk file and set the file name with *SetFileNaam(…).*

Because VTK has a pipeline architecture you have to connect the output from the reader to the input of a mapper (or filter). This is done with the method; *SetInput(reader.GetOutput()).*

Exercise 2.3 Your own VTK file

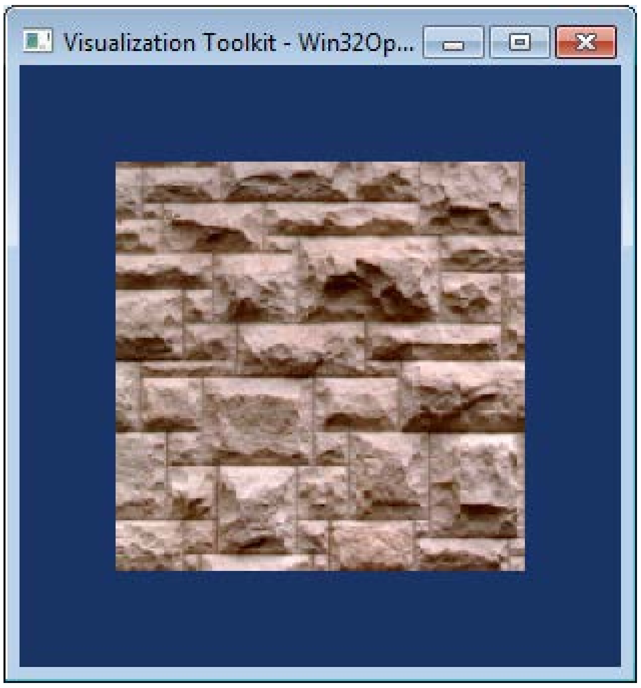
Now make your own vtk-file (ascii & unstructured grid)
Points: 0 0 0,0 1 0,0 2 0,1 2 0,1 1 0,1 0 0,2 0 0,2 1 0,2 2 0
Use 4 times celltype 9(quad)
Cells: 0 5 4 1,1 4 3 2, 5 6 7 4, 4 7 8 3
Colortable: 0,0,0,1,2,1,0,0,0

The result is something like this.



Exercise 2.4 Texture file

A texture is in VTK a vtkImageData dataset. You can load the image with a reader, for example with the vtkBMPReader() and SetFileName(…). vtkTexture is a filter, so use it in the way you did before. You can map the texture on a plane (vtkPlaneSource()). The actor of this plane has the method SetTexture(tex).



Exercise 2.5 The visualization pipeline

In this exercise you will familiarize yourself with the stages required to create a visualization pipeline in VTK. The pipeline you are going to create will display the bone surface of a knee joint from a CT scan.

Reading the data: The datafile from the CT scanner is located on the intranet and is called `vw_knee.slc`. In order to read this file into VTK you will need an SLC file reader object called `vtkSLCReader` to read this volumetric file format.

Creating the Surface: We use the Marching Cubes algorithm to create the surface. This algorithm is implemented by the `vtkContourFilter` object. Looking in the help for the definition of this object you will see it has a method to specify the value of the created surface, try starting with 80 density units. The filter can create more than one contour surface, they are specified with an index before the contour value.

Extra objects: To finish the application you will need to add an actor, a mapper and a rendering window. The marching cubes algorithm produces triangular polygons in the same manner as the `vtkConeSource` object did in the previous exercise, so the same `PolyDataMapper`, actor and window objects can be used.

Linking together: Now that all the objects have been initialised, the pipeline can be set up. The syntax is the same as in the cone example – use the method `SetInput` with the substituted results from calling the object to connect with the method `GetOutput`. To connect a mapper to an actor use the method `SetMapper` in the actor, and similarly to add an actor into the renderer, use the method `AddActor`.

Setting properties in the actor: One thing you will notice when you build the pipeline is the hideous default colour of the surface. This is due to the mapper colouring the surface by its default redblue color table. In order to fix this you need to set the mapper to not color its output surface and set the color explicitly in the actor. Firstly use the method `ScalarVisibilityOff` in the mapper. Colors in VTK are set via the `vtkProperty` object. In order to access this object you need to call the actor with the method `GetProperty`. This will return an object which has a method `SetColor` which can be called with 3 floating point numbers describing red, green and blue color values. Try setting the color to something approximating bone.

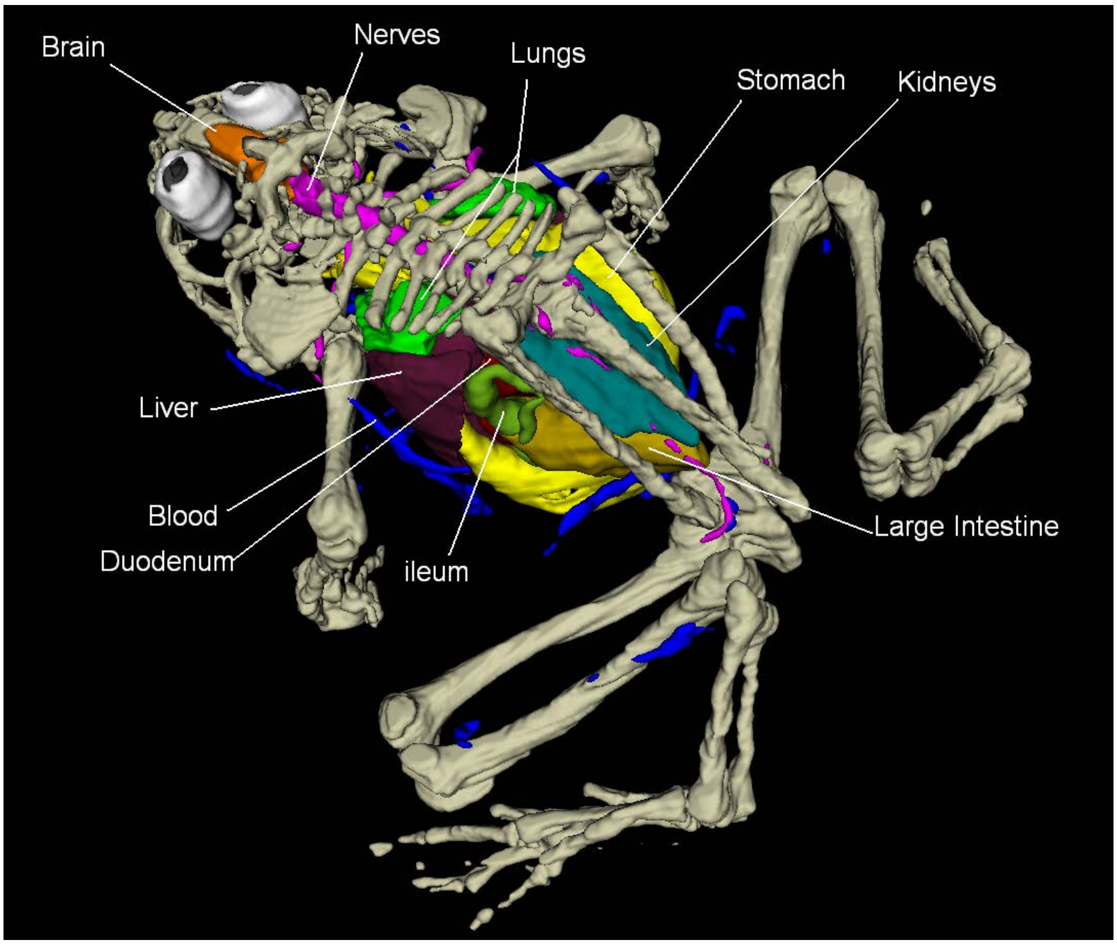
Subsampling the data: The second thing you will notice after the hideous color is the appallingly slow interaction and loading speed. This is caused by the MarchingCubes object (`vtkContourFilter)` producing too many triangles for rendering at interactive rates. One way of fixing this is to subsample the original grid of points. VTK provides the object `vtkExtractVOI` to perform this task, use the method `SetSampleRate` with an appropriate factor, specified for each axis.

Tidying up: As a finishing touch add an outline round the data with a `vtkOutlineFilter` object. This object takes as input the data and produces polydata which can be mapped using a `vtkPolyDataMapper` object.

An alternative volume data file `neghip.slc` is also available on the course web page. For this file try a surface value of 0.5 with your VTK script.

Exercise 2.6 The Virtual Frog

Make a 3D model from the Frog.zip. There are 16 vtk files for the different organs. Just make an actor for every organ and give them a different color. The skin is on a wrong position, so place the skin over skeleton. Also make it possible to toggle invisibility for different organs.


Brain
Nerves
Lungs
Stomach
Kidneys
Liver
Blood
Duodenum
ileum
Large Intestine

Part 3 Visualization techniques

Exercise 3.1 Scalar Visualization Algorithms

A scalar field is a function from space or space-time to a real value. This real value typically reflects a scalar physical parameter at every point in space(or in space and time). One example is temperature, which is a scalar quantity defined everywhere in space and time. In a visualization context, we work with discrete scalar fields that are defined on a grid. Each point in the grid is then associated with a scalar value. So scalars are single data values associated with each point/cell in the dataset. There are many different algorithms to visualize scalar data. Two common algorithms are Color Mapping and Contouring.

Exercise 3.1.1 Color Mapping

In color mapping scalar values are mapped through a lookup table to a specific color. Scalar values are used as an index into a color lookup table.

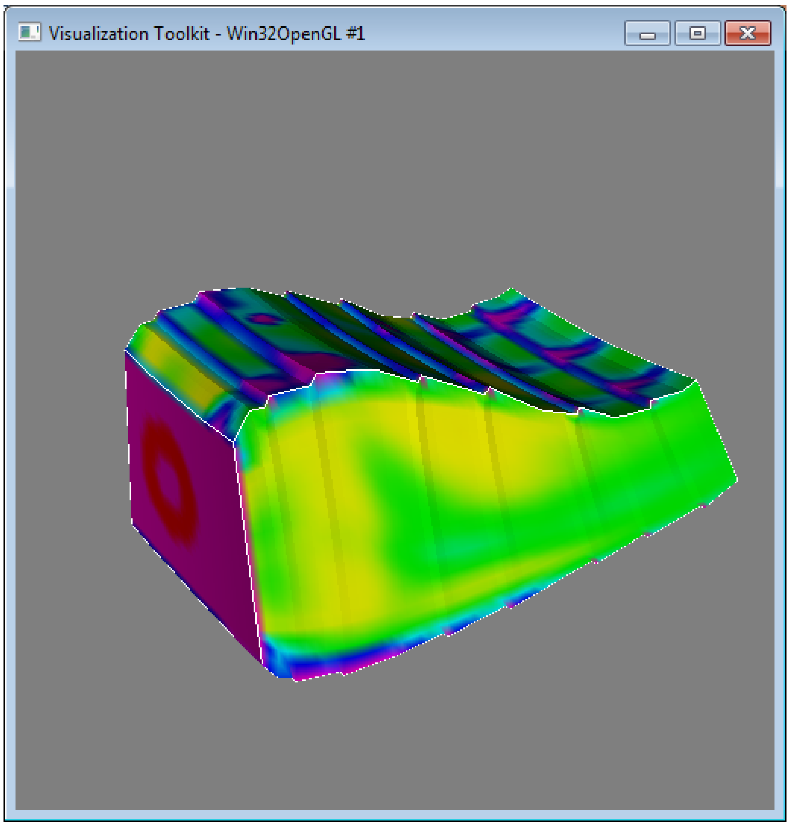
$$index = NumberOfColors * \frac{InputScalar - MinimumValueColor}{MaximumValueColor - MinimumValueColor}$$

The primary VTK classes used for color mapping are **vtkLookupTable** and **vtkDataSetMapper**. The basic idea is to specify a HSVA (Hue-Saturation-Value-Alpha) ramp and then generate the colors in the table by using linear interpolation into the HSVA space. Using an instance of vtkLookupTable you can set both the number of colors (size of the table) and the hue range for the table.

colormapping

Read the file “density.vtk” with the *vtkStructuredGridReader()* in your application. The mapper is a now *vtkDataSetMapper().* Make a vtkLookupTable and initialize some properties. For example *SetNumberOfColors, SetHueRange, SetSaturationRange, SetValueRange, SetAlphaRange, SetRange*

For the mapper the LookupTable and the ScalarRange is set. The rendering is like you did before.



Try this out:

- Change the number of colors in colormap
- Reverse the Hue Range
- Change the Scalar Range

Exercise 3.1.2 Contours / Isosurfaces

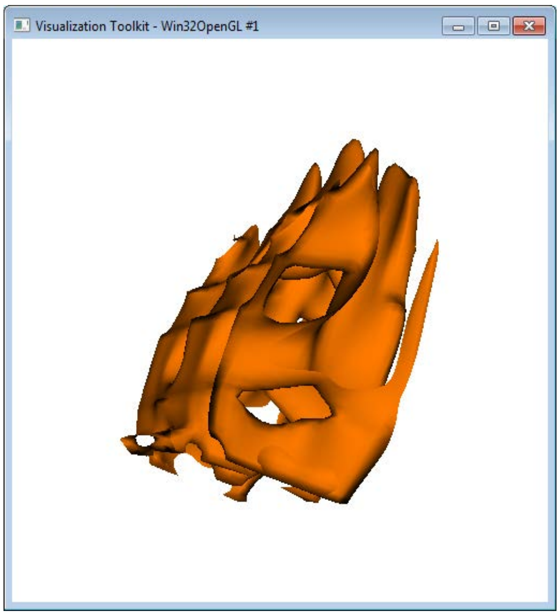
A contour plot is another useful technique for visualizing scalar fields. The primary examples on contour plots from everyday life is the level curves on geographical maps, reflecting the height of the terrain. Mathematically, a contour line, also called an isoline, is defined as the implicit curve `f(x,y)=c`. The contour levels c are normally uniformly distributed between the extreme values of the function `f` (this is the case in a map: the height difference between two contour lines is constant), but in scientific visualization it is sometimes useful to use a few carefully selected c values to illustrate particular features of a scalar field.

For 3D scalar fields, isosurfaces or contour surfaces constitute the counterpart to contour lines or isolines for 2D scalar fields. An isosurface connects points in a scalar field with (approximately) the same scalar value and is mathematically defined by the implicit equation `f(x,y,z)=c`.

So contouring is a technique where one constructs a boundary between distinct regions in the data, and contours are lines or surfaces of constant scalar value. This is a natural extension from color mapping as our eyes instinctively separate similarly colored areas into distinct regions. The first step in contouring is to explore the data space to find points near a contour or region of interest. Once found these points are then connected into either contour lines (**isolines**) for two-dimensionsal data or into surfaces (**isosurfaces**) for three-dimensional data. The lines or surfaces can be color mapped using the scalar data. The primary VTK classes used for contouring are **vtkContourFilter** and **vtkPolyDataMapper** (vtkContourFilter generates polygonal data).

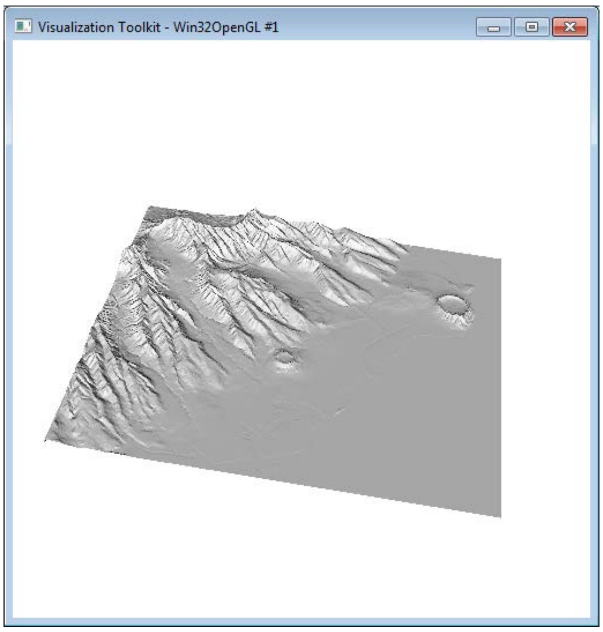
Isolines

You can read subset.vtk for isolines and the density.vtk for isosurfaces. Connect the reader with a *vtkContourFilter()*. For filtering *SetValue(0,0.26)* change this, to see the effect. The mapper is again a *vtkPolyDataMapper()*.



Try this out:

- Make this image with Honolulu.vtk



Exercise 3.2 Vector Visualization Algorithms

A vector field is a function from space or space-time to a vector value, where the number of components in the vector corresponds to the number of space dimensions. Primary examples on vector fields are the gradient of a scalar field; or velocity, displacement, or force in continuum physics.

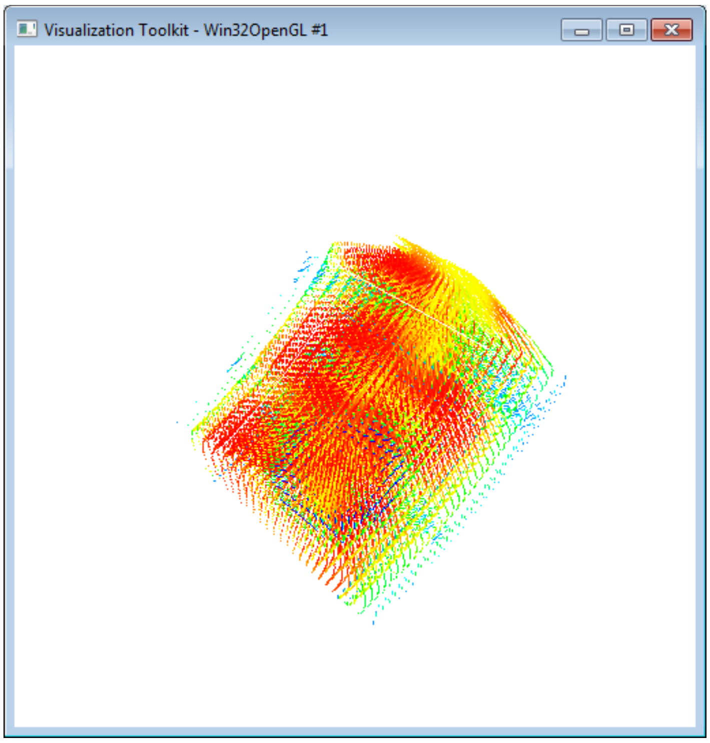
So vector data is a three-dimensional representation of direction and magnitude associated with each point/cell in the dataset and vector data is often used to describe rate of change of some quantity. For example vectors can be used to describe fluid flow. There are several algorithms that can be used to visualize vector data.

Exercise 3.2.1 Hedgehogs

One visualization technique for vector data is to draw an oriented, scaled line for each vector. The lines may be colored according to vector magnitude or some other scalar quantity (e.g. temperature or pressure). The result looks similar to a bristly hedgehog. Often you will need to adjust the scaling of the lines to control the size of its visual representation. The primary VTK class used for generating hedgehogs is **vtkHedgeHog**.

hedgehog

Again use density.vtk for this visualization. Make a lut and a *vtkHedgeHog()* filterpipe for the output of the reader. With *SetScaleFactor(float)*, you can set the scale factor to control size of oriented lines. Use a *vtkPolyDataMapper()* as a mapper and set the Lut and Scalarrange of it.

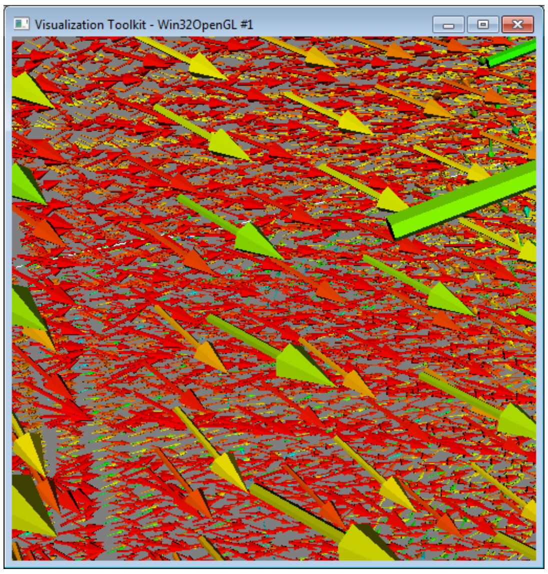


Exercise 3.2.2 Oriented Glyphs

A similar technique to using hedgehogs is to use oriented glyphs. Glyphs are polygonal objects such as a cone or an arrow and can be used in place of the lines in the hedgehogs. They too can be colored and scaled. The orientation and color of the glyph can indicate the direction and magnitude of the vector. The primary VTK class used for generating orientated glyphs is **vtkGlyph3D**.

Isolines

Again use density.vtk for this visualization. Make an arrow object and set its properties. Then make a glyph object. May you need to filter points if too many arrows cause clutter.



```
glyph=vtk.vtkGlyph3D()
glyph.SetSource(arrow.GetOutput())
glyph.SetInputConnection(reader.GetOutputPort())
glyph.SetVectorModeToUseVector()
glyph.SetColorModeToColorByScalar()
glyph.SetScaleModeToDataScalingOff()
glyph.OrientOn()
glyph.SetScaleFactor(0.2)
```

Try out: SetScaleModeToScaleByScalar and SetScaleModeToScaleByVector.

Exercise 3.2.3 Streamlines

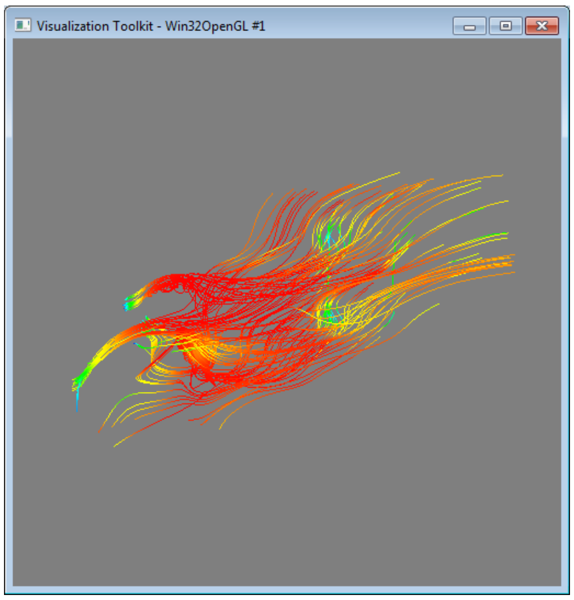
A streamline can be thought of as the path a massless particle takes flowing through a velocity field (i.e. vector field). Streamlines can be used to convey the structure of a vector field by providing a snapshot of the flow at a given instant in time. Multiple streamlines can be created to explore interesting features in the field. Streamlines are computed via numerical integration (integrating the product of velocity times delta T). Creating streamlines will require several VTK classes. First you will need to specify a starting point or points for the streamline(s). You can use either the VTK class **vtkPointSource** or **vtkLineSource** to create the starting points. Next you will need to pick an integrator such as **vtkRungeKutta4** to do the numerical integration. Finally you will need to create a stream tracer using **vtkStreamTracer**.

Streamlines

Again use density.vtk for this visualization and make the starting point and the streamtracer in this way.

```
seeds=vtk.vtkPointSource()
seeds.SetRadius(3.0)
seeds.SetCenter((reader.GetOutput()).GetCenter())
seeds.SetNumberOfPoints(100)

integrator=vtk.vtkRungeKutta4()

streamer=vtk.vtkStreamTracer()
streamer.SetInputConnection(reader.GetOutputPort())
streamer.SetSourceConnection(seeds.GetOutputPort())
streamer.SetMaximumPropagation(100)
streamer.SetMaximumPropagationUnitToTimeUnit()
streamer.SetInitialIntegrationStepUnitToCellLengthUnit()
streamer.SetInitialIntegrationStep(0.1)
streamer.SetIntegrationDirectionToBoth()
streamer.SetIntegrator(integrator)
```

Use a *vtkPolyDataMapper()* as a mapper.



Exercise 3.3 Modeling Visualization Algorithms

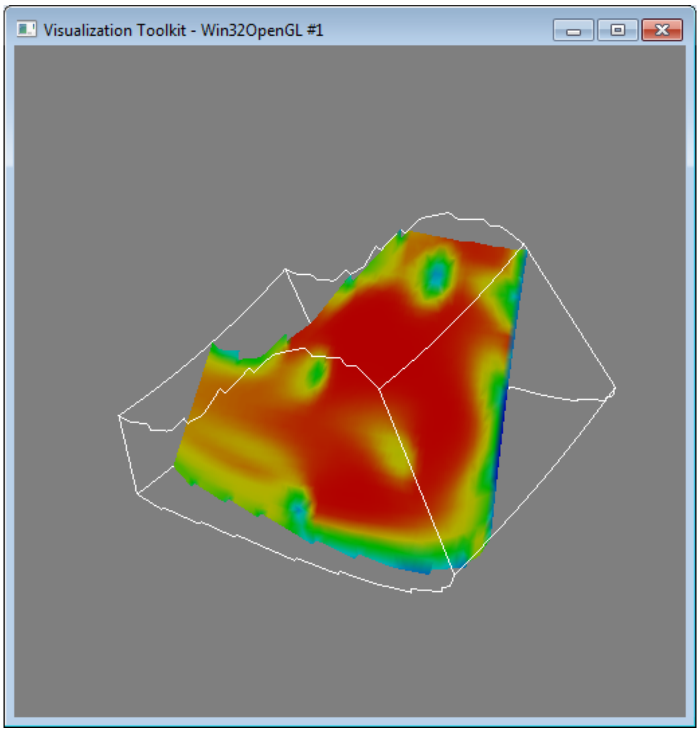
Modeling algorithms change dataset geometry or topology. They are often used to reveal internal details of a data set. Two common modeling visualization techniques are cutting (slicing) and clipping.

Exercise 3.3.1 Cutting/Slicing

Cutting also called slicing entails creating a "cross-section" of the dataset. Cutting uses an implicit function to define a surface with which to cut the dataset. Any type of implicit function may be used to create the surface. One simple technique is to use a plane to define the cutting surface thereby creating a planar cut. The cutting surface interpolates the data as it cuts and can be visualized using any of the standard visualization techniques. The result of cutting is always of type vtkPolyData. To create a planar cut we will need to create a plane using the class **vtkPlane** and we will need the VTK class **vtkCutter** to do the actual cutting.

Cutting

Again use density.vtk for this visualization. Make a vtkPlane() and set the origin of the plane on the center of density. Use SetNormal() to give the plane a normal. Then give the plane to the SetCutterFunction of vtkCutter. Then make like always a mapper, actor and a renderer.

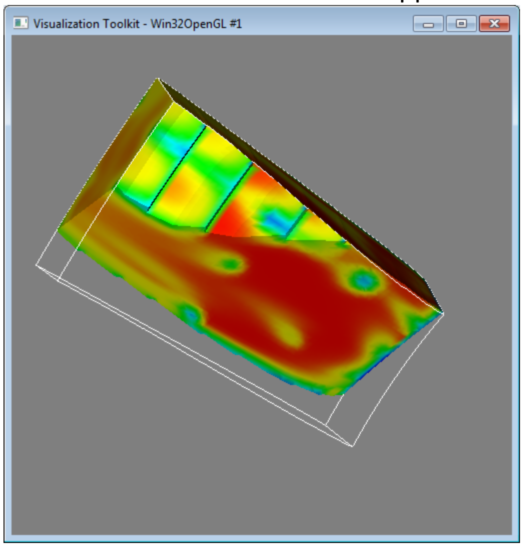


Change the normal to orientate the cutting plane.

Clipping

Clipping, like cutting, also uses an implicit function to define a surface. But instead of creating a cross-section of the dataset, clipping uses the surface to "cut" through the cells of the dataset, returning everything inside of the specified implicit function. The result of clipping is an unstructured grid. You can again create a plane using the class **vtkPlane** and use the VTK class **vtkClipDataSet** to do the actual clipping of the dataset.

The only change in the former program is the vtkClipDataSet. Connect this filter to the outputport of the reader. Also use a DataSetMapper instead of a PolyDataMapper.



Try out:

- Clipping with a sphere instead of a plane

Volume of interest

The filter vtkExtractVOI extracts pieces of the input image dataset, and can also perform subsampling on it. The output of the filter is also of type vtkImageData.

vtkExtractVOI will extract a subregion of the volume and produce a vtkImageData that contains exactly this information. In addition, vtkExtractVOI can be used to resample the volume within the VOI.

Try this out:

```
import vtk

quadric=vtk.vtkQuadric()
quadric.SetCoefficients(.5,1,.2,0,.1,0,0,.2,0,0)

sample=vtk.vtkSampleFunction()
sample.SetSampleDimensions(30,30,30)
sample.SetImplicitFunction(quadric)
sample.ComputeNormalsOff()

extract=vtk.vtkExtractVOI()
extract.SetInput(sample.GetOutput())
extract.SetVOI(0,29,0,29,15,15)
extract.SetSampleRate(1,2,3)

contours=vtk.vtkContourFilter()
contours.SetInput(extract.GetOutput())
contours.GenerateValues(13,0.0,1.2)
```

```
contMapper=vtk.vtkPolyDataMapper()
contMapper.SetInput(contours.GetOutput())
contMapper.SetScalarRange(0.0,1.2)

outlineActor=vtk.vtkActor()
outlineActor.SetMapper(contMapper)

ren1=vtk.vtkRenderer()
renWin=vtk.vtkRenderWindow()
ren1.SetBackground(0.5,0.5,0.5)
renWin.SetSize(500,500)
renWin.AddRenderer(ren1)

iren=vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

ren1.AddActor(outlineActor)
ren1.SetBackground(1,1,1)
renWin.SetSize(500,500)
```

Exercise 3.4 Visualize raw data

Get the file “coordinaten.txt”and “waarden.txt” from sharepoint. These files contain 16 xyz points and 16 vector values for these points. So how can you visualize this data in VTK?

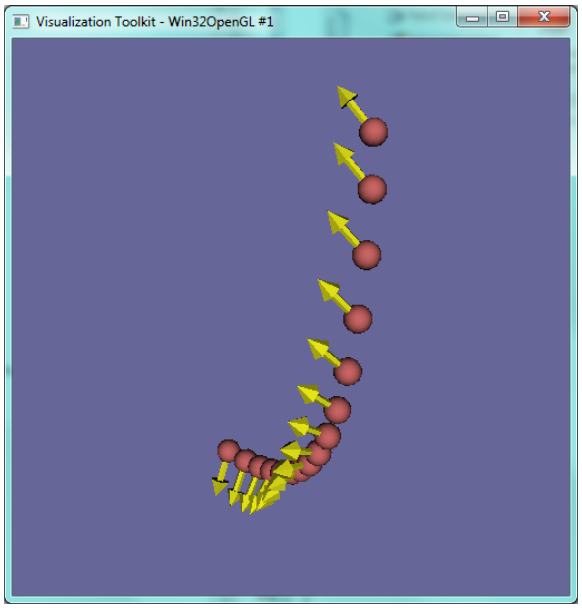
You can make a vtk file from this data or handle the data in your program.

Possible steps if handle the data:

- Load the data in two numpy arrays.
- Make a dataset of the type unstructured grid.
- Make vtkPoints for your pointdata.
- Make an 3 component vtkDoubleArray for your vectordata.
- Put the numpy array data in the points and in the vectors double_array
- Set points and vectors in the dataset “UnstructuredGrid”. For vectors you need ‘GetPointData()’.

Now you can visualize this data. You can use glyphs for this visualization.

For example balls with arrows.


Visualization Toolkit - Win32OpenGL #1

Part 4 Volume visualization

Volume rendering is a technique for showing certain values in a 3D volume of data. Each element is called a voxel and contains some value. 3D volumes are obtained in different ways and can contain for example medical or meteorological data. The main problem is to visualize the interesting values in an informative way and removing the non-essential data.

Inspiration → http://www.cs.utah.edu/~ramanuja/sci_vis/prj4/REPORT.html

Exercise 4.1 Simple volume rendering with MIP

Maximum Intensity Projection (MIP) is a volume rendering technique commonly used to depict vascular structures.

Class reference: *vtkVolumeRayCastMIPFunction is a volume ray cast function that computes the maximum value encountered along the ray. This is either the maximum scalar value, or the maximum opacity, as defined by the MaximizeMethod. The color and opacity returned by this function is based on the color, scalar opacity, and gradient opacity transfer functions defined in the* ***vtkVolumeProperty*** *of the* ***vtkVolume****.*

Steps:

1. Load "Torso.vtk". What is the datatype?
2. Create a Maximum Intensity Projection (MIP) function and a Volume Ray Cast Mapper. Connect them and also connect to the input data.
3. Create a Volume and set the Mapper.
4. Create the renderer and note that AddActor is not used, instead AddVolume is used.
5. Now you can proceed as usual.
6. Try out another sample distance.

Exercise 4.2 3D Medical imaging

Radiology is a medical discipline that deals with images of human anatomy. These images come from a variety of medical images devices, including X-ray, X-ray Computed Tomography (CT), Magnetic Resonance Imaging (MRI), and ultrasound. Each imaging technique has particular diagnostic strengths. The choice of technique is the job of the radiologist and the referring physician. For the most part, radiologist deal with two-dimensional images, but there are situations when three-dimensional models can assist the radiologist's diagnosis. Radiologists have special training to interpret the two-dimensional images and understand the complex anatomical relationship in these two-dimensional representations. However, in dealing with referring physicians and surgeons, the radiologist sometimes has difficulty communicating these relationships. After all, a surgeon works in three dimensions during the planning and execution of an operation; moreover, they are much more comfortable looking at and working with three-dimensional models.

The exercise deals with CT data. A CT image consist of levels of gray that vary from black (for air), to gray (for soft tissue), to white (for bone). See Headsq.zip on the intranet.

Read the input

Medical images come in many flavors of file formats. Here we use flat files without header information. Each 16-bit pixel is stored with little-endian byte order. Also, as is often the case, each slice is stored in a separate file with the suffix being the slice number of the form *prefix.1*, *prefix.2*, and so on (exercise 1.11). Medical imaging files often have a header of a certain size before the image data starts. The size of the header varies from file format to file format.

VTK provides several image readers including one that can read raw formats of the type described above vtkVolume16Reader.

Lookup:

- *SetDataDimensions*
- *SetImageRange*
  SetDataByteOrderToLittleEndian
- *SetFilePrefix*
- *SetDataSpacing*

To read this data we instantiate the class and set the appropriate instances variables as follows.

```
reader=vtk.vtkVolume16Reader()
reader.SetDataDimensions(64,64)
reader.SetImageRange(1,93)
reader.SetDataByteOrderToLittleEndian()
reader.SetFilePrefix("headsq/quarter")
reader.SetDataSpacing(3.2,3.2,1.5)
```

Illustration:

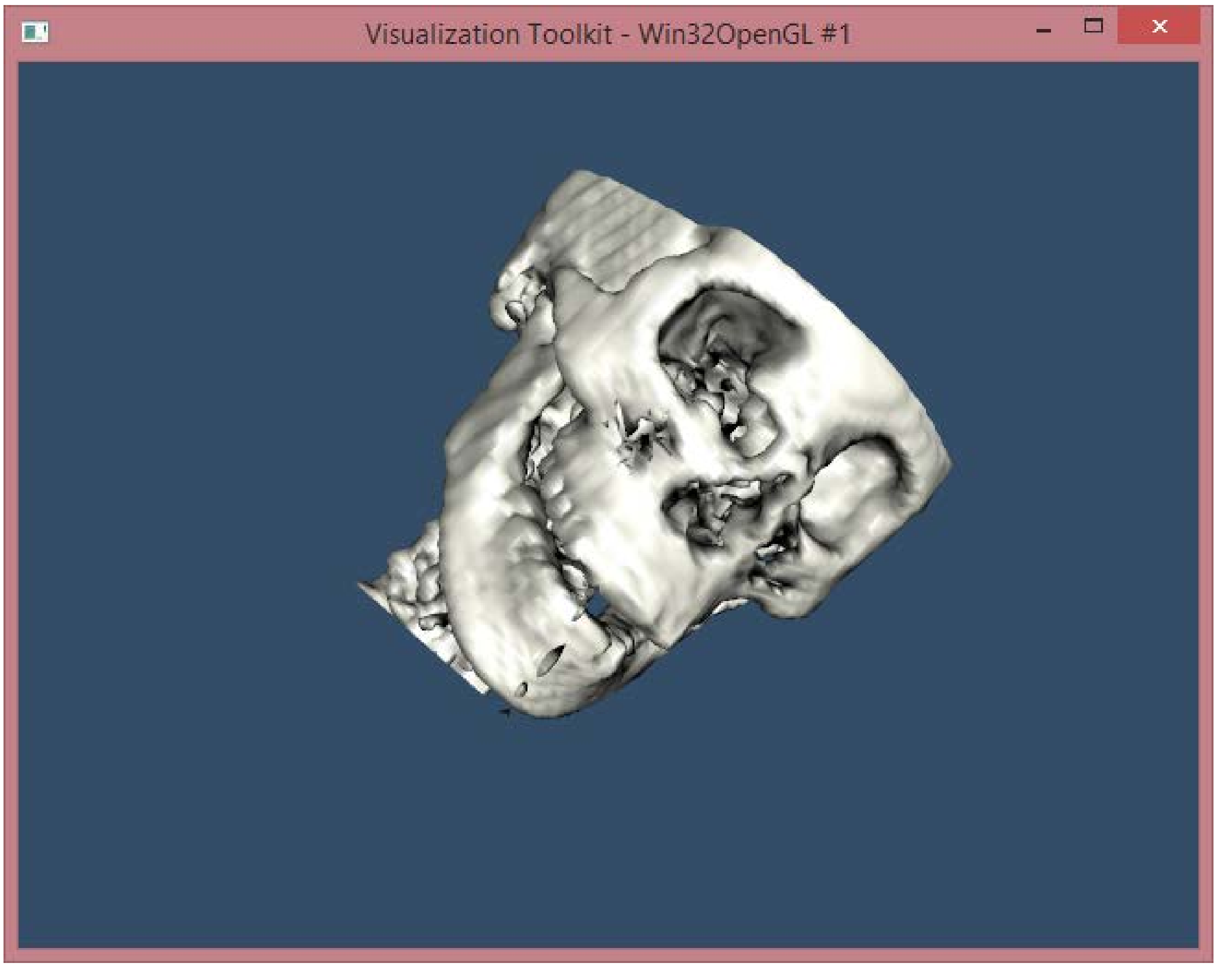
The reader is used to read a series of 2D slices(images) that compose the volume. The slice dimensions are set, and the pixel spacing. The data endianess must also be specified. The reader uses the fileprefix in combination with the slice number to construct filenames using the format FilePrefix.slicenumber. In this case the Fileprefix is the root name of the file: quarter. The original dataset was 256^2 now the slices are 64^2 pixels. That is why the resulting dataset is called quarter. The DataSpacing is set to 3.2 mm per pixel because of change in resolution.

Create an Isosurface

In this part of the exercise you generate an isosurface for the skin. The steps are familiar.

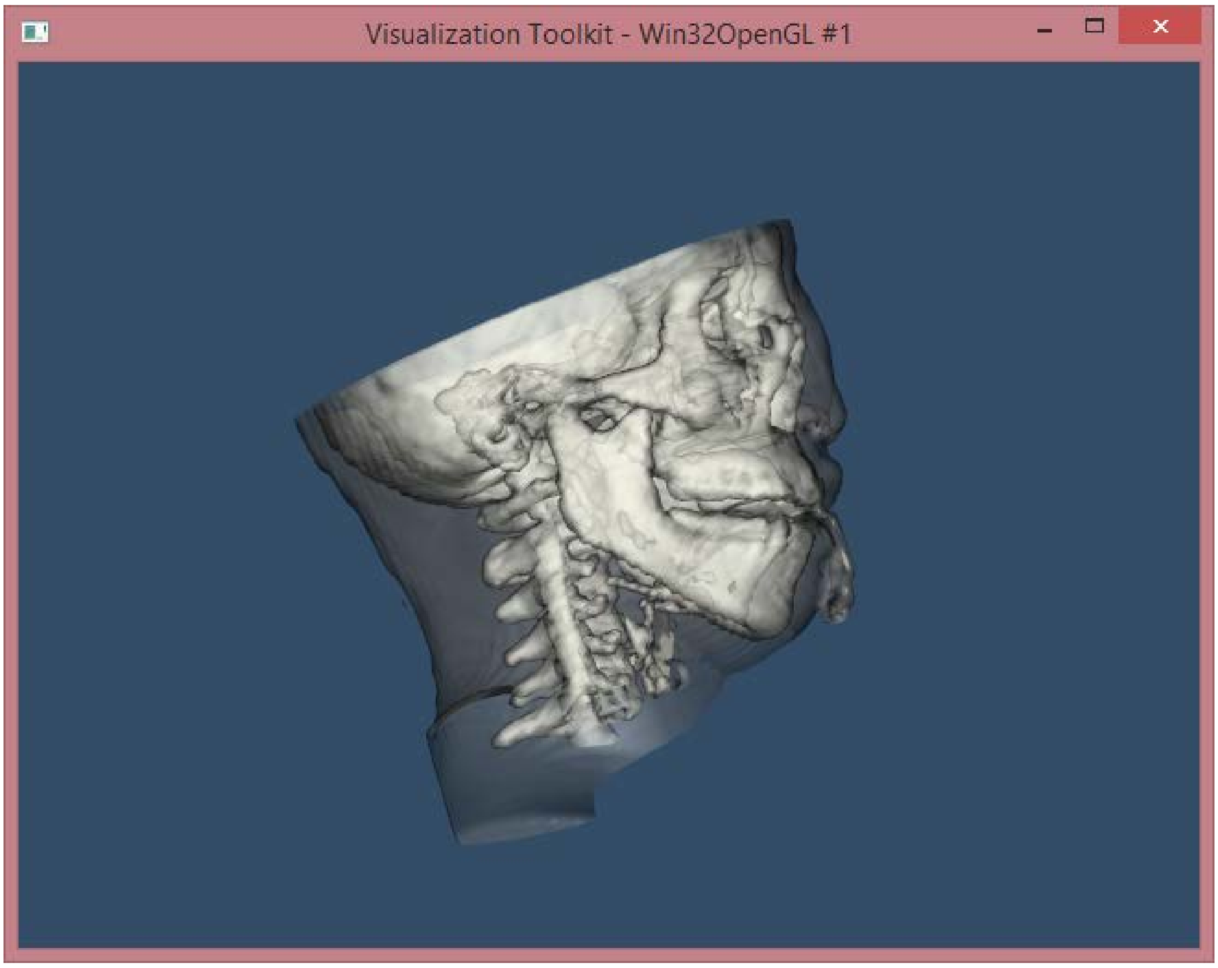
Steps:

1. Connect a contourfilter.
2. The value 500 and 1150 are known to correspond to the skin and bone of the patient.
3. Connect a PolyDataMapper and set ScalarVisibilityOff()
4. Connect a renderer, actor and camera. (choose a bonecolor!



Make a skinfilter and render both tissues.

You can improve the visualization in a number of ways. First, you can choose a more appropriate color for the skin. Use SetDiffuseColor to get a fleshy tone for the. Also add a specular component to the skin surface and you can experiment with the opacity.



Exercise 4.3 Knee visualization

The aim of the practical is to produce an enhanced visualization of the human knee dataset (`vw_knee.slc`).

Together with the material presented in lectures and previous practical assignments you are required to come up with an improved visualization of this dataset.

A surface for the skin

Using the same pipeline as for the bone, add an additional surface with an approximate density for skin. Start with a value of 50 density units, but this value is not the best so a little experimentation is required. Choose an appropriate color for skin. It may be difficult to see both the bone and skin surface at the same time in the same window. Try adding an extra `renderWindow` and/or renderer object to see the two visualizations side by side to determine a suitable method of visualization.

Seeing through the skin

Assignment 4.4 asked you to set the color of the surface by calling a method in the `vtkProperty` object. As well as having a property for color, the object also has a property for opacity (transparency). Try modifying the opacity property to make the skin surface semitransparent.

The surface actually has two `vtkProperty` objects associated with it – one for the front faces of the polygons and one for the backfaces.

The color and transparency of front and back faces of the skin surface can hence be set independently, a useful technique for giving a greater sense of ‘3D depth’ yet allowing the viewer to see through the skin. Try experimenting with different combinations of color and opacity to improve your visualization.

Cutting a hole in the skin

Transparency can be useful for seeing through surfaces, but it is often difficult to see details in a semitransparent object. The alternative is to use an opaque surface but cut a circular hole in it in an appropriate place by clipping it with an implicit surface. To do this, create an implicit sphere using the `vtkSphere` object. Polygons can be cut by the sphere using the `vtkClipPolyData` object.

Hint : set the following attributes in `vtkClipPolyData` : `GenerateClippedOutputOn`

`GenerateClipScalarsOn SetValue 0`

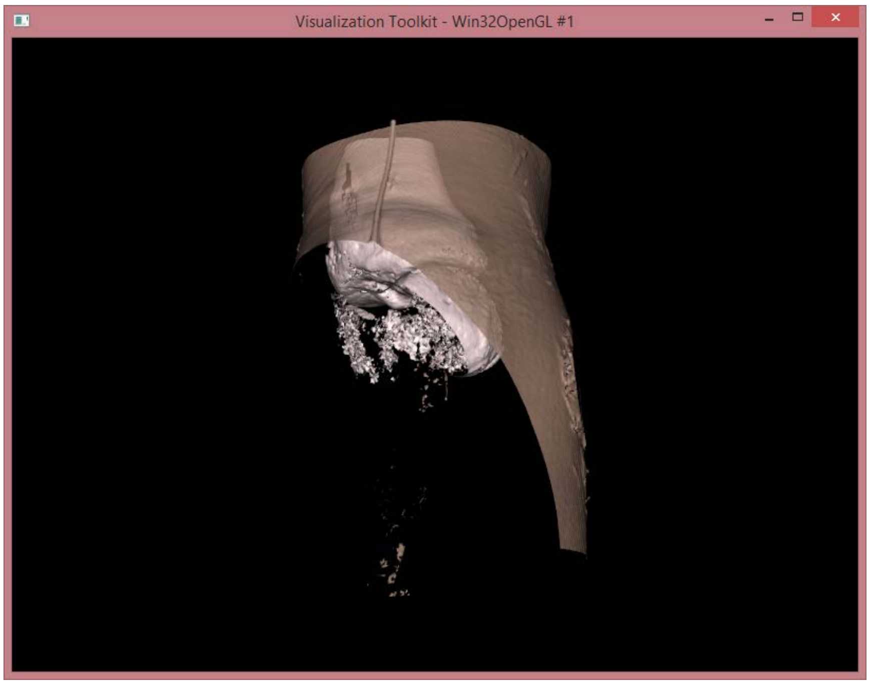
Positioning the sphere

Now that the sphere has been created, the next problem is to position it in the right place and make it the correct size. This is a bit tricky since the sphere is invisible unless it intersects a part of the already created bone geometry. There are several ways of making the sphere visible for easy positioning.

Final Visualization

For your final visualization you may choose to combine opacity, clipping (as described above) and other visualization features presented in lectures. You are required to use what you have learned to provide an enhanced visualization of the available data.

For example:

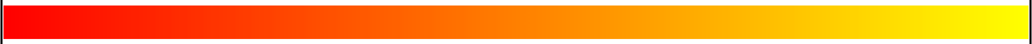
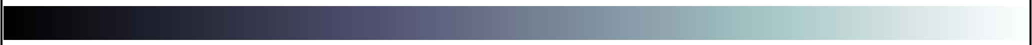
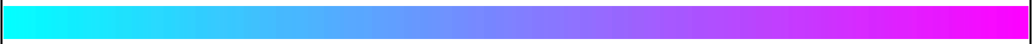
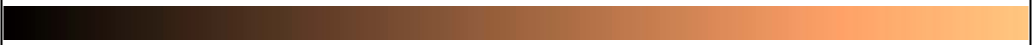
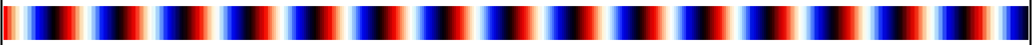
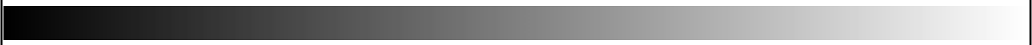
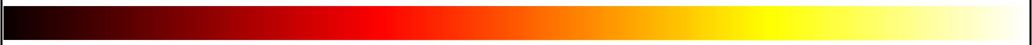
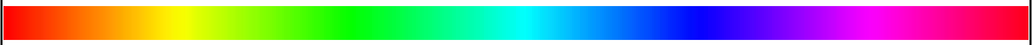
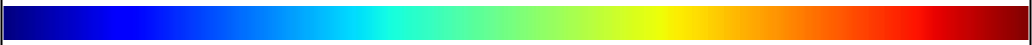
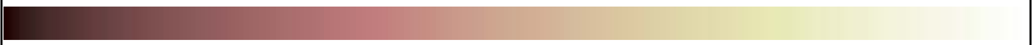


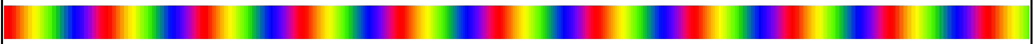
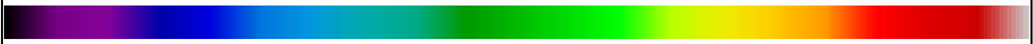
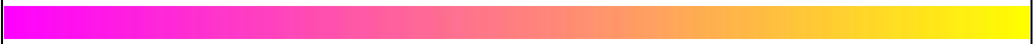
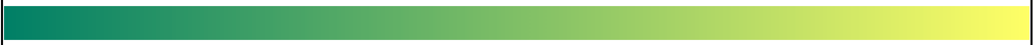
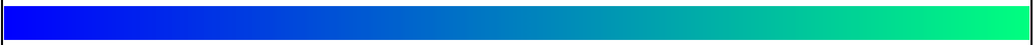
Appendix 1 Colormaps

All colormaps can be reversed by appending `_r`. For instance, `gray_r` is the reverse of `gray`.

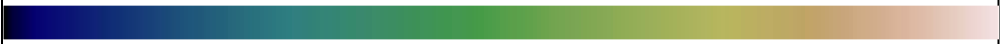
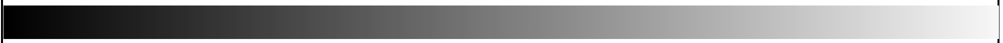
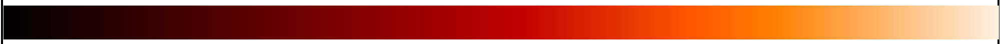
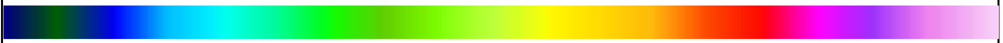
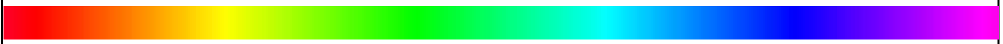
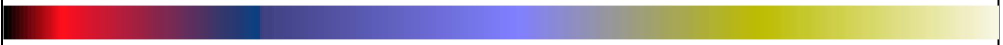
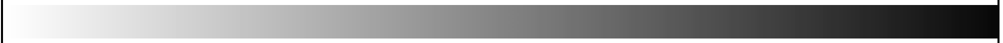
If you want to know more about colormaps, checks Documenting the matplotlib colormaps.

Base

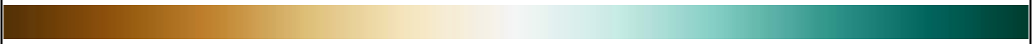
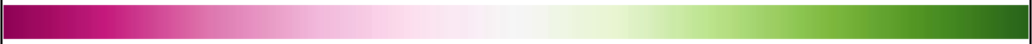
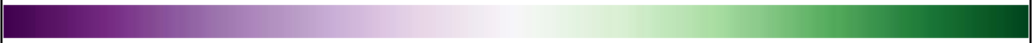
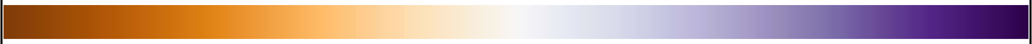
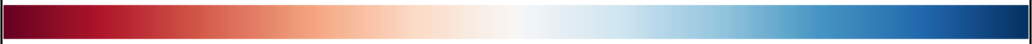
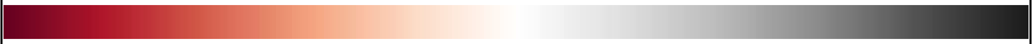
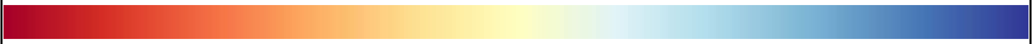
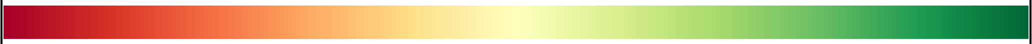
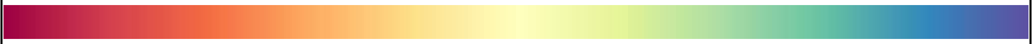
Name	Appearance
autumn	
bone	
cool	
copper	
flag	
gray	
hot	
hsv	
jet	
pink	

Name	Appearance
prism	
spectral	
spring	
summer	
winter	

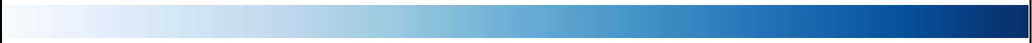
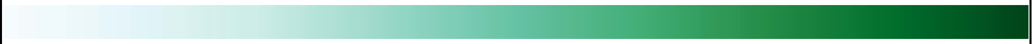
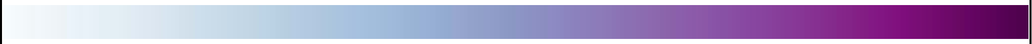
GIST

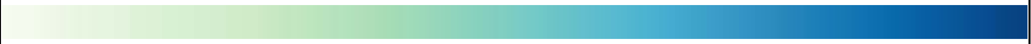
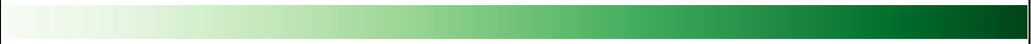
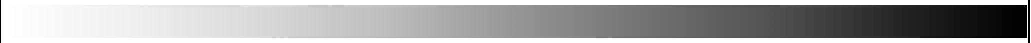
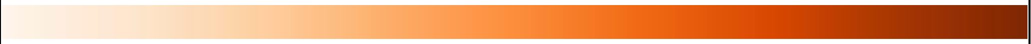
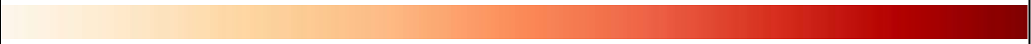
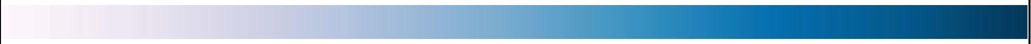
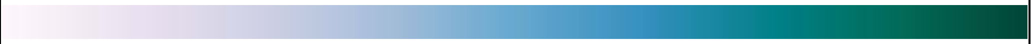
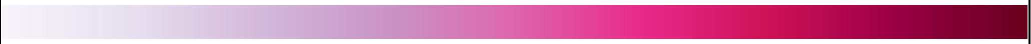
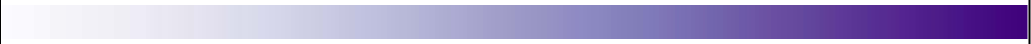
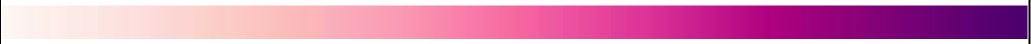
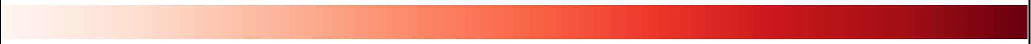
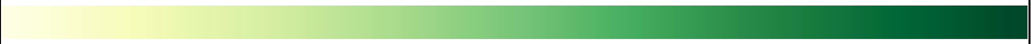
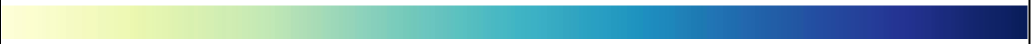
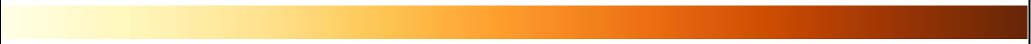
Name	Appearance
gist_earth	
gist_gray	
gist_heat	
gist_ncar	
gist_rainbow	
gist_stern	
gist_yarg	

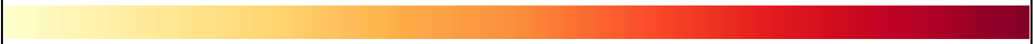
Sequential

Name	Appearance
BrBG	
PiYG	
PRGn	
PuOr	
RdBu	
RdGy	
RdYlBu	
RdYlGn	
Spectral	

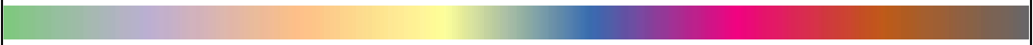
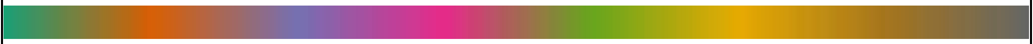
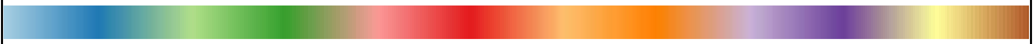
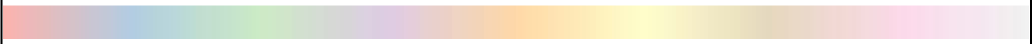
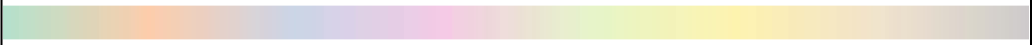
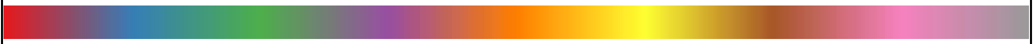
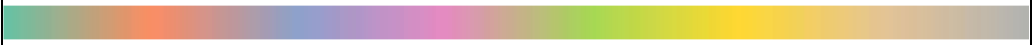
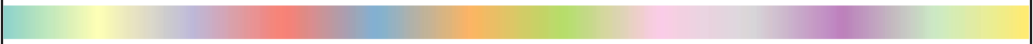
Diverging

Name	Appearance
Blues	
BuGn	
BuPu	

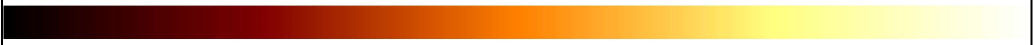
Name	Appearance
GnBu	
Greens	
Greys	
Oranges	
OrRd	
PuBu	
PuBuGn	
PuRd	
Purples	
RdPu	
Reds	
YlGn	
YlGnBu	
YlOrBr	

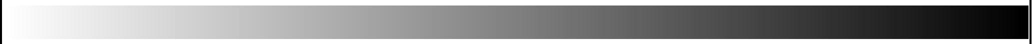
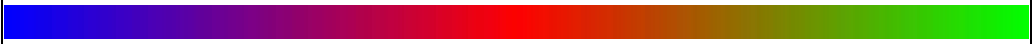
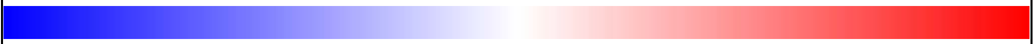
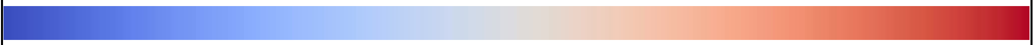
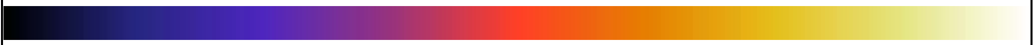
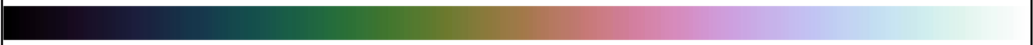
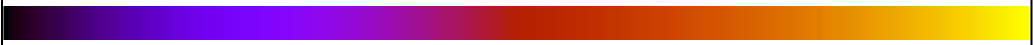
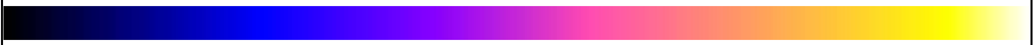
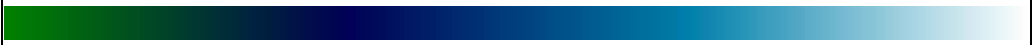
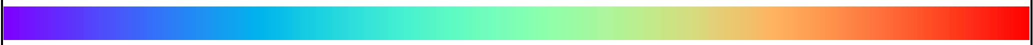
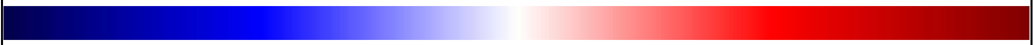
Name	Appearance
YlOrRd	

Qualitative

Name	Appearance
Accent	
Dark2	
Paired	
Pastel1	
Pastel2	
Set1	
Set2	
Set3	

Miscellaneous

Name	Appearance
afmhot	

Name	Appearance
binary	
brg	
bwr	
coolwarm	
CMRmap	
cubehelix	
gnuplot	
gnuplot2	
ocean	
rainbow	
seismic	
terrain