Assignment: 3D viewer

1 Introduction

During this assignment, you will gather the content of multiple DICOM files and provide an interface allowing to easily navigate and display between the different slices of the acquisition.

It is requested that you start from the provided solution to the first assignment. It is strongly advised to take a look at the provided solution and to create new classes for this assignment.

The final solution should contain multiple features which can be enabled or disabled. Start by reading the entire document in order to have a better idea of the required modularity.

All the examples shown in this document are only possible solutions, you should not necessarily try to match the interface but rather think about the functionalities. If you find more ergonomic solutions, feel free to implement them. At the end of the assignment you should obtain an interface roughly similar to the one presented in Figure 1.

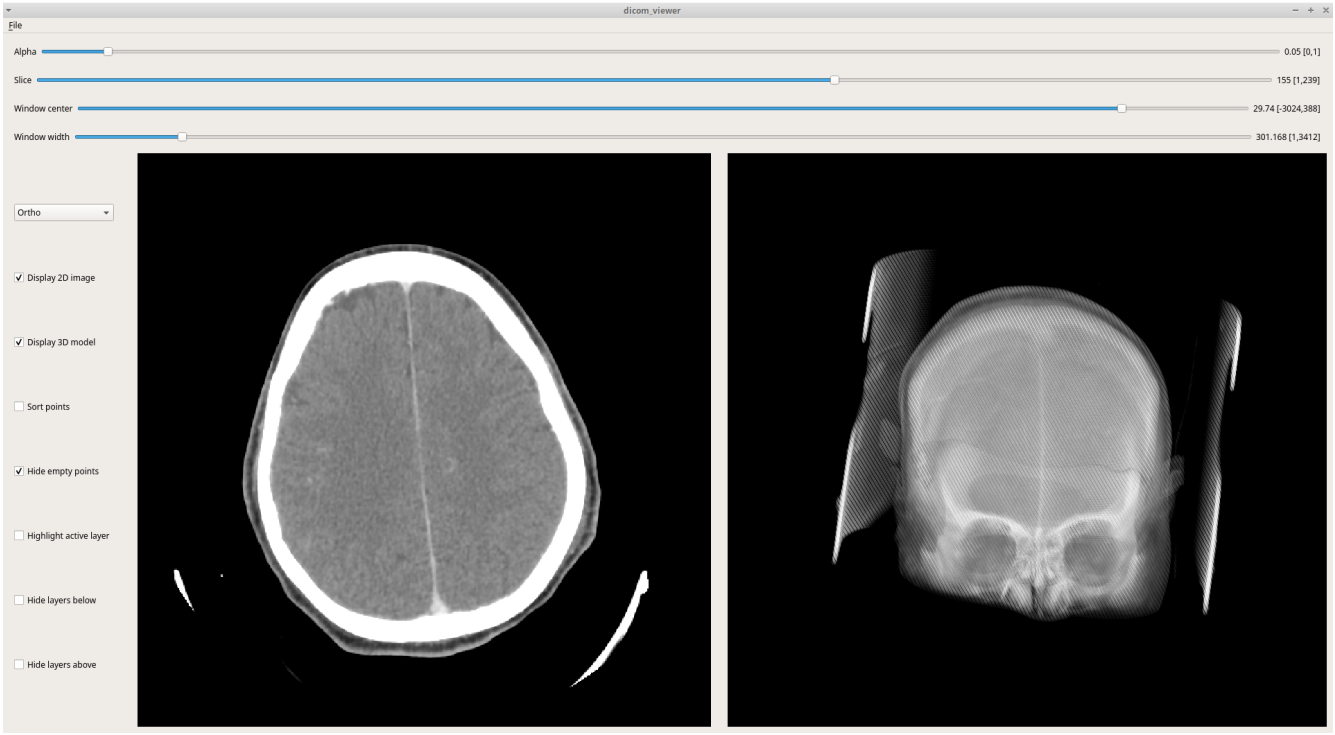

Figure 1: Expected interface at end of assignment

Some datasets only contains a limited number of slices. Make sure to use examples with more than hundred slices to have better appearances and to test the performances.

2 Navigating through slices

This assignment is a major evolution of the initial `DicomViewer` created during previous assignment. Therefore it is quite important when changing a feature to think about how to maintain previous functionalities, but also to think which functionalities make no sense anymore.

2.1 Opening multiple files

The first step to navigate in the 3D acquisitions is to load and store several `DcmFileFormat` in the `DicomViewer`. A simple solution is to use `getOpenFileNames` which returns a list of paths instead of `getOpenFileName` which allows to select a single file. Replacing the current solution is likely to break many of the previous features. A simple solution to ensure compatibility at this point is to consider that the first element opened is the active file.

When opening multiple files at once, make sure to check that they can be considered as a correct and complete collection. A few potential issues you might cross are the following:

- Files are not from the same patient.
- A file has been duplicated.
- Some files are missing.

You might think of other issues while developing other functionalities, be sure to **at least** display a warning if some of the content is unexpected.

2.2 Navigation slider

Now that all the files are loaded, the next goal is to change dynamically the file shown in the `QLabel`. This can be achieved using a `QSlider`, in order to have an intuitive interface, the name of the slider and its value should be shown. You can create a new `QWidget` which encapsulates several elements as shown with the class `DoubleSlider` from the provided solution.

This slider should only be shown when more than one file is loaded.

2.3 Adjusting existing tools

Some of the existing tools need to be adapted with the changes you just made, a very good example is the `showStats` method which display a message with basic information about the content loaded. Modify it to add some global features such as the number of slices. For some of the existing information, it is interesting to have access to two different versions, one for the frame and another from the collection. Write two methods:

- `getFrameMinMax`: returns the extremum values used in the active image.
- `getCollectionMinMax`: returns the extremum values used among all files.

Add their content to the `showStats` information.

3 3D Viewer

The next step is to display the content of all the files in a 3D environment. In order to do this, you will use the `OpenGL` library and integrate it inside your `Qt` window.

3.1 Generating 3D data

Create a class to contain your 3D row data as a 3-dimensional array. Then write a method to extract the images contained in all the `DcmFileFormat` opened based on `getOutputData`, using the current window.

3.2 A basic OpenGLWidget

In order to display the point cloud, create a new class with `QOpenGLWidget` as parent class.

3.2.1 Creating a display zone

Add an instance of your new class to your `DicomViewer`, it should appear as a black area near the 2D image. If you are using a `QVBoxLayout` or a `QHBoxLayout` for your interface, consider switching to a `QGridLayout` to make a better usage of the available space.

3.2.2 Draw the point cloud

In order to draw all the data points in the OpenGL window, you have to edit the `initializeGL` and `paintGL` functions. Be sure to properly place the setup of the camera in the `GL_PROJECTION` part and the description of your data in `GL_MODELVIEW`.

It is advised to start by converting your volumic data to a vector of points with associated properties such as their color and their layer. To ensure you have a satisfying initial point of view with any kind of data, a good practice is to center the object in the space and to rescale it ensuring it fits in a $[-1, 1]^3$ cube.

When using a narrow window, the volume contains a lot of empty pixels with a value of zero, add a `boolean` flag `hide_empty_points` to your `QOpenGLWidget`. When enabled, it should hide all the points with a value of 0.

Since there is a high density of points, the visualization does not allow to see easily what is inside the head, add an alpha component to your points in order to allow more transparency. Create a new `DoubleSlider` to change dynamically the value of the alpha channel between 0 and 1.

3.2.3 Camera management

Implement a minimal navigation system allowing to control the camera with the following features:

- `left_click + mouse_motion`: rotate the camera around the center of the object
- `mouse_wheel`: zoom in and zoom out of the object.

Make sure to have a system where you can easily access the transformation matrices by storing them with `QMatrix4x4`. You can also experiment different methods of projection using `QMatrix4x4::perspective`, `QMatrix4x4::ortho` and `QMatrix4x4::frustum`.

While implementing your simple camera manager, make sure that changing the size of the widget does not stretch the display. The rotation of the camera should also be applied in the camera referential in order to avoid risks of gimbal lock and to provide a natural behavior.

3.3 Customizable options

Now that you can simply view your object and move your camera around it, add more control by adding the option to change dynamically:

- The value of the `hide_empty_points` variable.
- The type of camera that should be used.

3.4 Highlighting the active layer

Currently, changing the active layer only affects the 2D image. Add the following options that can be changed dynamically:

Highlight active layer: When this option is enabled, all the points from the active layer are drawn without transparency.

Hide layers below: When enabled all points of the layers below are hiden.

Hide layers above: When enabled all points of the layers above are hiden.

The combination of those options should allow to view slices in context, see Figure 2.

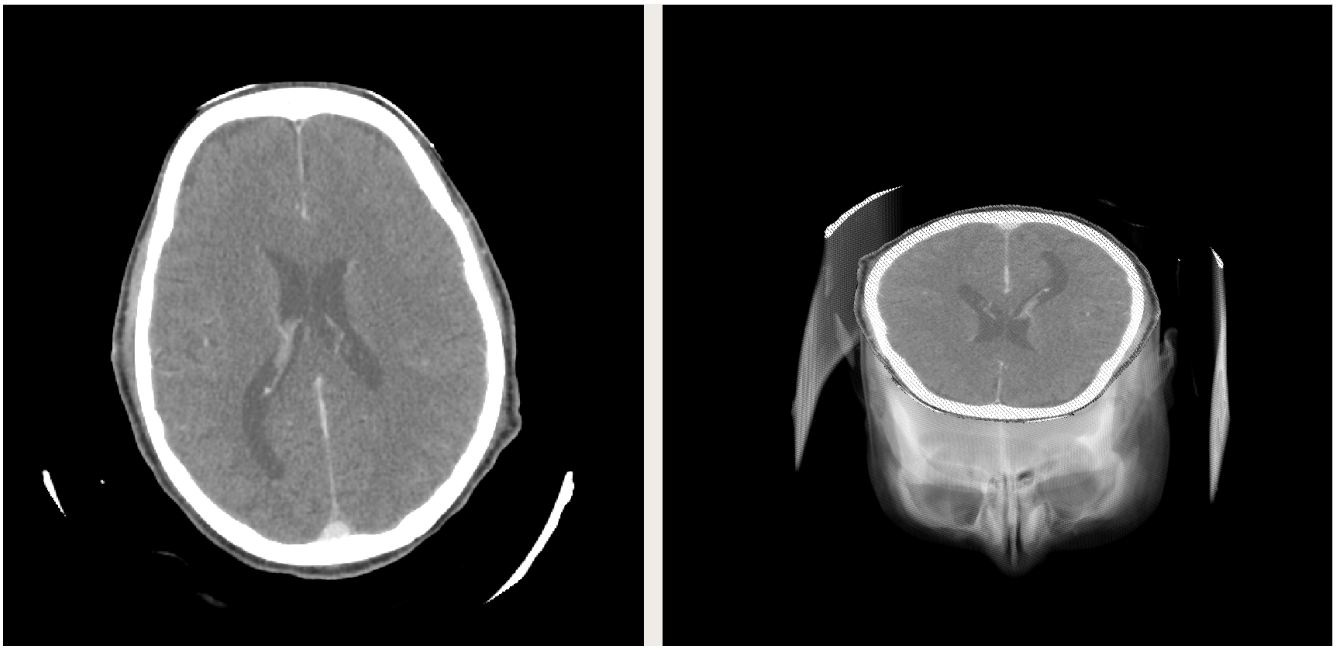

Figure 2: Display with highlighted active layer and layers above hidden.

3.5 Choosing the widgets to show

While having simultaneously the 2D image and the 3D representation is nice, sometimes the user wants to focus on a single one. Add the option to change dynamically the following options:

- Hide/show the 2D image
- Hide/show the 3D representation

When only one widget is shown, it should grow appropriately and take the available space.

3.6 Depth-sorting (optional)

When using `OpenGL` to draw elements with transparency, the order of drawing points or faces can create artefacts, see Figure 3. To solve this, one of the option is to sort the different elements and to show the furthest ones first. Add an option allowing to sort the points by distance to the camera before drawing them. On large files with millions of points, the animation should not be fluid at all anymore.

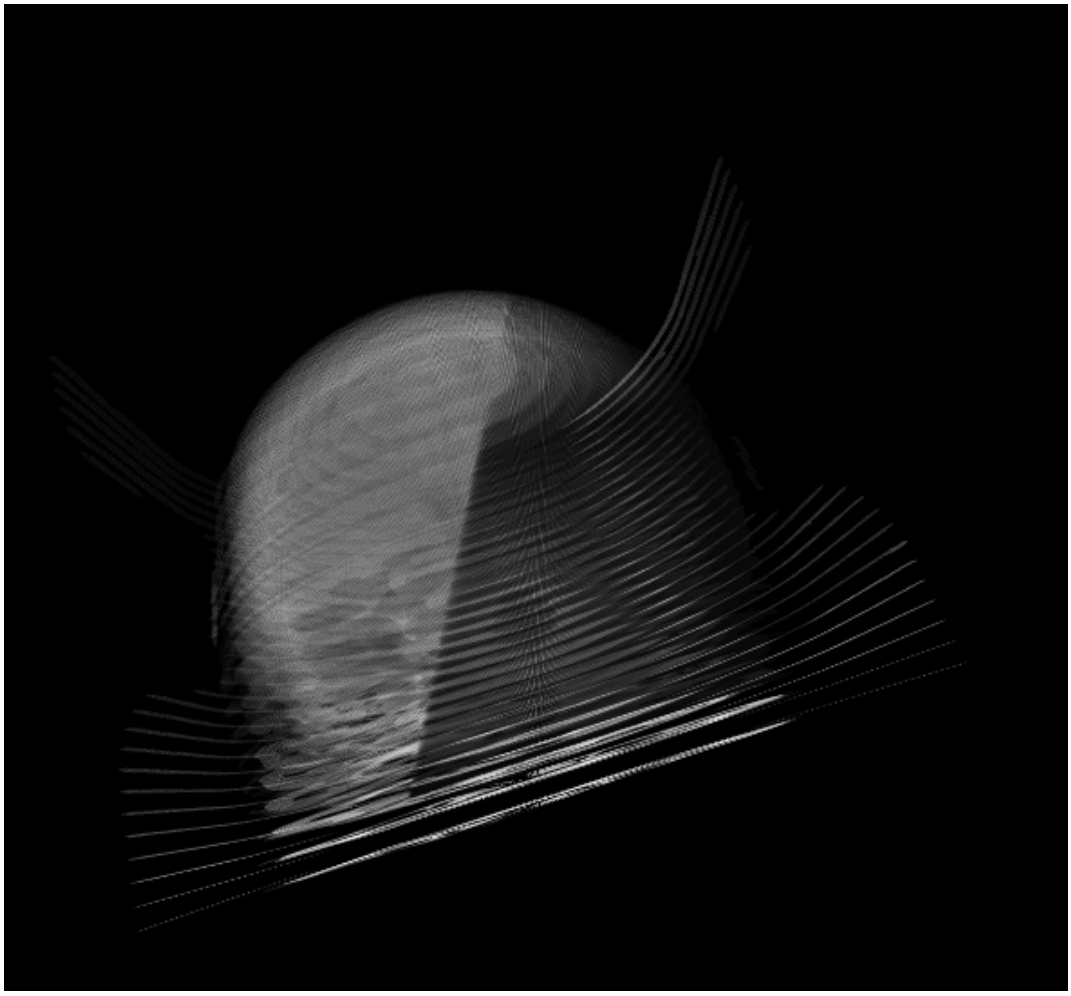

Figure 3: Depth-sorting artefact due to order of drawing

3.7 Optimization (optional)

As you have experimented when using depth-sorting, performance quickly become an important problem when drawing millions of points. Of course, there are many ways to reduce the computation time, but keep in mind that optimizing code often leads to less flexibility and for now you need to be able to make your code evolve quickly.

A simple element that you can use to improve reactivity without reducing the flexibility is to avoid unnecessary heavy computations. An example is what happens when the slider controlling the transparency level is moved: only the drawing should be updated, there should not be any update of the points positions and `getOutputData` should definitely not be called.

Among the global properties expected for your program, we expect that:

- When there is no interaction from the user, the CPU usage should be really low.
- There should not be memory leaks when interacting with the software.

3.8 Exporting the 3D view (optional)

When saving an image, let the user choose whether the 2D image or the 3D representation should be used.