Network-integrated Pre and Post-Processing in Scientific Computing
M2 CSMI - S3 / M1 IIRVIJ - S1 / 2024–2025
Travaux Pratiques

An introduction to VTK - scalar field visualization - iso-surface extraction

Introduction

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization, and 2D plotting. It supports a wide variety of visualization algorithms and advanced modeling techniques, and it takes advantage of both threaded and distributed memory parallel processing for speed and scalability, respectively.

VTK is designed to be platform agnostic. This means that it runs just about anywhere, including on Linux, Windows, and Mac; on the Web; and on mobile devices.

- Website : https://vtk.org/
- Full documentation : https://docs.vtk.org/en/latest/index.html
- C++ API documentation : https://vtk.org/doc/nightly/html/

Exercice 1 : Setup & test VTK

Setup: to use vtk and its C++ API you will need:

- VTK : install the `libvtk9-dev` package
- Cmake : install the `cmake` package
- An editor (your favorite one)

Test: you can now test your configuration with the *cylinderExemple*.

- In *cylinderExemple/*, create a new directory named *build*
- From this *build* directory, use the `cmake` command to execute the *CMakeLists.txt* at the root of the *cylinderExemple/*.
- If everything is right, you can now compile your code with the `make` command.
- Run the program

If everything works, congratulations, you've just run your first VTK program! You will now be able to code it yourself in the next exercices.

Exercice 2 : Load a dataset and render it

The code skeleton provided with the exercise package is composed of 3 main classes:

- Editor: This class stores the actual data-structures and drives the computations;
- UserInterface: This class is in charge of the rendering management and of the processing of the user interactions;
- IsoSurfacer: This class is right now a place-holder for your own custom isosurfacer implementation.

Note: VTK uses its own memory allocation and management system. This means that VTK objects cannot be declared directly. Instead, pointers to VTK objects must be declared as follows:
`vtkRenderer *renderer_ = NULL;`

Later, the memory for this object needs to be dynamically allocated with the following instruction:
`renderer_ = vtkRenderer::New();`

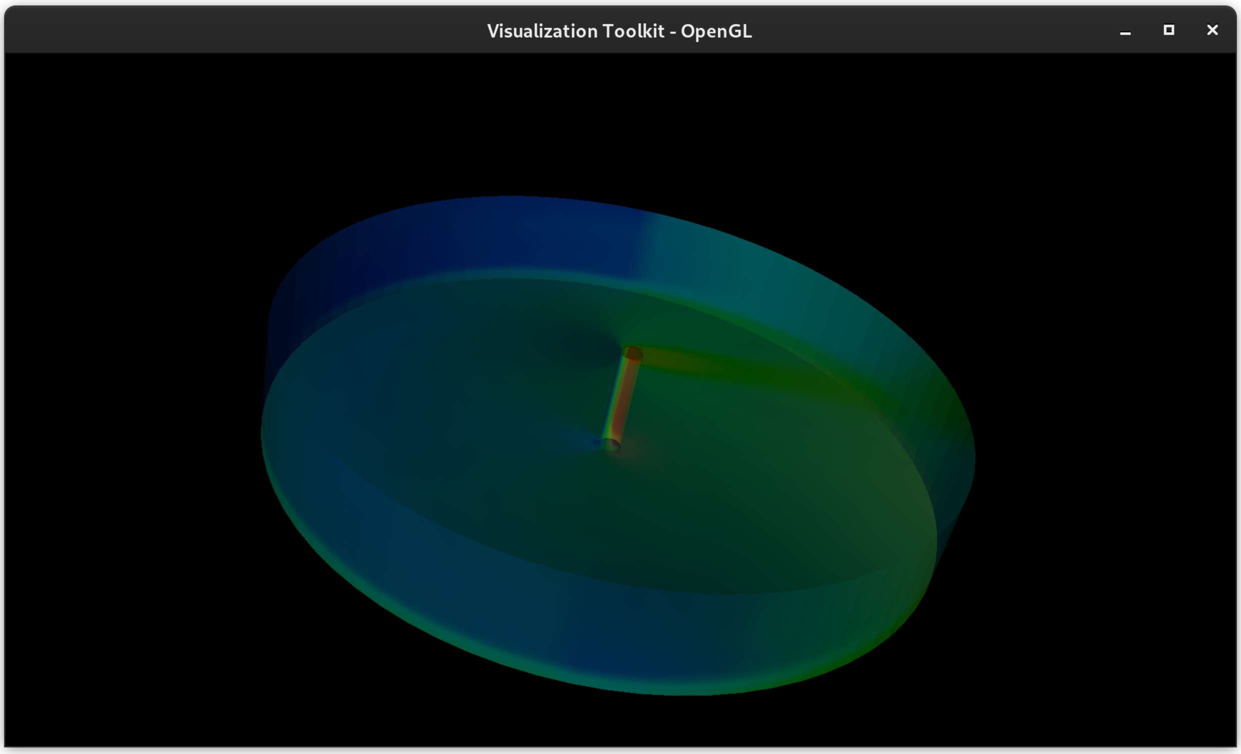
The above lines declare a pointer to a vtkRenderer object (named `renderer_`) and allocate the necessary memory for it (with the call to the function `New()`).

Complete this code :

- From the main, find where the mesh is loaded. Complete the function to load an unstructured grid from a file. Set the scalar data of the input mesh with its first array of scalar data.
- From the main, find where the rendering is trigger. Complete the function to render the mesh in a window, interact with it and visualize its boundary with a color map and transparency :
  - `vtkRenderWindow` (to create a graphical window)
  - `vtkRenderer` (to perform the actual rendering tasks within the graphical window);
  - `vtkDataSetMapper` (to convert our triangulation into a set of graphics primitives that are "renderable");
  - `vtkActor` (to represent and interact with our triangulation in our visualization).

  - Link the triangulation to the `vtkDataSetMapper` object with its function `SetInputData`() (see the class documentation).
  - Link the mapper to the `vtkActor` object with its function `SetMapper()` (see the class documentation).
  - Link the actor to the `vtkRenderer` object with its function `AddActor()` (see the class documentation).
  - Link the renderer to the `vtkRenderWindow` object with its function `AddRenderer()` (see the class documentation).

Compile it with cmake and make. Execute it with the *data/post.vtu* dataset using the '-*t*' argument. It is a simulation of liquid oxygen diffusion. You should obtain this :



Exercice 3 : Iso-surface extraction with VTK

First, we will review iso-surface extraction using VTK. To do this, we will use the VTK notion of Filter. In VTK, a filter is a standardized processing unit that accepts data on its input (SetInput()) and delivers data on its output (GetOutput()) after it has been triggered for execution (various parameters can also be defined to tune the filter). In VTK, filters can be plugged to each other beforehand to form a Pipeline. Hence, a visualization program can be quickly put together by simply connecting filters together and running the pipeline. To trigger the execution of a filter independently of a pipeline, its function Update() should be called.

1. :

   Once the skeleton program is running, hitting the key 'a' will trigger the code for iso-surface extraction with VTK. In the UserInterface class, find out where this event is captured.

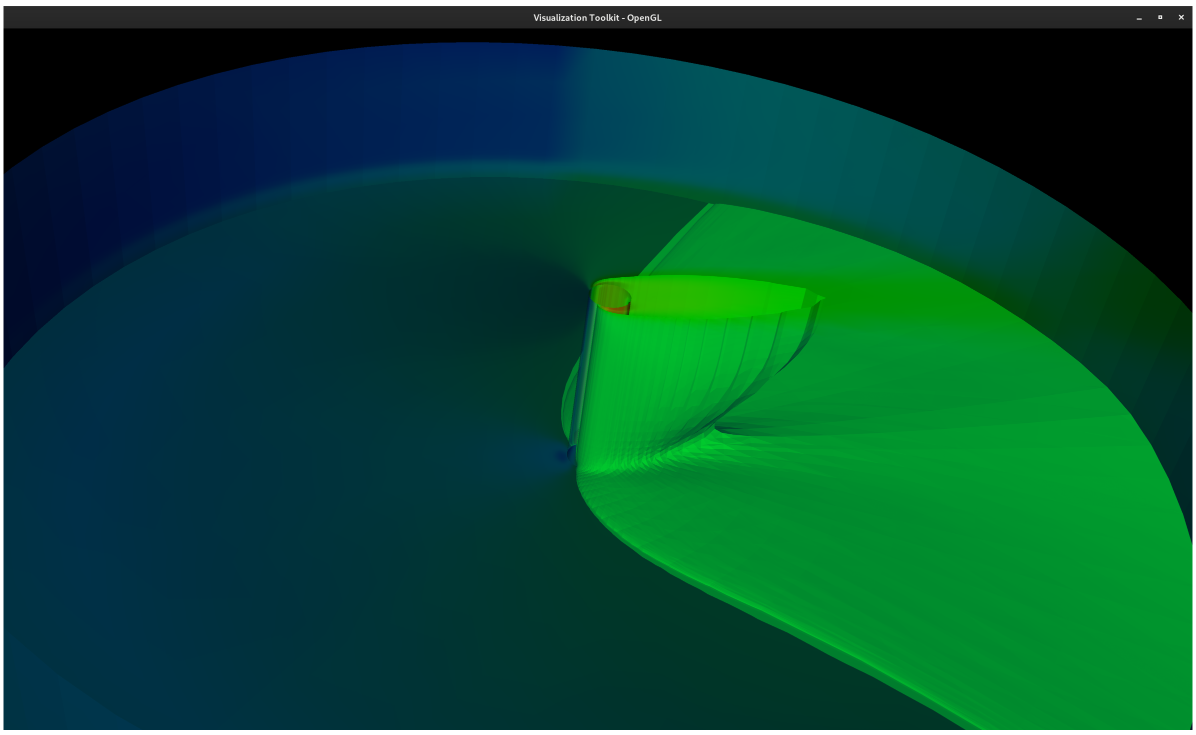
2. :

   From this point (capture of the 'a' key event), follow the function calls. Find out which function of the Editor class is in charge of the iso-surface extraction with VTK.

3. :

   Complete the function identified in the previous question to extract an iso-surface with VTK. Be careful and pay attention to what this function returns and how its returned value is employed by the rest of the code skeleton (don't hesitate to browse the VTK documentation).

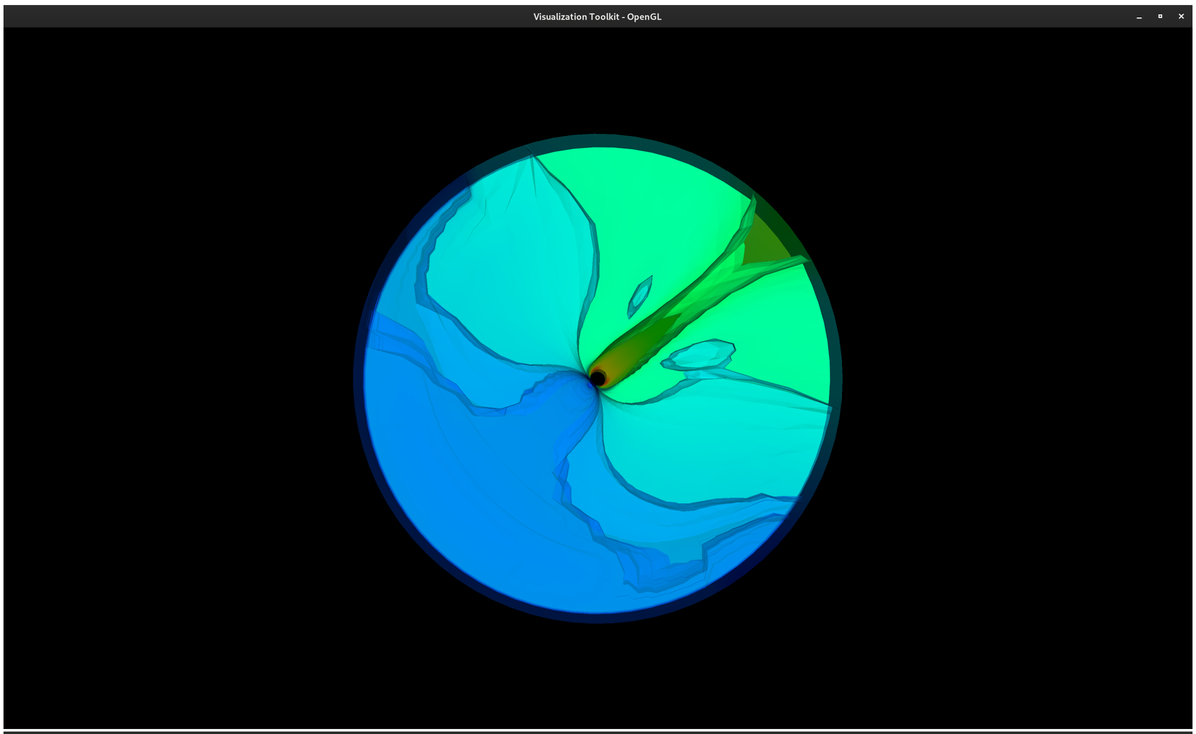
   If you correctly answered to the previous questions, you should now see the following result by pressing the key 'a':

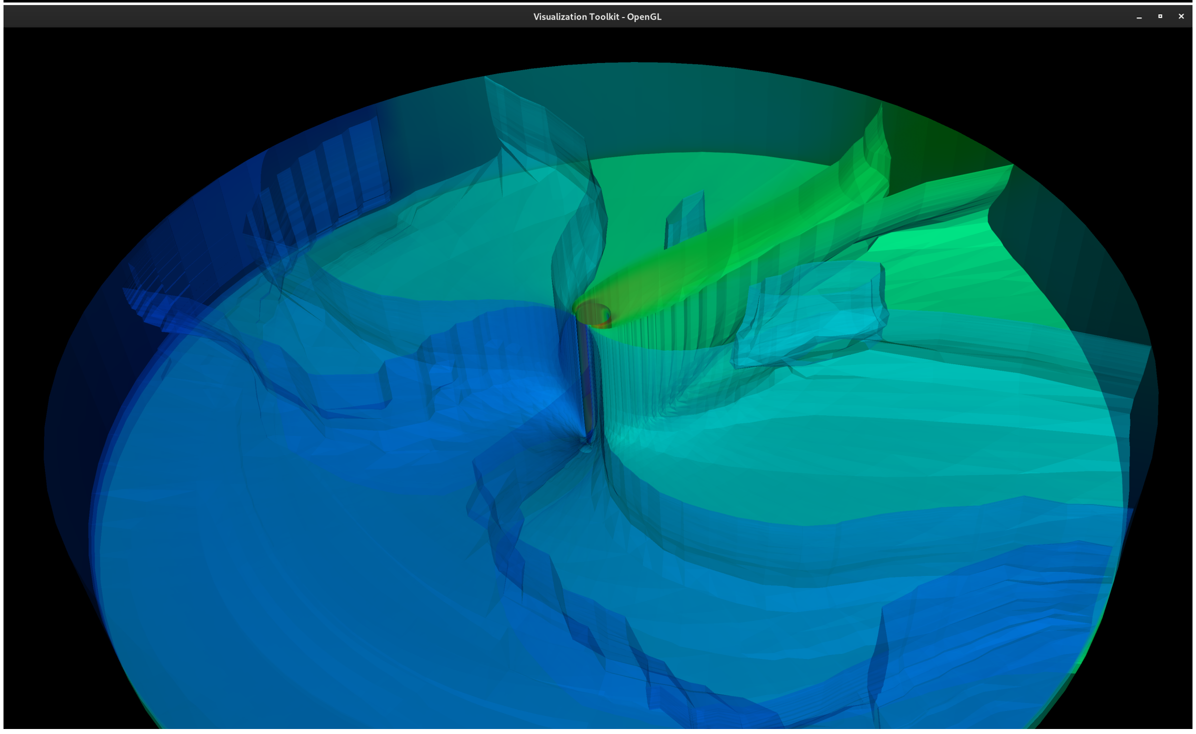


4. :

   Go back to the function of the UserInterface class capturing the key events (question 1). For each keystroke, follow the corresponding function calls in order to understand the purpose of each keystroke.

   Then, try to reproduce the following visualization:





5. :

   Extract an isosurface for the isovalue 57.5 (on the post.vtu dataset). Then move it up to isovalue 58. The isosurface changed its topology. What kind of topological change is it?

   Scan the entire function range to see if these events occur in other places.

Exercice 4 : Implementing your own isosurfacer

In the rest of the exercise, we'll move on to the implementation of your own custom isosurfacer. To proceed, we'll implement a VTK filter. You will find an initial skeleton for this filter in the IsoSurfacer class. When running the program, to switch back and forth between the VTK implementation and your custom implementation, hit the key v. The rest of the keystrokes work as before.

For the following questions, make sure you always execute your program such that the custom implementation is selected (by hitting v once).

In the remainder of the exercise, we will implement an isosurface extraction algorithm in the class IsoSurfacer.

1. :

   The naive algorithm for isosurface extraction consists in:

   - looping over the entire list of tetrahedra of the input tet-mesh, and for each tetrahedron:
   - computing, if it exists, the polygon which intersects the tetrahedron along the level set;
   - adding the vertices and the faces of that polygon to the output isosurface.

   Have a look at IsoSurfacer.h to see how to access the input tet-mesh, the input scalar field and the output surface from within the IsoSurfacer class. The corresponding variables are already initialized for you in the code skeleton.

   To implement this algorithm, we will proceed as follows:

   - We will implement a function IsCellOnLevelSet() that will check if a cell (or simplex) is intersected by the queried level set;
   - We will implement a function ComputeEdgeIntersection() that will compute for a given edge its intersection point with the queried level set;
   - We will implement a function ComputerSimpleIntersection(), that will apply for a given tetrahedron the above two functions to generate the polygon which intersects the input tetrahedron along the queried level-set;
   - We will implement a function SimpleExtraction(), that loops over the tetrahedra of the input mesh and apply the above function.

2. :

   In IsoSurfacer.h, complete the function IsCellOnLevelSet(). This function returns true if the cell is traversed by the level set and false otherwise.

   Warning, this function takes as input the abstract notion of vtkCell, which is dimension independent. In other words, this function should work if either an edge or a tetrahedron is given as argument.

   Don't hesitate to have a look at the VTK documentation to understand how to retrieve the vertex Ids of a cell and to retrieve the corresponding scalar values from the scalar field given the vertex Ids.

3. :

   In IsoSurfacer.h, complete the function ComputeEdgeIntersection(). Given an input edge expressed as a pair of vertex Ids, this function returns the point where the level set intersects the edge.

4. :

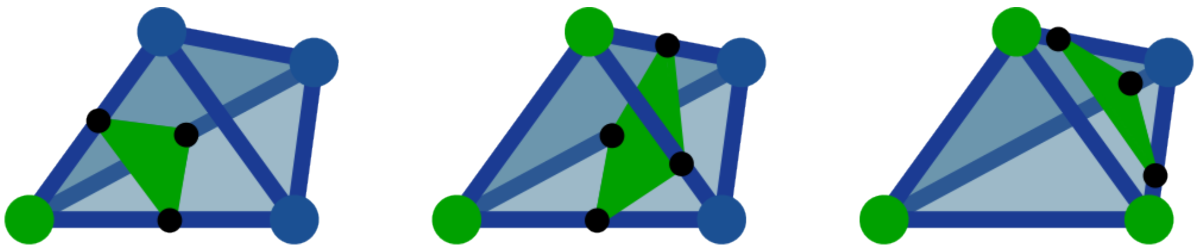
   Complete the function IsoSurfacer::ComputeSimpleIntersection(). Given a tetrahedron given as argument, this function computes the polygon which intersects the tetrahedron along the level set and add it to the output isosurface.

   In particular, this function should loop over all the edges of the tetrahedron. If the current edge is intersected by the level set (IsCellOnLevelSet()), its intersection point with the level set should be computed (ComputeEdgeIntersection()).

   Once an intersection point is computed, to create the corresponding vertex in the output isosurface, use the following call (see the VTK documentation for further details): Output->GetPoints()->InsertNextPoint().

   This function will return the vertex Id of the newly created vertex. Finally, to create the intersection polygon, use the following code: Output->InsertNextCell(VTK_POLYGON, pointIds) where pointIds is a vtkIdList that you'll have to create to store the vertex Ids of the new vertices you will have created.

   As a reminder, the possible intersections of a tetrahedron by a level set are summarized in the following figure:



5. :

   Complete the function IsoSurfacer::SimpleExtraction(), which loops over the set of tetrahedra of the input mesh. If a tetrahedron is intersected by the level set (IsCellOnLevelSet()), then the actual intersection polygon should be computed (ComputeSimpleIntersection()).

6. :

   Execute your code and test to extract an isosurface with your own implementation. This doesn't look great, right? What do you think went wrong?

7. :

   To answer and correct the above question, complete the function IsoSurfacer::ReOrderTetEdges(), which given the list of the edges of a tetrahedron which are crossed by the level set, re-order this list such that the list describes a consistent winding around the tetrahedron.

   Next, use this function appropriately from within the function IsoSurfacer::ComputeSimpleIntersection().

   If you got the answer right to this question, your isosurface should no longer exhibit the cracks you've got in the previous question.

8. :

   For this question, make sure you're compiling your code in Release mode instead of Debug (in CMakeLists.txt).

   Now compare the timings, for a unique isovalue, between your custom implementation and the VTK implementation (by hitting v to switch between the two).

   Why is your code so slow?!!

Acknowledgments These exercices are inspired and adapted from an exercise originally written by Julien Tierny. Thanks Julien!