Software for 4D data analysis

Marc-Olivier Andrez

17 July 2010

Contents

1 Introduction

In medicine, patient diseases can often be understood by images of organs. It can also be very interesting to know the ground structure before drilling a well. In both cases, image acquisition of the 3D objects (body, ground, etc.) can be achieved by tomography (http://en.wikipedia.org/wiki/Tomography). Resulting data are $f(x, y, z)$ 4D data, with isosurfaces (f values constant) corresponding to objects having identical structures.

In this document, I will present a program that I have developed: this program is creating isosurfaces from 4D data sets, using the marching cubes algorithm. The input of this program can be a hard coded set of 4D data ($f(x, y, z) = x^2 + y^2 + z^2$) or DICOM files. The isosurfaces are created using the marching cubes algorithm and then displayed in a simple openGL scene renderer.

2 The marching cubes algorithm

2.1 Input data: DICOM files

DICOM files are dedicated to medicine. They are containing images and information about the patient, the way the data have been acquired, etc. My program only deal with the images data. They are composed of slices of 2D pictures having identical sizes ; the $f(x, y, z)$ values are defined on a (x, y, z) grid:

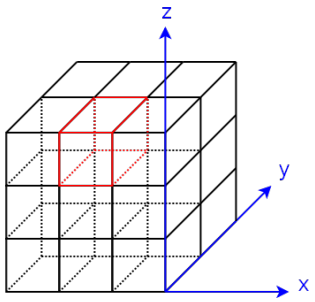

Figure 1: DICOM domain of definition

A first method to create isosurfaces for a (x, y, z) set of points consists in creating a mesh composed of tetrahedrons and then, for each tetrahedron, creating the intersection between the isosurface and the tetrahedron.

For DICOM files, the mesh is already known: it is a $x \times y \times z$ grid. Storing every (x, y, z) point is not needed, only the f value must be in memory. This is saving a lot of memory: the total number of points can quickly become big. For example, for a $512 \times 512 \times 100$ grid, there will be 26.214.400 f values, i.e. about 200 MBytes if a value is coded by a "double" C++ object. If every (x, y, z) point were stored, $3 \times 26.214.400$ values would be kept, that is to say $3 \times 200 = 600$ MBytes. However, only the $512x$, $512y$ and $100z$ values are needed and this is sparing a lot of memory (600 MBytes).

2.2 Why using the marching cubes algorithm?

Computing a tetrahedron mesh is time consuming. More important, it can decrease the results precision. Here is a 2D example to understand the potential precision problems introduced by a created mesh:

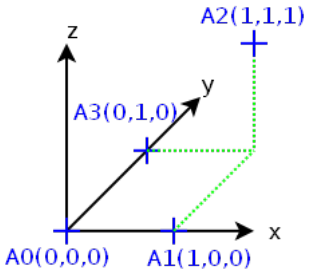

Figure 2: 2D square

There are two ways to split the square $A0, A1, A2, A3$ into two triangles (red lines):

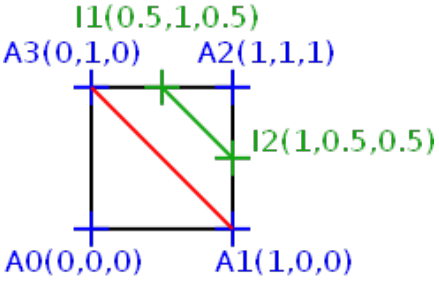

Figure 3: $(A1, A3)$ split

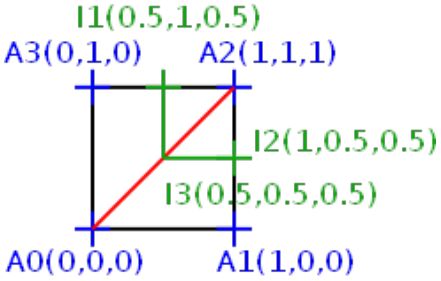

Figure 4: $(A0, A2)$ split

In both cases, I have drawn the $z = 0.5$ isolines (green lines). You can notice that the results are different: in Figure 3, there is only one isoline whereas two isolines are created in figure 4. The isolines are depending on the mesh generation, what is annoying. And we can expect that the 4D mesh generation is more complex than the 3D one.

The marching cubes algorithm consists in computing isosurfaces for each cube, using the values of the function $f(x, y, z)$ on the cube vertices (http://en.wikipedia.org/wiki/Marching_cubes). No artificial mesh is created, what is ensuring better memory and time performances, and a better precision. That's why it has been used in this program.

3 Implementation

3.1 Program overview

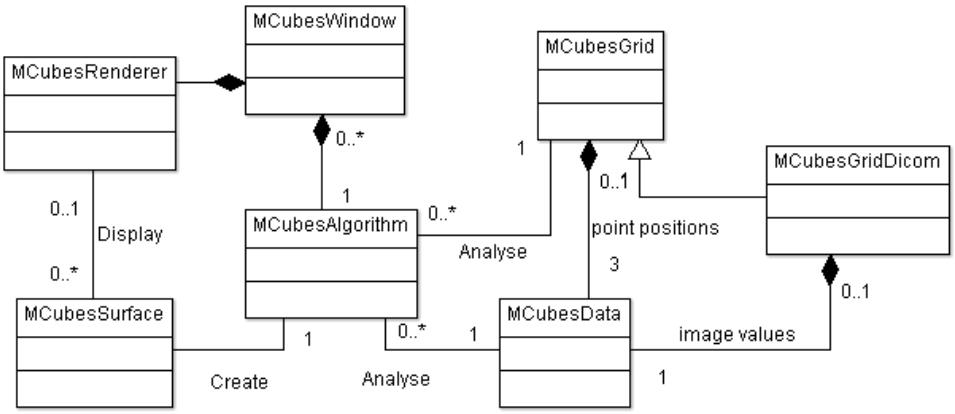

Figure 5: program overview

As I said in the introduction, the program is composed of three parts:

1. Reading DICOM files ($MCubesGridDicom$). It consists in filling a (x, y, z) grid ($MCubesGrid$) and the associated f values ($MCubesData$) using a given DICOM file.

2. Creating isosurfaces using the marching cubes algorithms ($MCubesGrid$, $MCubesAlgorithm$). From a (x, y, z) grid ($MCubesGrid$) and the associated f values ($MCubesData$), a surface ($MCubesSurface$) is created by $MCubesAlgorithm$.

3. Displaying the isosurfaces ($MCubesRenderer$, $MCubesSurface$). The $MCubesRenderer$ is dispaying a given surface ($MCubesSurface$).

Concerning interface with DICOM files, I have used the dcmtk library. This part could be greatly improved and better take into account DICOM meta data.

The 3D scene renderer is based on the OpenGL library. It has few functionalities but we can turn around the displayed surfaces.

The main part of the code deal with the marching cubes algorithm. A presentation is done in the next subsection.

3.2 Implementing the marching cubes algorithm

Marching cubes algorithm

For each grid cube:

- For each cube vertex, compute the sign of "$f(x, y, z) - isoValue$"
- Compute the iso surface
  - Search the marching cubes configuration (using the cube vertices signs)
  - Get the edges intersecting with the isosurface (these edges are associated to the marching cubes configurations)
  - Compute the intersecting points (which are on the previous edges) using linear interpolation

To ensure good performances, I have solved two main problems when implementing the marching cubes algorithm. The first problem is the number of marching cubes configurations: $2^8 = 256$. Programming and testing each configuration is a lot of work. On http://en.wikipedia.org/wiki/Marching_cubes, only 15 unique configurations are given, the others can be computed using rotations and symmetries of the unique configurations. For this reason, I have written a small program, "*MCubesConfigurationGenerator.cpp*", that is generating the 256 configurations from the 15 unique ones. The output of this program is a c++ file, "*MCubesAllConfigurations_init.cpp*", which is containing the code initializing the "*MCubesAllConfigurations*" object (this object is dealing with the marching cubes configurations). The "*MCubesAllConfigurations_init.cpp*" is then added to the project and compiled with the other source code.

The second problem is concerning the isosurface generation. The intersecting points of a cube with the isosurface are shared with other cubes (if the corresponding edge is not on the edge of the grid). We could ignore this and, foreach cube, add the intersecting points to the surface. This means that lots of intersecting point will be present four times in the surface (an edge inside the grid can belong to four cubes). Since the number of cubes

can be huge (about the number of grid points), the number of intersecting points can consum too much memory: this solution is not acceptable.

As a consequence, we have to ensure that intersecting points are shared between cubes. This can be done by creating a link between grid edges and surface points: for each edge, the index of the intersecting points added to the isosurface is stored. However, storing the indexes for all the edges is consuming too much memory: in the grid, there are about $3 \times x \times y \times z$ edges (about 3 edges for each point) and an index is stored on four bytes ('integer' type). Hopefully, the edges of the cubes at a given z are sufficient: $3 \times x \times y$ indexes are stored. The final implementation is taking into account this and is doing some indexes operations...

4 Results

4.1 The 'sphere' example

Here are the results of the marching cubes executed on a hard coded test function, $f(x, y, z) = x^2 + y^2 + z^2$, $x \in [-1; 1]$, $y \in [-2; 2]$ and $z \in [-3; 3]$. This results can be obtained by clicking on the "sphere" icon and changing the isosurface value with the slider or the spin box.

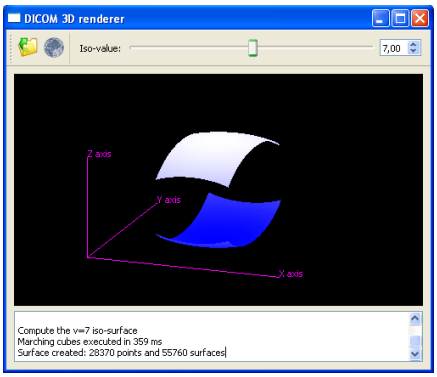

Figure 6: surface $f(x, y, z) = 7.0$

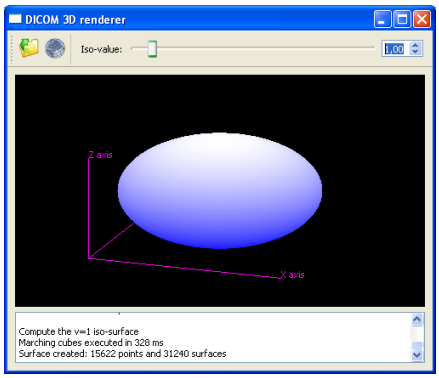

Figure 7: surface $f(x, y, z) = 1.0$

I would like to mention that the execution time of the marching cubes algorithm in function of the number of points is linear. This can be tested by changing the grid used for the sphere function.

4.2 A DICOM file: MR-MONO2-8-16x-heart.dcm

Now, I opened the attached "data
MR-MONO2-8-16x-heart.dcm" DICOM file and set the isovalues to 0.0:

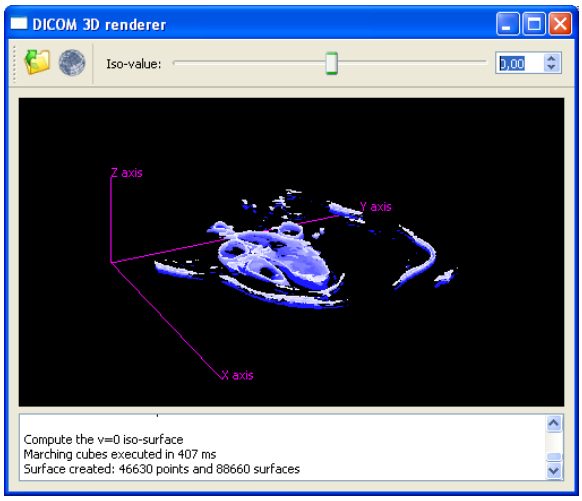

Figure 8: "MR-MONO2-8-16x-heart.dcm" file

A heart is displayed!

5 Improvements

As we have seen in the previous section, the program can create isosurfaces from hard coded functions or from DICOM files. To improve the quality of this software, we could first use a memory debugger and a performance profiling tool. The interface with DICOM files could also be considerably improved, as well as the 3D scene renderer.