JOGL LIVE RENDERING TECHNIQUES IN DATA ACQUISITION SYSTEMS

C. Cocho*, F. Cecillon, A. Elaazzouzi, J. Locatelli, Y. Le Goc, P. Mutti, H. Ortiz, J. Ratel
Institut Laue-Langevin, Grenoble, France

Abstract

One of the major challenges in instrument control is to provide a fast and scientifically correct representation of the data collected by the detector through the data acquisition system. Despite the availability nowadays of a large number of excellent libraries for off-line data plotting, the real-time 2D and 3D data rendering still suffers of performance issues related namely to the amount of information to be displayed.

The current paper describes new methods of image generation (rendering) based on JOGL library used for data acquisition at the Institut Laue-Langevin (ILL) on instruments that require either high image resolution or large number of images rendered at the same time. These new methods involve the definition of data buffers and the usage of the GPU memory, technique known as Vertex Buffer Object (VBO). Implementation of different modes of rendering, on-screen and off-screen, will be also detailed.

INTRODUCTION

The NOMAD system [1] is used to set up and monitor all the experiments carried out within the neutron instruments at ILL. NOMAD is based on client-server architecture: the C++ server contains all the routines and algorithm/mathematical methods to set up and control the instrument devices; on the other side the Java SWT client is the graphical user interface (GUI) used control and data visualization.

One important part of the GUI is the display where the data obtained in the experiment are shown. There are different graphic libraries than can be used to render data. The Python scientific graphics libraries GuiQwt [2] and PyQtGraph [3] offer interesting functionalities and provide good performances for online rendering of 2D graphics. However integration of Python code in the Nomad environment is difficult. The Java libraries TANGO [4] and Jzy3d [5] are easy to integrate and also offer the rendering of real-time 2D graphics but they are based on Java 2D [6] that benefits from a hardware acceleration on major platforms but cannot reach the performance of an OpenGL-based solution that is closer to hardware and allows to take advantage of the most performant capabilities of the graphic card.

Typically the Position Sensitive Detectors (PSD) generate arrays of counted events which are either 1D or 2D corresponding to the geometry of the detector. An additional dimension can be added if time-dependant results are obtained. To visualize the detector data in Nomad, we display a 2D color image.

The reason why OpenGL [7] was chosen as a graphic library in Nomad because we need a powerful library capable to do onscreen rendering of large amounts of data with a high refresh frequency. As the Nomad GUI is developed in Java, we use the library JOGL [8] to be able to have full OpenGL functionalities.

In the last years, the evolution of the instruments at ILL has increased the quantity of data that needs to be rendered and has led to performance problems. We found out two main problems. The first was the long rendering time required when having large amounts of data corresponding to a single measure. The need to have real time data display complicates the task. We decided to improve our rendering technique by testing new possibilities based on the usage of vertex arrays and vertex buffers [9].

The second problem involved multiple data rendering i.e. rendering different types of data or data generated by different detectors. In this case, it was not necessary to render the data on-screen and therefore the off-screen rendering technique was implemented.

METHOD

OpenGL is based on a client-server structure [10]; the client side corresponds to the part of the program that is in CPU memory while the server side is the part that resides in the Graphics Processing Unit (GPU) hardware and memory. Scenes in OpenGL are defined as meshes of triangles, which represent the geometry of the objects drawn (known as vertices). Vertices are associated to additional rendering information such as colors, textures. More realistic rendering can be obtained by adding lighting and shading information.

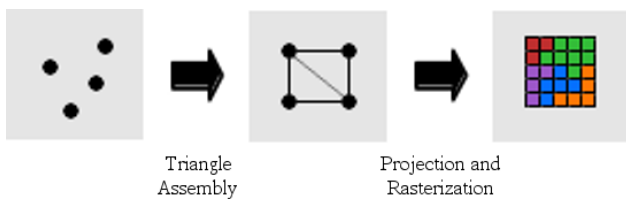

Figure 1: Simplified OpenGL pipeline.

OpenGL takes as input a set of vertices that are transformed progressively into pixels. They are temporarily assembled into triangle primitives before being transformed into pixels during the rasterization

*cocho@ill.fr

ISBN 978-3-95450-139-7

phase (See Figure 1). Note that OpenGL can render meshes with perspective projection ("classic" 3D) or orthographic projection. To render our 2D images, we need to convert the detector data into a set of 2D vertices associated to a color attribute and use the orthographic projection.

On-screen Rendering

The initial algorithm used in Nomad for on-screen real time rendering was the Immediate Mode. This consists of the CPU communicating to the GPU all the data vertex by vertex which implies that there are as many calls to the GPU as vertices that need to be rendered. The Immediate Mode is a widely used rendering technique due to its simplicity; however it has a main drawback: the rendering time increases considerably with increasing data size.

The bottleneck in this rendering algorithm is in the excessive communication between the CPU and the GPU. In order to avoid that, OpenGL provides the capability to create and store all the data in one array in RAM and then send it to the GPU. This technique is called Vertex Array (VA) and allows to decrease the rendering time by reducing the number of interactions between the CPU and the GPU.

There are two main steps in this technique: the first one is to define and fill the arrays and the second one is to copy them to the GPU memory. The simplest arrays we can have are the ones containing the vertices data and another with the color data**.** The use of vertex arrays allows reusing the vertex data: that is when multiple areas share some vertices, these vertices will not be repeated in the arrays, and therefore the amount of data stored is smaller. In this case, it is possible to index our data and use an index buffer in collaboration with the vertex one. For our 2D rendering, we finally chose the use of indexed vertex arrays.

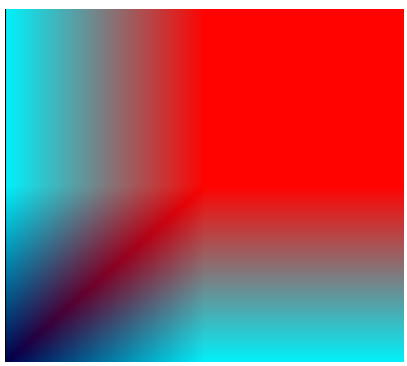

Figure 2: Smooth rendering of a 2x2 detector.

The result of VA rendering is a smooth image, that is, an image where the transition between the colors associated to each vertex is smooth due to the behaviour of OpenGL (See Figure 2). For the scientific purposes of Nomad it is necessary to maintain the same look and feel of the data produced by the detectors; that means we need a pixelated image where each detector pixel is clearly defined. To achieve this we have to quadruple our data because each vertex will have four colors associated (See Figure 3).

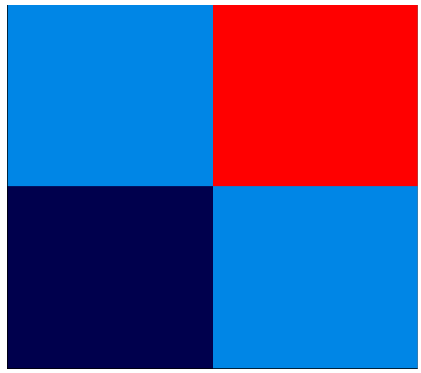

Figure 3: Non-smooth rendering of a 2x2 detector.

Even though the number of function calls is reduced, VA technique requires not only the initialization of the arrays (at least one vertex and one color array) but the arrangement of the data in order to let the GPU to read it properly.

A drawback of VA technique is that it stores the data in the client side (CPU) and therefore does not make use of the high-performance memory of the GPU. OpenGL provides the buffer object functionality that allows storing data in the server side (GPU). A buffer object is an OpenGL object used to store an array of data in the GPU memory. When the buffer object is used to store vertex data they are called vertex buffer objects (VBO) [11]. As with VA, the implementation of the VBO technique requires the initialization of the arrays which will be instantiated and filled using the same process.

In the Nomad system, most of the render frames correspond to new data that need to be rendered (due to the high refresh frequency). As the dimensions of the data do not change (because the detector geometry does not change), we do not need to recreate the vertex positions. However due to the increasing number of counts with time, and taking into account this variation of the number of counts is represented (through the pipeline, as described in Figure 1) by a colors scale, we need to update the color values. The VBO technique does not require to send the data to the GPU memory if it has not changed. This fact represents an advantage compared to VA that needs to transfer the entire arrays to the GPU at each frame.

Off-screen Rendering

The implementation of VA and VBO based techniques in Nomad was carried out in order to avoid high render times when having a single heavy measure (data produced by a big surface detector). However there are other situations where the problem is not the size of the data to be rendered but the number of different data sources to render at the same time. This is the case of instruments

ISBN 978-3-95450-139-7

which have multiple detectors and therefore generate multiple images. Having real time rendering was not required and, as the rendered images were meant to be used not only in displays, we decided to do off-screen rendering.

The off-screen rendering consists of rendering an image which is not going to be displayed directly but saved in memory or as an image file. In our case, the off-screen rendering was done by implementing the Frame Buffer Objects (FBO). A FBO [12] is an OpenGL object that allows to define Framebuffers. A Framebuffer is a group of buffers used for rendering purposes.

Each JOGL application has one or more contexts that store the rendering states (selected buffers, etc.). When creating an OpenGL context a default Framebuffer is created. It represents the display or window where the image is rendered. In fact it is the combination of up to four color buffers (most commonly known as the front and back buffers). In addition to that, OpenGL allows any user to define their own Framebuffers for multiple purposes such as rendering to textures and doing off-screen rendering. Framebuffers are composed of different types of buffers depending on the type of data to store. In our case, as we wanted to implement off-screen rendering, we used the OpenGL buffers foreseen to that: the render buffers. Once the FBO are initialized, following the same process as with other OpenGL objects such as VBO, the rendering algorithm is exactly the same as the one used in on-screen mode.

In our case, the main purpose of the off-screen rendering was to generate images which represent the detector status for the running experiments. The status of each running experiment can be followed thought the ILL web page (http://nomad.ill.fr). As the generation of the off-screen images is independent of the Nomad client, we implemented an off-screen client which renders the data when necessary.

RESULTS

We carried out different performance tests for the on-screen rendering of detector data. We focused on measuring the time necessary to render a specific amount of data. The tests were done by setting up a simple display were the data rendered changed with a given frequency.

We run the same tests for the three methods: Immediate, Vertex Array and Vertex Buffer Object. For all of them the aim was to study the evolution of the rendering times as a function of the size of data.

Even though we studied the overall rendering time, its definition is strictly dependent on the rendering technique. In the case of the Immediate Mode, the rendering time is the time to pass and render the data. In the case of VA and VBO, the whole time can be divided in three parts: the first one is the initialization time, the second one is the updating time (if necessary) and the third one is the drawing time.

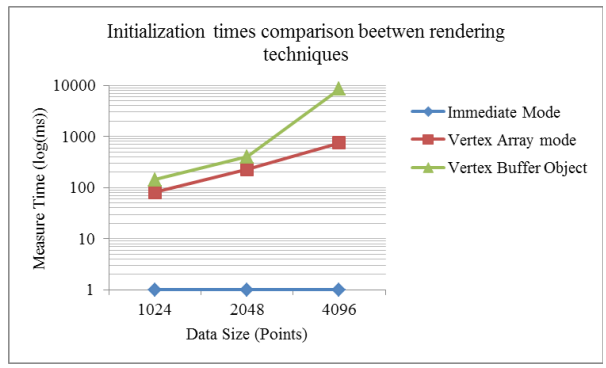

Figure 4: Initialization time comparison for Immediate Mode, Vertex Array and Vertex Buffer Object rendering techniques.

For the Vertex Array, the initialization time is the time required to initialize and fill the arrays of vertices, colors and indices. The update time is the time to pass all the data including the vertex array as well as the color array from the CPU to the GPU.

For the Vertex Buffer Object, the initialization time is also the time to initialize and fill the arrays of vertices, colors and indices. However the update time is the time to pass only the color array as the size of the data is constant and the vertex coordinates do not change.

Tests were performed on a linux PC running at 3.33GHz and 4GB of memory with nVidia Quadro (256MB) graphics card. Note that results may vary depending on graphics cards.

The results showed that VBO technique has the best performance (See Figure 5). The only drawback is the higher time needed to initialize the buffer objects (time which depends on the buffers size), as shown in Figure 4.

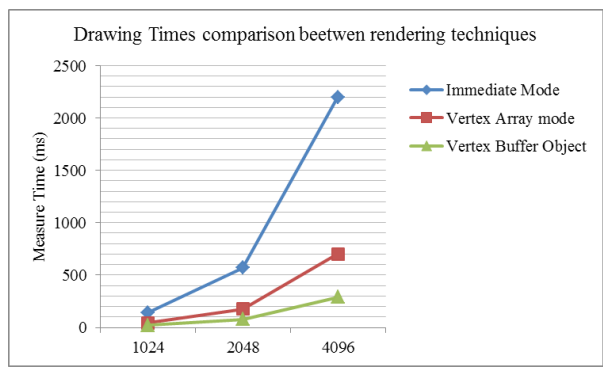

Figure 5: Drawing time comparison for Immediate Mode, Vertex Array and Vertex Buffer Object rendering techniques.

While doing the performance tests we found out that the Java Virtual Machine JVM (JVM) runs out of memory when using data size exceeding 4096*4096 points.

Most of the objects created by Java reside in the heap memory (the Garbage collector is in charge of cleaning it). However the JVM also uses native memory. Both

ISBN 978-3-95450-139-7

memory sizes can be configured by changing the value of specific JVM properties [13].

A group of memory tests were done in order to study how the memory configuration of the JVM influenced the maximum size of our buffers. The tests consisted of increasing heap and native memory size independently for a specific size of data and analyse when the JVM runs out of memory. We repeated the same process changing the data size.

These memory tests provided us the ideal JVM configuration for a specific amount of data to render.

CONCLUSION

After all the tests were carried out, VBO technique was shown to be the most efficient, especially for large amounts of data.

However for large detector geometries producing large amounts of data, we found that the amount of data to render was excessive regarding the size of the display. Most of the time such a quantity of information is not appreciated by our vision. A further optimization is trying to reduce the size of data buffers used in VBO technique by reducing the quantity of data to render.

Moreover, as we are interested in having a pixelated image, the vertex reuse provided by VBO technique (and also VA) is not utilised. In this case, there is little interest in having an index buffer. We can just define two buffers one for the color data and another for the vertex data. In this case, the overall memory used will be also diminished.

It is also important to point out that even though we were interested in using VBO to increase our rendering performance, there is another important characteristic in VBO: the capability to use the same data stored in GPU memory for different purposes. That means if our application must render the same data in different ways, for example on-screen and off-screen, it will only be necessary to store the data in the GPU one time.

REFERENCES

[1] P. Mutti, "Nomad-More than a Simple Sequencer". ICALEPCS'11, Grenoble, France.
[2] guiQWT. http://code.google.com/p/guiqwt/
[3] PyQtGraph. http://www.pyqtgraph.org/
[4] TANGO. http://www.tango-controls.org/
[5] Jzy3d. http://www.jzy3d.org/
[6] Java 2D. http://docs.oracle.com/javase/tutorial/2d/
[7] Nicholas Haemel, Graham Sellers, Benjamin Lipchak Richard S.Wright Jr, "OpenGL SuperBible," in *OpenGL SuperBible*, Fifth Edition ed., ch. 2: Getting Started, p. 34.
[8] Java Binding for the OpenGL (JOGL). https://jogamp.org/jogl/www/
[9] Mark J.Kilgard, Modern OpenGL usage: Using Vertex Buffer Objects well, September 9, 2008.
[10] Nicholas Haemel, Graham Sellers, Benjamin Lipchak Richard S.Wright Jr, *OpenGL SuperBible*., ch. 3: Basic Rendering, p. 81.
[11] OpenGL Vertex Specification. http://www.opengl.org/wiki/Vertex_Specification
[12] OpenGL Framebuffer Object. http://www.opengl.org/wiki/Framebuffer_Object
[13] JVM Memory Settings and System Performance. http://www.ibm.com/developerworks/library/j-nativememory-linux/

ISBN 978-3-95450-139-7