Visual Rendering & Oculus Rift

Marco Godi & Francesco Giuliari

May 10, 2015

Contents

1 Introduction

The purpose of this project is to make a Volumetric viewer compatible with the Oculus Rift headmount display.

1.1 Volumetric rendering

Volumetric rendering is the name used to describe techniques used to display a 2D projection of a 3D data set. It differs from “normal” rendering Techniques because while more costly it offers more the opportunity to interact with the 3D model so we can split the model, see his internal components and overall get more information at run time.

1.2 The Oculus Rift

The Oculus Rift is a virtual reality headset by Oculus VR that offer the user a fully immersive experience so they can see and interact with the application as if he was seeing with his own eyes instead of looking at a monitor. As of today the most advanced model is the DK2, for this project we’re using the DK1. Our application is not compatible with DK2 because the approach we used is not supported on the newer model and the updated approach needs a more in-depth knowledge of the OpenGL pipeline. Luckily anyone with the required knowledge can make our application compatible by modifying the code where the distortion is applied (More details on later sections).

2 Requirements

2.1 Hardware

- It needs a fairly powerful computer (expecially a powerful video card) to run smoothly because the Volumetric rendering is a resources hungry technique and, since the oculus needs to have a different texture for both of the eyes, the rendering cost doubled.
- The graphic card needs to be able to use OpenGL 3.1 .
- Oculus Rift DK1 or DK2

2.2 Software

- The application at the moments only supports Windows operationg system since it uses OS dependant libraries, it was developed and tested on Windows 8.1.
- To succesfully use the Oculus Rift headest the needs to have the Oculus Runtime software available at the Oculus VR official site.

3 External Libraries

Since this project needed to be completeted in a fairly reasonable amount of time we decided to use some external libraries for an easier management, for the volumetric rendering we're using the VTK libraries, and, for the Oculus Rift management we're using the Oculus SDK libraries

3.1 VTK

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization, and information visualization. VTK also includes ancillary support for 3D interaction widgets, two- and three-dimensional annotation, and parallel computing. At its core, VTK is implemented as a C++ toolkit, requiring users to build applications by combining various objects into an application. The system also supports automated wrapping of the C++ core into Python, Java, and Tcl, so VTK applications may also be written using these interpreted programming languages.

3.2 Oculus SDK

The Oculus SDK is the libraries offered by Oculus VR to manage Oculus Rift headset implementation. It offer methods to keep tracking of the headpose, obtain hardware specifics form the Oculus headset and it allow for creation of virtual heaset useful for debugging the application.

4 Project

4.1 Approach

The application uses the VTK libraries to load the Dicom files (or MHA or VTI files) which are to be visualized as a 3D dataset by a volumetric algorithm, then creates a split screen which shows two different points of view of the visualized dataset (representing the two eyes of the user) with the appropriate distance determined by the eye spacing and then applies a barrel distortion shader so that the image is fixed to display properly on the Oculus Rift.

4.2 Classes

4.2.1 Main

This class has the main function which is obviously the entry point for the program. The command line options are passed at the SmartVolume class, then a custom rendering pipeline is set up wo allow postprocessing (in the vtkStereoDistortPass) for the split vision and the barrel distortion, then we use a utility class defined here to enable Oculus headpose tracking before finally starting the application.

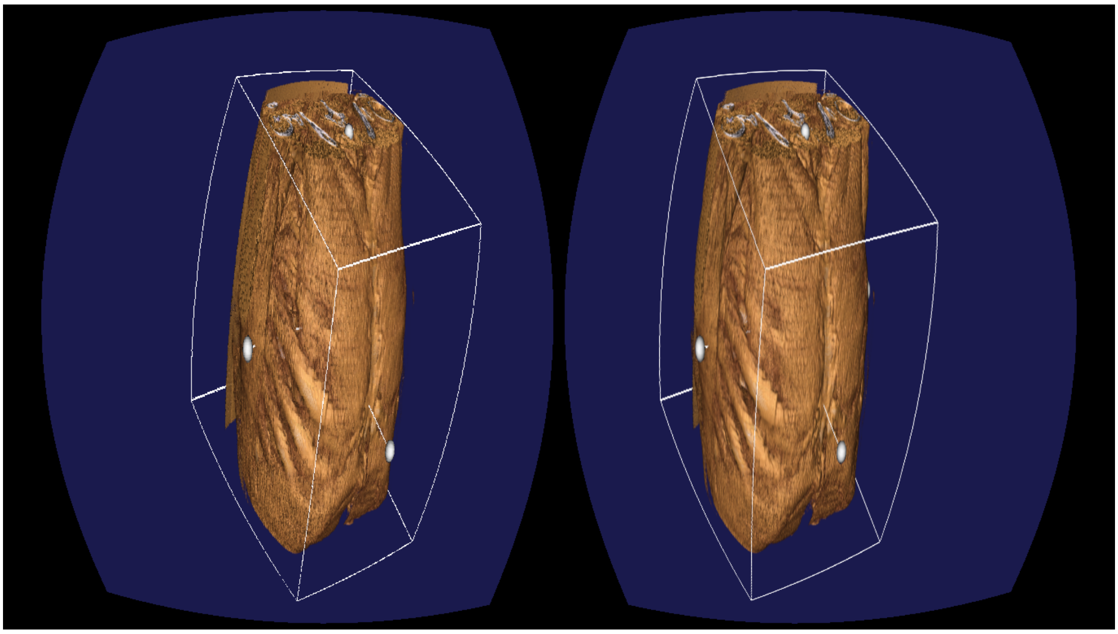

Figure 1: The end result of the output as seen on a normal screen.

4.2.2 SmartVolume

This class can receive command line options (you can pass them through the Init method) which are described in the PrintUsage function. The purpose of this class is to create the graphical environment(window with model and interactions), to load the requested input data (which MUST be passed through string arguments, if DEBUG is enabled it opens the "abdomen" DICOM dataset in the project folder), tovisualize it with the requested blending mode (which are the different ways to map the dataset density values to the desired RGBA data) and to offer interaction with it through mouse and keyboard.

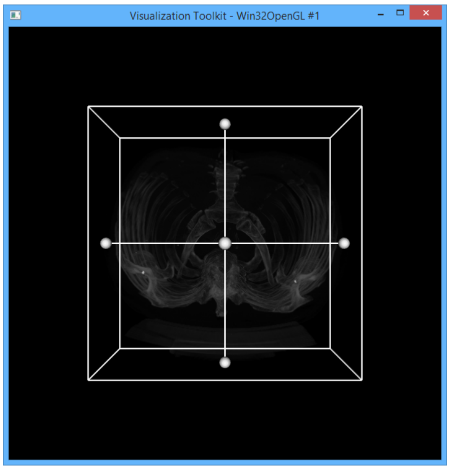

Figure 2: Output of the SmartVolume class.

With the mouse you can rotate the model and interact with an optional bounding box around the model which allows you to cross section the dataset on your desired plane. With the keyboard you can change the blending mode, reset the model position, change the opacity levels.

The key bindings are in the KeypressCallbackFunction function, while the mouse interaction is defined in the VTK default vtkInteractor.

Some local variables of significance:

- opacityWindow e opacityLevel together represent the range of values of density in the dataset in which the alpha value is set (from 0 to 1 from start to end in the range).
- reductionFactor goes from 0.0 to 1.0 and represents the percentage of samples taken from the dataset (reducing this value increases performance at the cost of quality.
- initCam and boxtrans are used in the reset function bound the the R key.

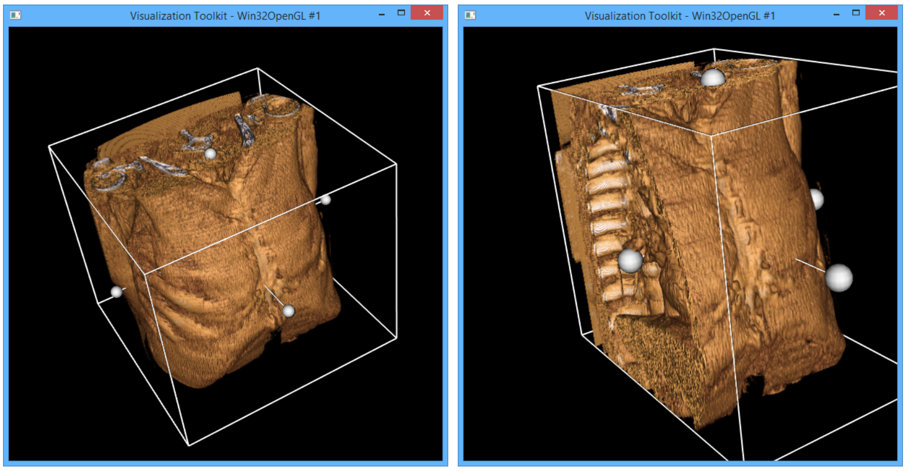

Figure 3: The model can be rotated and also sliced.

4.2.3 VtkStereoDistortPass

This class intervenes at every rendering cycle. At a conceptual level, it renders the scene two times from two slightly translated positions, and then puts them together after barrel distorting them. This is the scene rendered two times

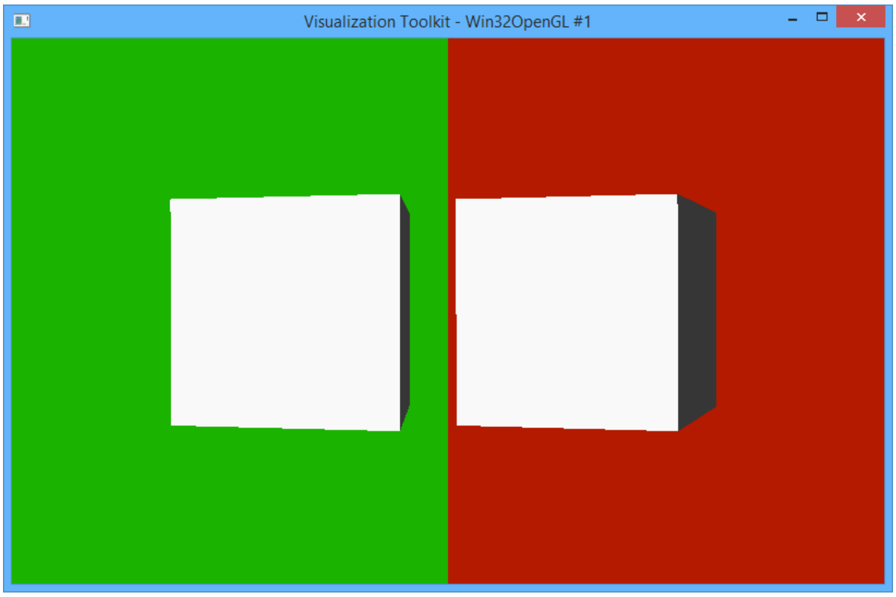

Figure 4: You can see here the translated eye position.

This is realized by using a fragment shader with two textures as input. It needs a value for the InterPupillaryDistance to be set. A lot of code is just setup for the Textures, the FBOs and the ShaderProgram.

The fragment shader (distortion.fs) determines whether the pixel is in the left or the right part of the screen and calculates distortion coordinates accordingly (using parameters set in the vtkStereoDistortPass Class. Then it takes the corresponding pixel color from the determined texture in which the two rendered scenes are stored. Almost all of the OpenGL and graphical setup code is based on the VTK vtkGaussianBlurPass source code.

4.2.4 Oculus_Middleware

This class is used as a class used to initializate the Oculus and return hardware related information such as the head pose and textures size.

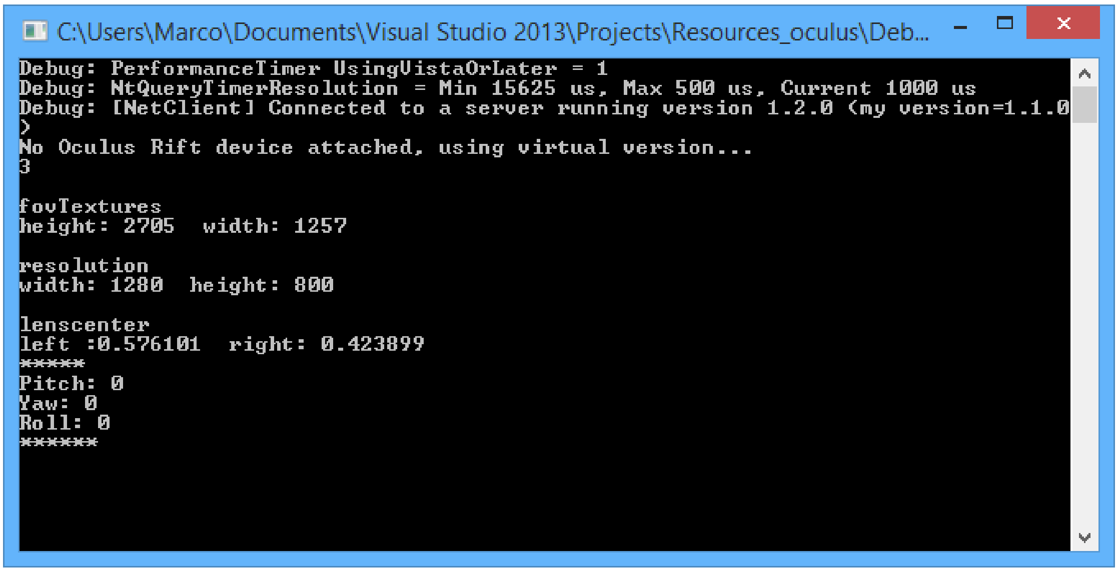

Figure 5: Informations returned from the oculus_middleware.

5 Development journal

Over the course of this stage we had to go through many stages in which we were faced with different problems each one with many alternative solution. In this section will be posted which of these solutions were implemented in the project and the reasons for its choosing.

5.1 Volumetric rendering libraries

The first thing we had to decide was how to handle the volumetric rendering, the possible libraries are linked below

Voreen
Voreen is an open source rapid application development framework for the interactive visualization and analysis of multi-modal volumetric data sets. It provides GPU-based volume rendering and data analysis techniques and offers high flexibility when developing new analysis workflows in collaboration with domain experts. The Voreen framework consists of a multi-platform C++ library, which can be easily integrated into existing applications, and a Qt-based stand-alone application.

VTK
The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization, and information visualization. VTK also includes ancillary support for 3D interaction widgets, two- and three-dimensional annotation, and parallel computing. At its core, VTK is implemented as a C++ toolkit, requiring users to build applications by combining various objects into an application.

Open Inventor
Open Inventor® is an object-oriented, cross-platform 3D graphics toolkit for the development of industrial-strength, interactive applications using C++, .NET or Java. Its easy-to-use API, its extensible architecture, and its large set of advanced components provide developers with a high-level platform for rapid prototyping and development of 3D graphics applications.

After a long discussion among ourselves where we studied the pros and cons of each library we decided to use the VTK libraries mainly for these two reason :

- Vooren, while offering a whole framework and many already implemented function, has almost no documentation and we didn't have enough time to learn it from scratch.
- Open inventor is not open sourced.

5.2 Omegalib

Now that we decided to use the VTK as volumetric rendering library, we looked up projects similar to this one so we could use that as a base or as an inspiration.After a little search we found a project that seems to be the silver bullet for our problem, that being Omegalib by UIC Electronic Visualization Laboratory.

The main features that leaded us to try using Omegalib are:

- It implements two rendering libraries, OpenSceneGraph and VTK.
- It natively redirects the output on various devices, among them is the OculusRift.
- Support for a wide range of input peripherals (controllers, motion capture systems, touch surfaces, speech and brain interfaces), through the Omicron toolkit.
- It offers a python wrapper so that the program could be written using a simpler language.

We decided that learning to use the Omegalib was our best bet to make this project easier to extend for future development, so we rewrote our vtk application as a python script. On paper the last thing that needed to be done was assigning the oculus rift configuration to the application, but when we tried doing that it didn't work and the results weren't as expected. We attempted reviewing the whole library code that manage the output control, specifically how it interacted with the oculus rift configuration and then contacted the Omegalib developer on the project mailinglist to see if there was a way to see what the problem was and how to resolve it. Turns out that, while Omegalib implements the VTK, it doesn't support the volumetric rendering, at least it doesn't as of now, it may be implemented in the future.

Because of this any and all plans of using Omegalib for this project fell through so we were back at the beginning and had to go with pure VTK again.

So what we needed was to find a way to implement all of these features:

- Volumetric Visualization of 3D datasets
- "Two points of view" management (for the two eyes of the Oculus user)
- Applying barrel distortion
- Inserting head pose tracking into a VTK application
- Outputting video on the Oculus Rift

We looked for other similar projects and two caught our attention: multipass_vtk by zadacka (https://github.com/zadacka/multipass_vtk) and and application by Przemysław Brudny and Mateusz Wójcik by the Universidade de Aveiro (which you can find here http://sweet.ua.pt/paulo.dias/rva/TrabalhosRVA_2.htm). The first one sadly only supports Linux and uses an older version of the Oculus SDK, but uses an interesting approach (the multipass system offered by VTK) which we'll look more into later. The second one also uses an older version of the Oculus SDK. Also it uses a software distortion of the image, making it too slow for an interactive experience.

While we were studying these projects, we started developing other components of our software. Starting from a VTK example, we created a Volumetric viewer which takes a series of .dcm files (or a .vti or a .mha) and visualizes it on screen, making it possibile to manipulate it by moving the camera and using widgets to make cross sections of it in real time.

While we were looking into a way to apply a barrel distortion and output our software onto the Oculus Rift, we also created a program which visualized onto two viewports on a split screen a normal VTK application, each viewport having a translated camera based on the eye distance value. We made this software and it worked, but then we discarded it because we found another solution, easier to manage and less verbose. We also looked into pre-existing OpenGL Oculus Rift applications to get an idea about how to make a VR software. We found very soon that the latest way to do it involved pure OpenGL which is very problematic to use in a VTK application without knowing it very well because you could mess withthe VTK pipeline. So we opted for a deprecated way, which includes distortion using a shader in post-processing and using Mirrored Desktop so you can use the Oculus as an external monitor.

So we learned of two ways to apply post processing into a VTK application:

- The external way: WindowToImageFilter, which "takes a screenshot" of a viewport and lets you use it as an Actor2D... Two problems: it's very slow (it loads the screenshot from video memory to RAM and the to video memory again) and also doesn't support shaders.
- The internal way: Multipass (inspired by zadacka's multipass_vtk) which lets you build the rendering pipeline of an application, sometimes down to the OpenGL code. Using a Gaussian Blur Post Processing example as a starting point we built a class which solves two of our needed features: "Two points of view" (by making the render pass render the scene two times from different positions) and barrel distortion (by applying a shader to the texture in which we store the rendered scene).

So we had a volumetric viewer, we could have video from our two eyes, we could distort it and output all of this into our Oculus Rift by mirroring the screen. We only needed to handle the user's Head Pose, which we did by polling it trough the Oculus SDK every few milliseconds.