Mathematisch-Naturwissenschaftliche
Fakultät

Fachbereich Informatik
Arbeitsbereich Visual Computing

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Medical Visualization

Prof. Dr.-Ing. Thomas Küstner

WS 2025/2026

Assignment 2

Remarks

The exercises should be completed in groups of three to four students. Please submit your exercises in ILIAS *before* 23:55 on the closing date. At *least* one member of the group *must* be present at the exercise group meeting, being prepared to explain *each* exercise. Random groups will be asked to present their solutions.

Python

- **Please put the answers to comprehension questions into a separate PDF file.**
- **If there is a built-in function for an algorithm you are supposed to implement, do *not* use it.**

VTK

For the programming tasks, Python is used together with the Visualization Toolkit (VTK).
You will find the necessary information about the required classes and functions here:

- Python Examples: **https://examples.vtk.org/site/Python/**
- Reference to all functions: **https://vtk.org/doc/nightly/html/index.html**
  (If you view a class description, be sure to click one the 'More...' right at the top)
- Documentation: **https://docs.vtk.org/en/latest/**

LLMs

We strictly do not allow the use of AI to solve entire exercises. Solutions or code submissions that have been entirely generated by LLMs like ChatGPT or Claude will be graded with 0 points.

Submission

Create a folder **lastname1_..._lastnameN** with your team members last names in alphabetical order. Copy your files into this folder. Hand in all of your solutions and files necessary for the code to run (no need to upload the exercise sheet). Create a zip file **lastname1_..._lastnameN.zip** from the folder. Submit the zip-file to Ilias.

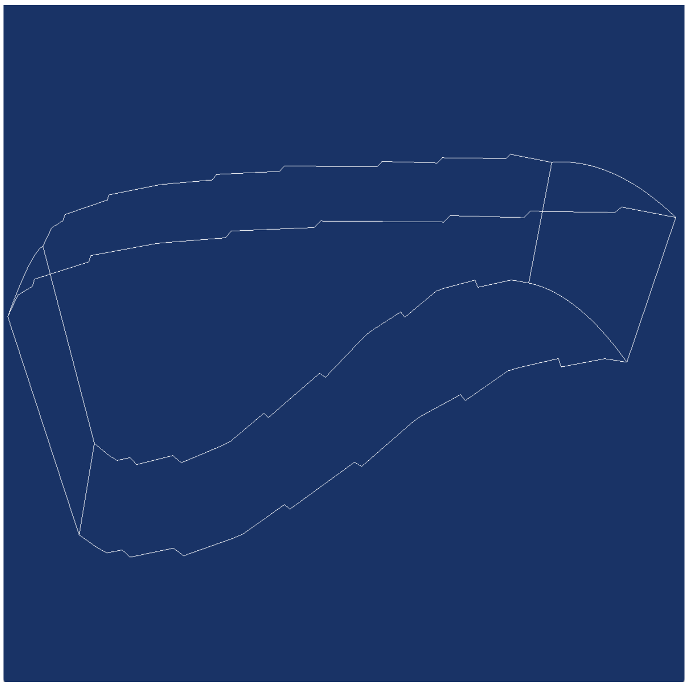
General Information

The purpose of this assignment is to become familiar with different visualization techniques in VTK. The techniques presented here are just a small selection of the possibilities that VTK offers. In order to compare the methods used, the following tasks use the same dataset, which consists of values collected from a running combustion engine. (The environment has not been harmed because the data was already collected in the past.) The goal of the tasks is to implement various visualization techniques. For each task, a file named **Color_Mapper.py**, **Cutter.py**, **Isosurf.py** or **Probe.py** is provided, in which the function **visualize(data_source, renderer)** should be implemented. The different implementations are carried out according to the tasks described below and can be executed by appropriately setting the variable **visualization_mode** in the provided template.

Hint: To position the visualization elements, it is helpful to use the coordinate ranges of the supplied data as a guide. The physical bounds of the supplied data in world coordinates can be obtained using the **GetBounds()** method when called from the **data_source** data object and the data index ranges (i.e. $(i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max})$) of the data array using the **GetExtend()** method.

Exercise 1: Understanding the Code Template (3 points)

For the completion of this assignment, a code template is provided that reads a set of scalar and vector data to be visualized and implements the standard VTK rendering routine. First, the path to the dataset must be entered in the line where the variable **data_root** is defined. The code is then executable and produces the following output:

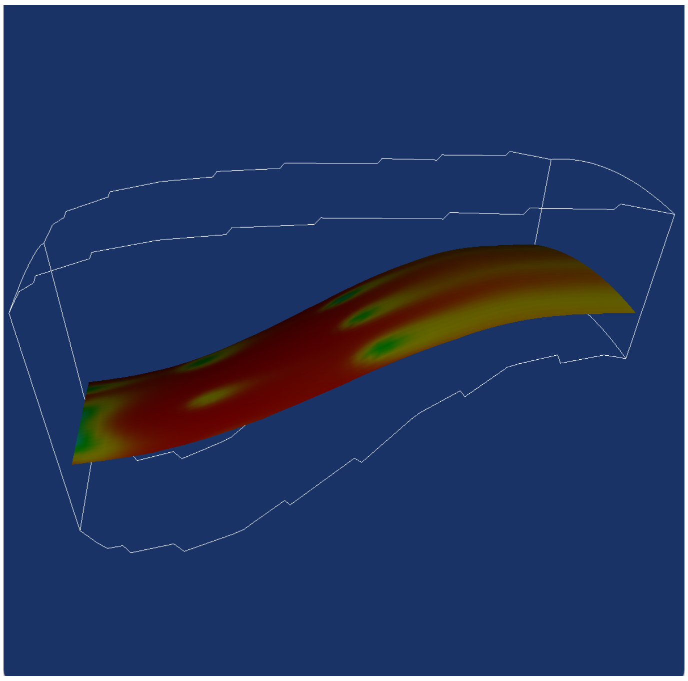


It is important to understand what happens during the data reading process. Consider the first code block and explain what each individual command does (you may do so at the indicated comment locations in the code instead of a separate PDF). The VTK documentation can be helpful here. In particular, the entry for vtkMultiBlockPLOT3DReader is recommended (especially the section **Detailed Description**). It might also help to briefly familiarize yourself with the Plot3D data format.

Exercise 2: Color Mapping (6 points)

Use the class **vtkStructuredGridGeometryFilter** and its functions **SetInputData()** and **SetExtent()** to define a plane that lies within the data of the given dataset. Remember to connect the output of the data to the plane and create a Mapper (**vtkPolyDataMapper**) and Actor.
Create a LookupTable using **vtkLookupTable** that realizes the color mapping. Setup the LookupTable so that the scalar values are automatically colorized with a Hue ranging from red to blue. Don't forget to call the **Build()** function at the end. After that, the LookupTable must be connected to the mapper of the plane. Use the function **SetScalarRange()** of the mapper and pass it the value range of the dataset (which can be obtained using **GetScalarRange()**).

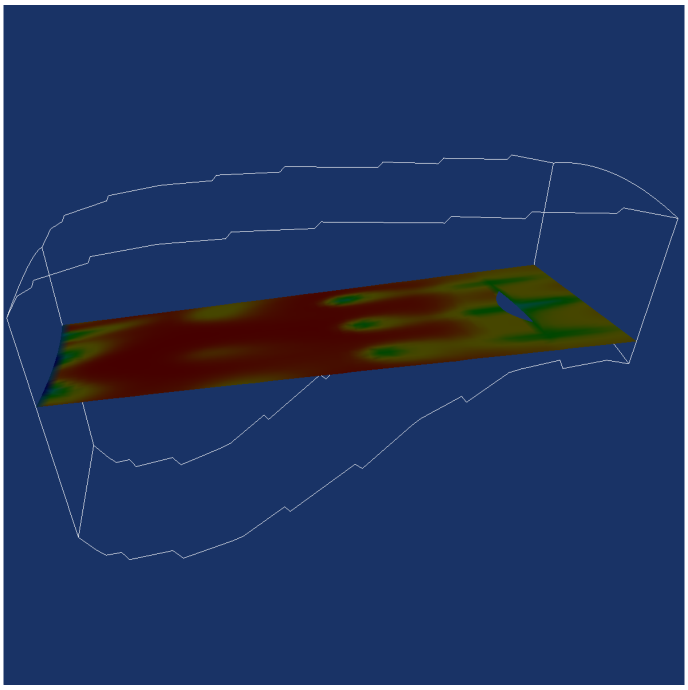
The result should look approximately like the following (it of course depends on the chosen location of the plane and values of the LookupTable):



Exercise 3: Cutting (6 points)

Create a plane within the given data using **vtkPlane**. To position the plane and change its orientation, use the functions **SetOrigin()** and **SetNormal()**.
Instantiate a cutter using the **vtkCutter** class and use its function **SetInputData()** to connect the output of the dataset to the cutter. Use **SetCutFunction()** to specify the plane as the cut function. Also create a LookupTable like in exercise 2 and connect it to the mapper.
Similar to the previous task, the data from the cutter must again be passed to a mapper and subsequently to an actor.

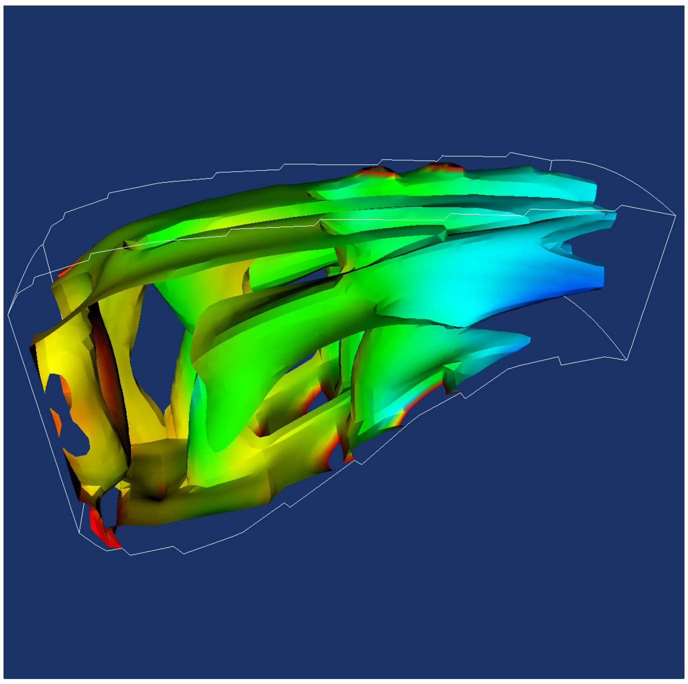
A possible result looks like the following (again depends on the position of the cutting plane and LookupTable):



Exercise 4: Color Isosurface (6 points)

Use the class **vtkContourFilter** to create an isosurface (surface of identical values) within the dataset. Its functions **SetInputData()** and **SetValue()** are particularly important here. It helps to understand the value ranges of the scalar data beforehand using **GetScalarRange()** again.
Remember to create a mapper and actor afterward and connect the mapper to the output of the **vtkContourFilter** instance.
We can now for example also use some of the other scalar data stored in the data source to colorize the isosurfaces. Otherwise we would have to use the same color for the entire isosurface, since by definition the surface visualizes data points with identical scalar values.
Adjust the mapper using the function **ColorByArrayComponent()** to use the "VelocityMagnitude" scalar data for colorization.
Also use the **SetScalarModeToUsePointFieldData()** function. Additionally, the correct scalar range has to be set again using **SetScalarRange()**. Since we are colorizing using the velocity magnitudes instead, the correct range has to be determined from the velocity magnitude data array. It can be accessed using
data_source.GetPointData().GetArray("VelocityMagnitude").

The result can look approximately like the following (the surface can look completely different depending on the set isosurface value):



Exercise 5: Probing (9 points)

Use the class **vtkPlaneSource** to generate a surface piece of a plane. With the classes **vtkTransform** and **vtkTransformPolyDataFilter**, it can be translated, scaled (which determines the size of the surface piece), and rotated.
Using these transformations, three surface pieces should be generated, and for each, a **vtkOutlineFilter** should be used to display their respective edges. Each **vtkOutlineFilter** must be connected to an appropriate mapper and actor. The color of the edge can be set using the function **GetProperty().SetColor(r, g, b)** of the actor.
Using the class **vtkAppendPolyData** the three planes can be combined to a single poly data object. Using **vtkProbeFilter**, a probe geometry consisting of the three planes can then be created within the dataset. Don't forget to also set the source data for the probe filter so that the filter knows which scalar data to probe using the planes.
The output of the probe geometry is finally connected to a **vtkContourFilter()** which can be used to automatically visualize a number of contours within the probe geometry. The contour filter, in turn, must again be connected to a mapper and actor for visualization. The renderer must be provided with all created actors at the end, including those of the surface pieces. Otherwise, their edges will not be drawn.

The result could look approximately like the following (depending on the location of the three planes):