OpenGL View Library

Adam Riečický*
Supervised by: Martin Madaras†

Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava
Bratislava/Slovakia

Abstract

In the paper, we propose a library for the viewing of OpenGL textures, models and other resources. The included library is adding a possibility for users to open an additional window beside their program, which can be used for displaying models, variables and textures in multiple ways. The user can configure window layout and customize what will be displayed in the window. The library also supports creating more layouts and switching between them during runtime. The user is free to apply his own shaders and vertex attributes for individual objects to customize the rendering. The library can be used as a supportive viewer or tool for debugging OpenGL applications.

Keywords: OpenGL, mesh viewer, texture viewer, C++ library, buffer visualization

1 Introduction

Nowadays, many programmers need to use graphical output in their programs. It may become useful also for the programmers for whom the graphical output is not the main intention. For this purpose, various graphical libraries such as OpenGL [1] or DirectX are chosen to get maximal rendering efficiency.

Since these libraries are working directly with graphic accelerators, the performance is robust, but it has its cost especially for the developers. Data stored in video card memory are hard to review by the program debugging and errors on this level are unpleasant and disturbing. Moreover, programmers often have these data stored on graphic card related with data in their application, which makes debugging even more frustrating.

Our goal was to create a visualization tool for these programmers, where they would be able to inspect models and textures stored in the video memory, and connect it with the data from their program. This should be done by providing a library with a simple interface which can create another window beside the user's application and offer previewing possibilities.

*a.riecicky@gmail.com
†martin.madaras@gmail.com

The paper is organised as follows. The first section is devoted to similar solutions. It provides a closer look on currently existing tools, and summarizes what is missing in those solutions compared to our solution. The second part specifies how the rendering of the structures is done. The next section is devoted to a rough description of the implementation. The last section summarizes the testing and results that we have achieved. A concrete outcome of this work is a library that can monitor and display different types of data for various applications.

2 Related Work

The most widely used debuggers which we discuss and compare in this section are gDEBugger [2], nSight [3], Vogl [4] and few others. Each of them offer slightly different features, but the main purpose is the same, similar with ours.

The gDEBugger is being promoted as an advanced OpenGL debugger, profiler and memory analyser. It is a stand-alone application, which allows the user to select an executable he wants to debug. It runs the executable under its environment and allows the user to display textures, shaders, OpenGL state and other resources created and used by the application. After pausing the application, the user can list down all textures, buffers, and objects located on the graphic card in a moment. It also informs the user about the performance and function calls. Since gDEBugger is running over an executable, the user cannot debug the variables from his application. Our library can display both memory and video-memory data, also beside the custom debugging environment.

In contrast, the nSight debugger from Nvidia extends traditional code debugging environments. It can be built in the Visual studio or Eclipse environment and offers the user an ability to see almost anything what is related to the video card memory, while he is debugging his own code. The nSight can be used for debugging on CPU, GPU and shaders simultaneously. However, an obvious main disadvantage of this product is that it supports the latest versions of the nVidia graphic cards only. Our solution is designed to support also older versions of OpenGL.

The Vogl is being promoted as an OpenGL capture / playback debugger. It is a new tool currently in alpha re-

lease, which support both Linux and Windows platforms. It handles logging all OpenGL state into the file, with reviewing possibilities. Compared to our solution, at the current state it does not offer any graphical output for debugging, despite the large amount of information logged.

There are also many other tools which can be used for debugging OpenGL contexts. The BuGLe [5] - similar to nSight but running on UNIX-like systems. It can be used for debugging and profiling OpenGL applications including shader code, buffers and a visual feedback of the textures, the color and the depth buffers. Since November 23 2014, BuGLe is no longer being developed. Another tool, the GLIntercept [6] can log all OpenGL calls but it was mainly designed for OpenGL to version 2.1. Can be declared that it is similar, however older solution then the Vogl.

Despite all the pros of the mentioned tools, there are many limitations. None of the tools provides a visualization of meshes, and that was the main reason and motivation to create our own. We wanted to let the user to see not only the array of values in the buffer, but also a 3D visualization of them. Debugging the meshes in a visual form is much more intuitive then listing a buffer values.

Most of the tools, like gDEBugger and nSight, need to pause the application before they can be used. It can be restrictive in some cases, for example when debugging an animation, and it may lead to complications such as need of frame-by-frame data debugging. Our library works real-time, allowing the users to view mesh or texture animations instantly.

In mesh processing algorithms, an output is often represented as an array of values, corresponding to individual vertices of the mesh. These values can be for example mesh diameter in vertex, skinning weights for single bone, curvature or other vertex properties. Rendering of these values in the user's application would need creation of specific shaders and buffers applied on a meshes. We wanted to offer the user a possibility for creating a link between data in the memory and data on the video card, with a visual output. Our solution transforms an array of values into a color data and then assign them to a vertices of a mesh, which is the feature that is not present in any of known debuggers.

In order to allow the user other possibilities, we added a visualization of texture data and displaying variable values. To sum up, our tool may not be as complex as the existing solutions mentioned before, but it provides visualization possibilities that are beyond the limits of the other tools.

Compared to other debugging tools, our framework is displaying data defined by the user only, instead of all OpenGL context. This may result in the better clarity of displayed data, and filtering all not requited buffers.

Similarly to the gDEBugger, our framework works in an additional window running beside the users application. This window displays all resources monitored by the tool and can be customized, depending on the needs and preferences of the user. This feature also makes it suitable for supplementary visual output applications, not only for debugging purposes.

3 Visualization Methods

Our solution is displaying user defined structures only. We needed a mean to uniquely identify them, which would also help the user to distinguish between them. All the data monitored by the library have therefore its unique caption, defined by the user.

3.1 Mesh Visualization

Mesh rendering is the feature of our framework on which we were focused the most. In its simplest form, our framework can render a vertex buffer displaying a mesh as a point cloud. This form of visualization does not require additional information about the mesh, beside a buffer and number of vertices. This allows the user to get a visual feedback in a single library function call. In the next step, the mesh data can be adjusted by other callings to specify vertex connectivity, texture coordinates or vertex attributes.

The vertex connectivity can be added by sending an index buffer. Specifying element type, viewer treats a vertex array as a sequence of elements. From that moment the mesh is not rendered just as a point cloud, but individual connected elements can be seen. Similar procedure can be used for the objects which should be textured, but there is a need to have the vertex connectivity specified in the moment. Object is then displayed as a full solid textured mesh, which can be viewed in the window.

After this, a rendering of the model is done by the library's internal shader program which can be replaced by the user specific shader program. It can modify the way how the model in a library window is rendered. It is possible for the user to use vertex attribute buffers for a shader input as well.

To each model, an array of values can be assigned. It manipulates the vertex colors depending on the value. For each individual vertex one value is taken from a field and transformed to a color using a selected color scale. There are several possibilities how the value can be changed into the color. For our solution we selected a linear color mapping function, which is trivial to implement, with a low computing cost, and result which is adequate and fully sufficient for our needs. The vertex value is mapped on a predefined linear color scale (e.g. blue for the minimal and red for maximal value), and then applied as a vertex color.

3.2 Other Data

Beside the one main purpose, which is a vertex-buffer and mesh visualization, we want to offer the user ability

to display other structures, to enlarge usage possibilities. Specifically there are two other options - inspection of the textures and tracking application variables.

Textures stored in the video memory can be monitored via their individual buffer ID generated by OpenGL. Sending this ID to the library allows the user to display specific textures in the Viewer window. All the textures that were sent to the library, can be displayed and viewed. This feature is not as complex as the mesh viewing, but we decided to implement it, because it often may come in handy to have it available. It can be used for example to review depth or color frame buffer textures, normal, diffuse and other texture properties of a model.

The last function allows the user to set a variable pointers to the library and then display actual values. Output of the each pointer can be formatted and inserted into a defined string line. These strings can then be selected in a library window and are displayed as standard text output. This function may be useful when there is a need to see variables in a real-time without the restriction of application pausing.

4 Implementation

The library offers a set of tools for previewing data structures. User can specify which OpenGL context he wants to share with the library, and which structures he wants to display. Viewing framework then makes a list of these structures and the user can select and see the actual look of the data at runtime.

Primarily, the library is designed to work with OpenGL version 4+, however it can be used with projects that are written in lower OpenGL version standards. The project is implemented as a static library that can be linked to any C++ project. The library header file, contains all function callings that the library provides.

4.1 Architecture

The library interface function callings can be used to specify which resources should be displayed in the library window. By using them, the user passes to the library all information needed. The framework stores all the data which has been sent and offers them to select at runtime (User-Library interaction described in Figure 1). Each structure has its own caption - a user defined text description of the resource. The user can identify and select the resource he wants to display by the caption.

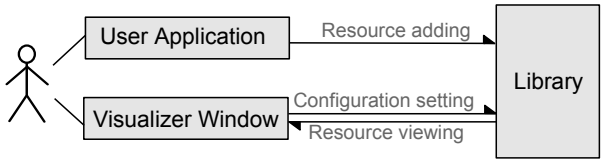

Figure 1: User - Library interaction

The Viewer Library runtime is separated into two modes to make it easier to adjust for different purposes. First, there is the configuration mode. It allows the user to create a layout of the window and specify parts of it, where the rendering of the individual structures will be done. This layout serves as a starting point for the second, and more important part - the viewing mode.

The viewing mode is the main feature of the program, which allows the user to list all stored data, such as meshes and textures stored in video memory or to display variables. It is possible to select in real-time what to display at the current moment. The number of structures and data that can be reviewed in this mode are dependent on the code interface calls. That means, that if there were no interface calls, nothing can be viewed in the viewing mode.

4.2 Context Sharing

Each application that is running OpenGL has its context. It stores all the state associated with the instance of OpenGL. Each resource generated and stored on the graphic card is specific for the instance of the context, which means that two applications running OpenGL cannot see nor access the context of the other.

Our Viewing Library uses context sharing. This means that on initialization, the context created by the user program needs to be shared with the library. It allows the library to operate on the same context as the user program. The difference between separated and shared context is shown in Figure 2.

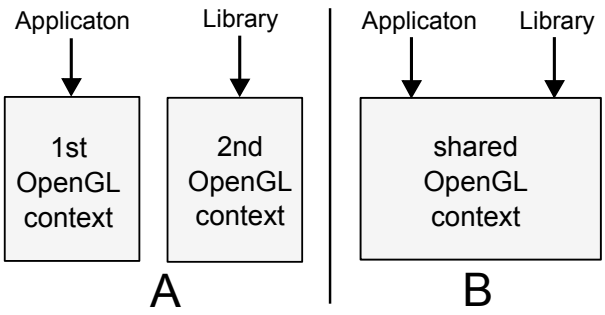

Figure 2: A - two separate OpenGL contexts, B - single shared OpenGL context

4.3 Viewing Mode

The viewing mode is the main part of the application, and it is the actual visual feedback for the user. In this mode the user can display all the data which were sent to the Library through the code. In Figure 3, an example of viewing layout is displayed.

Viewing mode distinguishes three types of the field, which can be rearranged in the configuration mode. Mesh view is used for displaying vertices from the vertex buffers and seeing the correlation of them with the value arrays

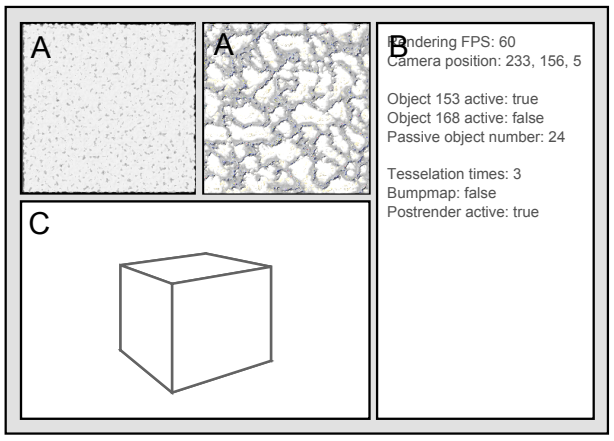

Figure 3: Example of the viewing mode layout.
A - Texture views, B - Variable display, C - Mesh view

from the computer memory. It is possible to rotate and zoom the individual objects in the view. Texture view determines an area where the textures can be displayed. Variable view is the area which displays text output and the actual value of the defined variables. Variables are added by the user program, and lines to display can be selected by the user for each individual variable display.

4.4 Configuration Mode

This mode is focused on creating a custom layout for the window, as the one which can be seen in Figure 4. The layout is represented as a set of rectangular fields. The type of each can be changed by the user. There are three main types of fields which can be selected: mesh, texture and variable field type.

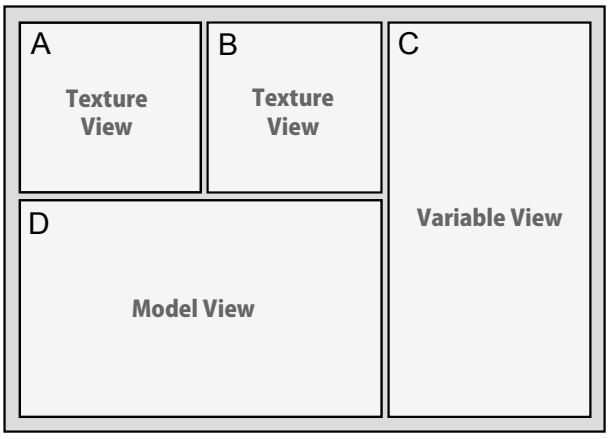

Figure 4: Example of the configuration.

Our framework supports profile creation. Each profile can hold different configurations of a window. The user can switch between profiles and modify them at runtime. Each session comes with one default profile which cannot be deleted and which can be selected any time. All other profiles can be freely renamed or deleted.

All changes made in the configuration mode are applied to the current profile. These changes are automatically saved. During the next creation of the Viewer window the previously created layout is loaded, therefore there is no need to configure the window layout every time the program is started.

4.5 Usage Example

To demonstrate the usage of our framework, lets assume an example. The user wants to preview two loaded models as a point cloud, a texture and a variable which holds number of renders of his application.

First step is including the library into the project and calling the initialization function at the start of the program. Then for each resource he need to send a buffer and his own description of it. It is one function calling for each of the models, and one for the texture. Finally, he uses another function to specify output string and set a pointer to the value which represent the number of renders. When he now runs an application, additional window pops up beside his application window (if there is any). This window is currently empty and there is no possibility to see anything yet. Currently there is *configuration mode* running, which means that the user can set up a layout. He creates layout which consists or each mesh, texture and variable field, and then switch to *viewing mode*. Once the mode is switched, resources can be displayed and previewed in the fields.

5 Results

To test all the features of the library we run it on an application used for model manipulation and computation. This application provides mesh processing algorithms and calculations on meshes. One specific feature of the program is the calculation of a Shape Diameter Function [7] for graphs, which can nicely demonstrate the use of the mesh and data linking in the library. The application is also working with other data, such as generated textures and variables.

In Figure 5 the tested application and the window of our Viewer running on top of it can be seen. Textures which were generated by the client application are instantly sent to the library. These textures can be displayed in the Viewer. Figure 6 shows a similar test on a different model. Both Figures show that the vertices of model are coloured in the Viewer window. These colors are visual representation of the Shape Diameter Function values, which is defining the diameter of the mesh in an individual vertex. The values were calculated by the application, sent to the Viewer as a pointer to an array and assigned to the model by its caption. Our conclusion to this test was that

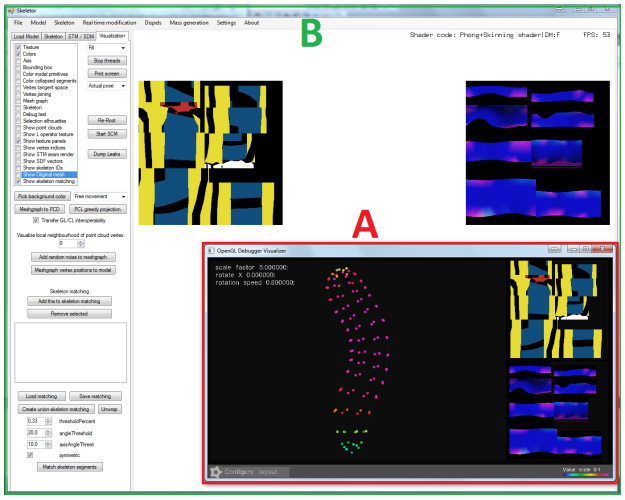

Figure 5: Viewer window (A) used beside an application (B).

the Viewer is correctly and tabularly displaying all data we sent to it.

Images indicate that the Viewer provides sufficient visual feedback for the user who is developing such an application. Without a need of the users own rendering environment, the visualization possibilities of the tool can be easily used to display several kinds of resources.

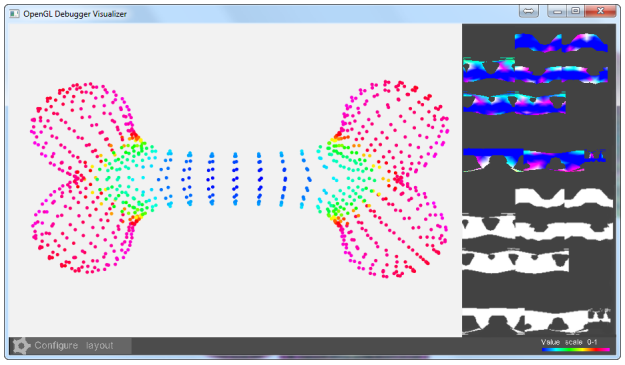

Figure 6: Model with applied values of Shape Diameter Function and its displacement and height map.

Since the tool provides multiple visual output options, there are many possible applications for it. For example it may become useful when the programmer needs to determine if some mesh/texture was loaded correctly. Using just a few commands it allows the user to have a visual feedback of a mesh or a texture, where he can inspect it and determine its correctness.

Another application can be a visual test of a shader program. For one user specified model there is a possibility to apply a shader program. The model can be rendered in the Viewer window with the shader. The user can see whether the render behaves as it should, or there are some undesirable artefacts. The user is also free to add another shader and the same model, and able to see both renders and compare them.

6 Conclusions

Our goal was to create a library which can display meshes with their supplementary data, textures and variable values on a separate window which can run beside the user's application in real-time. We reviewed similar existing tools, but since we were not able to find any solutions for exactly this type of problem, we have been inspired by existing debuggers for OpenGL, which were closest to the subject of our work.

Our library can be further expanded by adding more customization possibilities for the fields, user interface upgrades like texture zooming or functional expansions such as printing a matrix values. It can be used for viewing OpenGL and the user-program variable arrays. The library can be useful in several applications, mainly as an alternative tool for displaying OpenGL resources, debugging a program or just an auxiliary viewer running in a detached window.

References

[1] The Khronos Group. Opengl 4 reference pages. http://www.opengl.org/sdk/docs/man/, 1997-2015. [Online; accessed 19-February-2015].

[2] Graphic Remedy. gdebugger. http://www.gremedy.com/, 2004-2011. [Online; accessed 02-January-2015].

[3] NVIDIA Corporation. Nvidia nsight. http://www.nvidia.com/object/nsight.html/, 2015. [Online; accessed 15-March-2015].

[4] RAD Game Tools Valve Software. Vogl. https://github.com/ValveSoftware/vogl, 2015. [Online; accessed 15-March-2015].

[5] Bruce Merry. Bugle. https://www.opengl.org/sdk/tools/BuGLe/, 2007-2014. [Online; accessed 10-March-2015].

[6] Damian Trebilco. Glintercept. https://code.google.com/p/glintercept/, 2003-2012. [Online; accessed 10-March-2015].

[7] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Vis. Comput.*, 24(4):249–259, March 2008.