REPORT :
Acquisition and reconstruction

December 10, 2018
Enzo Peruzzetto,
Teiki Pepin
Report
Master 2 Informatique
University of Bordeaux

Contents

1 Introduction

The aim of this project is to produce a visualizer allowing to produce a representation of the shepplogan phantom based on a voxelized representation using the simple pgm3d format as well as being able to display obj files.
We attempt to identify and draw the surfaces of each volume before adding transparency to these surfaces in such a way that we can observe the inner part of the displayed model.
Finally, in order to be able to inspect the object, we intend to provide some user interface allowing to turn, move and zoom on the object in the screen.
The project has been developed for Qt 5.7.1, with the usage of Qt Creator 4.2.0 and OpenGL 4.5.0.

2 Project Implementation

Our project is composed in following way (figure 1) : all of our source code files are in the "ARM" folder, including a library called *tinyobjloader*, our input files which describe the models to display are in a folder called "models" and, in the same pattern, our shader files are in their own "shader" folder. We also have a readme file at the root which explains how to compile, run and use our program.

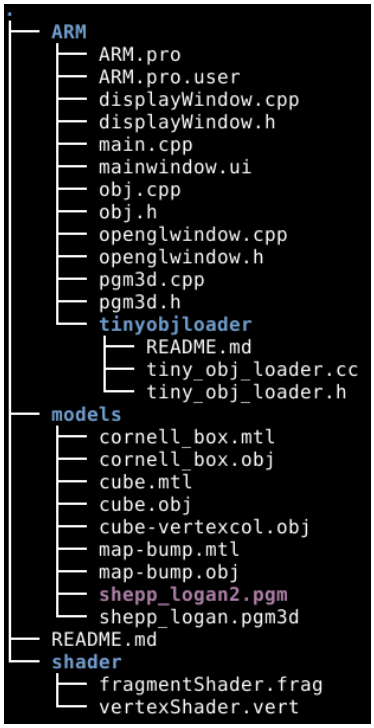

Figure 1: *Project Architecture*

The first issue we encountered in this project was the setup of our architecture. That is, the combination of OpenGL's rendering and Qt's window management. By following the documentation provided by Qt, we were able to create a class *OpenGLWindow* that inherits from *QWindow* which allows us to have a basic display with the functions *Initialise()* and *Render()*. Afterwards, in our class *DisplayWindow*, we overload its functions in order to display the render that we desire.

Furthermore, we have used the Vertex Buffer Objects since rendering via *glBegin()* and *glEnd()* have been deprecated in the latest OpenGL updates for performance reasons as well as using more recent functions in order to keep the source code easily maintainable.

2.1 Display

We started with setting up a rendering method that would allow us to display basic geometry with OpenGL's pipeline. For this, we first gather the coordinates and the colors of each vertices of the input file. We recover different arrays of these data for each display mode.

For the point cloud rendering (figure 2), the vertices included in the array correspond to each point of the point cloud with the following information : their x, y, z coordinates as well as their RGBA color channels.

Figure 2: *Point cloud rendering of a spaceship*

For the triangle based rendering (figure 3), we instead have three successive vertices for each triangle that is to be displayed. Each of these vertices contains the same data as those of the point cloud system.

With these data, depending on the selected display mode, we then proceed to the binding them to OpenGL and send these to the shaders which will display our output image on the active window.

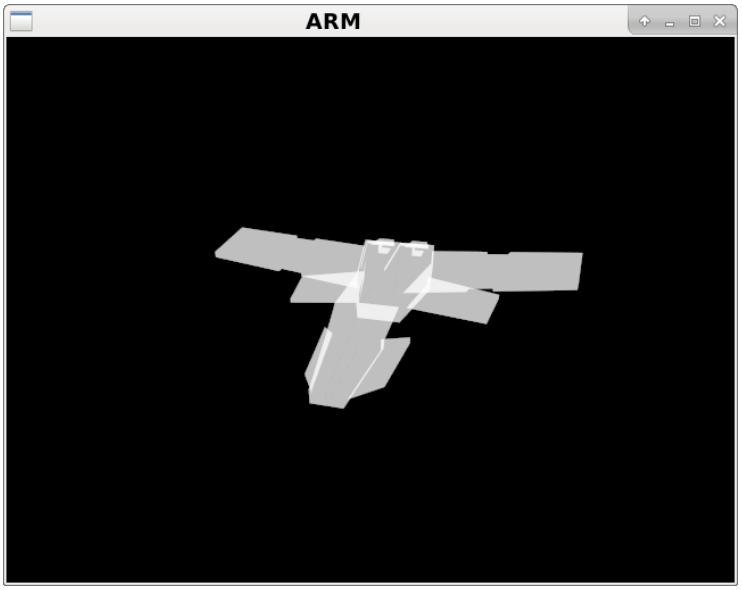

Figure 3: *Triangle mesh rendering of a spaceship*

In order to properly benefit from the display, we have added some user input features. The user can use the left click of the mouse or the arrow keys to rotate the model. The right click of the mouse will allow the user to translate the model along the display. In the same manner, the mouse wheel allows us to zoom in/out. These operations are done by modifying the projection matrix.

We have also implemented the L key to load a new model file. The Escape key allows us to close the window. P will let the user switch to the point cloud display, whereas T will let the user switch back to a triangle based display.

2.2 PGM3D File

In the event that the input file provided to our program is a PGM3D file, we first proceed to the parsing of the data within it. The first few header lines describe the file format, the dimensions of the scene within it and the maximum value a color can have. We then iterate on each line until the end of the file, with each line holding the grayscale value of a successive voxel along the 3D axes. These grayscale and coordinates values are saved into separate arrays that are kept private within the class.

One of the main features of this class is to demarcate our object. Indeed, after reading the file, our arrays contain all the voxels of the scene, some of which

don't actually belong to the displayed object. For this purpose, the function will iterate over each voxel and compare its color value to that of each of its six neighbours (top, bottom, right, left, behind, front). If all of these hold the same color value, we ignore the current voxel; otherwise, we add the voxel's coordinates and color to new arrays, which only hold the filtered voxels.

Once this is done, we have at our usage a list of all the points that determine the borders of specific parts of the objects. This point list can be used as is in order to display a point cloud of our model (figure 4). For the sake of visibility, all points of the point cloud are displayed with a white color and no transparency instead of their actual values.

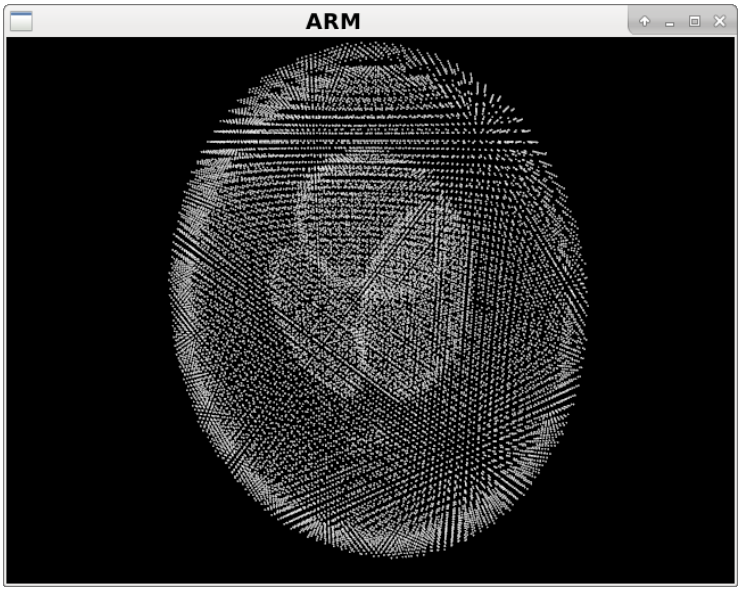

Figure 4: *Point cloud rendering of a Shepp-Logan phantom*

From the previous list of frontier voxels, we can also instead render a cube around each of these points in order to give our model a more solid and continuous appearance. For this, the straightforward approach is is to render all six faces of a cube around the given voxel. Since our rendering through OpenGL only supports triangles, these faces are further divided into two triangles each, for a total of twelve triangles per voxel. Although all of our voxels are bordering voxels with different color values, not all of their neighbours are different. That is, we can proceed to a color comparison of each voxel within our list of border voxels to each of its six neighbours in order to eliminate the need to render faces against an other voxel with the same color.

We then have a list of vertices that make up each triangle, as well as the color of the voxel that each triangle is associated to. We can then send these back our rendering function in order to have a cubic rendering of our model using triangles (figure 5).

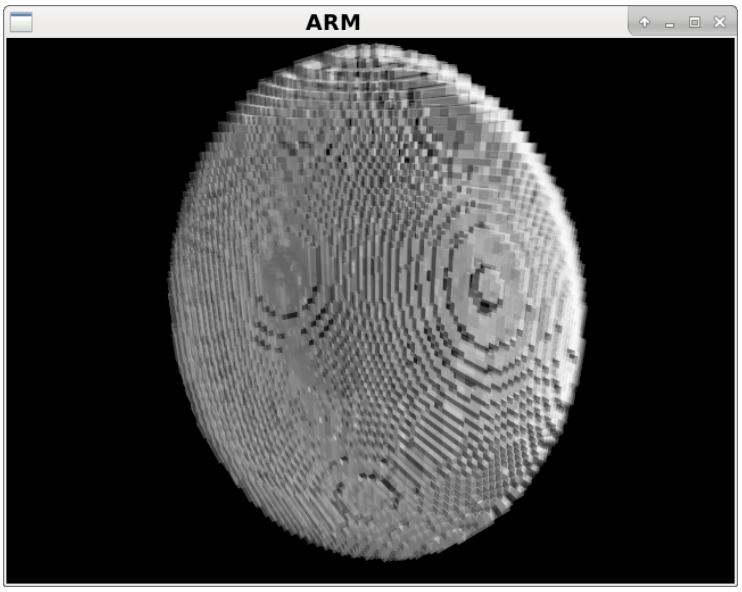

Figure 5: *Cubic rendering of a Shepp-Logan phantom*

2.3 OBJ File

We were asked to implement the rendering of OBJ files for this project. While such files are usually simple to parse, a complex scene can use any number of the many features of OBJ files from the basic vertex data to free-form curves and interpolation information. In order to be able to have OBJ support for our project in time, we have decided to use "tinyobjloader", an open-source library to take care of the file parsing.

The library will read all the polygons, as well as triangulating the faces that aren't triangles, and will store all of the vertices' coordinates, normals, colors and texture coordinates in its own *attrib_t* type. Similarly, all the data needed to link vertices to form a mesh are stored in *shape_t::mesh_t*, such as the number of vertices for each face, which is always three in our case, or the indices that make up each face.

From the data that is stored inside tinyobjloader's datatypes, we extract the relevant information that we need to feed our rendering function. For this matter, we first loop over each shape, then over each face of these shapes, and then over each vertex. For each vertex, we store its coordinates and its color and transparency values in dynamic arrays. These arrays are then sent back to the rendering function to display the model (figure 6).

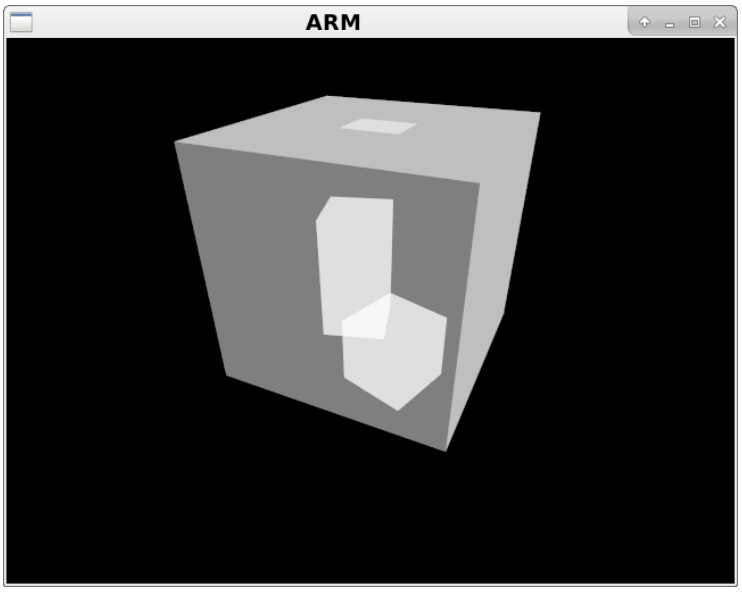

Figure 6: *Cornell box by triangle mesh rendering*

One issue that we had with the usage of this library is that its file loading function will also look for an .mtl file when required, but it does so in the active directory from which the program was launched. However, our .mtl files are stored along with their corresponding .obj files in our "Models" folder, we thus had to force our loading function to look for material files at a provided directory, which was the OBJ file's path chopped at the very last '/' character, which delimits folders. Although we were able to have our program read the .mtl files, these are not currently used in any way in the rendering of our program. As such, all vertices and faces are generated in white.

3 Conclusion

To summarize our project, we have realized the reading and parsing of PGM3D and OBJ files and we have implemented the rendering of the corresponding object. Afterwards, we were able to switch from a point cloud to a triangle mesh for OBJ files, whereas for the PGM3D files, we were able to realize the shift from a point cloud display to a cubic render of the object.

This project has allowed us to grasp the difficulty of going from a cloud point display to a triangle mesh one. Studying known algorithms such as the Delaunay triangulation or even convex hulls would have allowed us to finish the implementation of the aforementioned switch.

With more time, we could have improved our program by finding a way to do the mesh of PGM3D files as well as implementing the textures and colors for our OBJ files.