Impact of Modern OpenGL on FPS

Jan Čejka*
Supervised by: Jiří Sochor†

Faculty of Informatics
Masaryk University
Brno/ Czech Republic

Abstract

In our work we choose several old and modern features of OpenGL that applications use to render scenes and compare their impact on the rendering speed. We aim our comparison not solely on these features, but also on the type of hardware used for the measurements. We run our tests on a professional graphics card QUADRO 6000 and on a consumer graphics card GeForce GTX 580, and evaluate how actual hardware influences the results.

Keywords: OpenGL, Core profile, QUADRO, GeForce

1 Introduction

Graphics hardware is in constant evolution. New models come with new methods to solve the same problems in a more effective way. Developers now stand before a dilemma of whether to use an old method that had more time to be optimized in drivers, or a new method that better uses new technology but is not well established and may actually slow down the application when used inappropriately.

When analysing a well known graphics library OpenGL, we found that in its more than twenty years of evolution it really accumulated multiple solutions for the same problems. Considering for example rendering commands, we may use a pair of *glBegin* and *glEnd* commands and define geometry vertex by vertex, use *glDrawElements* and draw multiple primitives in a few commands, use functions such as *glMultiDrawElements* to reduce the number of rendering commands even further, or use indirect draw commands to manage rendering entirely from the GPU itself. Moreover, in case of static geometries, we still have an option to pack these commands into display lists and again reduce the number of commands that need to be processed.

To design fast applications, developers must decide which of these methods to implement. Their decision is based not only on the type of application, but also on the properties and the architecture of the hardware they use, and of course, their own experience.

*xcejka2@fi.muni.cz
†sochor@fi.muni.cz

We also had to make this decision in our application VRUT. This application is developed in a cooperation between a number of universities in the Czech Republic and the automobile company ŠKODA Auto a.s. The name stands for Virtual Reality Universal Toolkit and its purpose is to visualize detailed geometry in real-time. As such, it is highly demanding on efficiency of rendering. It is used by students as well as professionals, and therefore, it runs on various kinds of hardware, which also influences the speed of rendering.

We extended VRUT with a new rendering module and implemented several techniques that solve fundamental problems in rendering. In this paper, we describe them and compare their impact on the resulting frame rate. As this frame rate is affected by hardware, we present results of testing on two different NVIDIA graphics cards, GeForce GTX 580 and QUADRO 6000.

This paper is structured as follows. The next section presents several works that analyse modern OpenGL and its features. The third section describes methods we chose and tested in our application. Results of these tests are presented and discussed in the fourth section. The final, fifth section concludes our work and emphasizes the most important points.

2 Related work

OpenGL specification [7], located at OpenGL website *www.opengl.org*, contains detailed description of all OpenGL 4.4 functions. This website also lists all available OpenGL extensions and their description in form of plain texts. In addition to this, some extensions are also described on sites of other companies that define their own extensions; for example, NVIDIA presents at *https://developer.nvidia.com/nvidia-opengl-specs* a list of extensions that are available on many NVIDIA graphics cards.

Features of new versions of OpenGL are often presented at the SIGGRAPH conference; Lichtenbelt [3] gives us additional information about version 4.4. However, some researches focus just on a part of OpenGL. McDonald and Everitt [4] describe how techniques introduced in OpenGL 4.3 can reduce the number of functions that need to

be called to render the whole scene. Gateau [1] presents techniques developed by NVIDIA to render complex objects in only a few (possibly one) draw calls. These works unfortunately lack thorough testing and do not take different hardware into account at all.

Much information about hardware can be found on the hardware manufactures' web pages intended for developers[1,2]. Additional information is given at conferences; Kilgard [2] describes features of NVIDIA's graphics cards in association with newest versions of OpenGL. A list of main features in which QUADRO professional graphics cards and GeForce gamer cards differ can be found in [5].

3 Analysed techniques

In our work, we chose and compared several techniques that solve problems of rendering scenes. We focused on drawing commands, the type of pipeline, vertex array setup and the rendering context used.

3.1 Draw commands

The current version of OpenGL supports two main methods of drawing primitives. The first method uses functions *glBegin* and *glEnd* and defines vertices separately. The other method stores all data of these vertices in arrays, and uses functions like *glDrawArrays* and *glDrawElements* to draw them all at once. OpenGL also improves the latter method and offers functions like *glMultiDrawElements*, allowing the packing of many *glDrawElements* calls into one.

We may also use display lists in addition to these methods. These display lists allow the driver to store all commands in the most effective way and then recall them when appropriate. This method is used mainly when rendering with *glBegin* and *glEnd* as it saves many function calls, but it can be used to draw with vertex arrays as well.

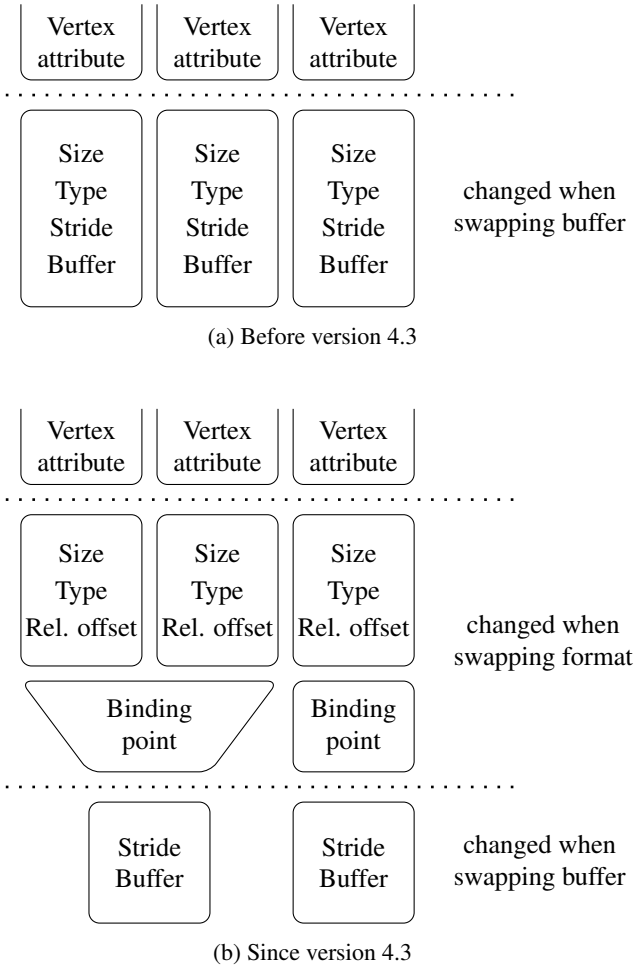
3.2 Fixed-function and programmable pipeline

OpenGL defines a set of operations that are applied to each processed primitive. These operations include transformation, lighting, texturing and many more, and form a pipeline. First versions of OpenGL defined a set of fundamental operations; to use them, programmers needed to set their parameters and activate or deactivate them if necessary. This is called fixed or fixed-function pipeline.

With time, the number of operations in pipeline increased, and so did the number of their combinations. As such, setting these parameters became impractical. Since version 2.0, OpenGL allows some parts of its pipeline to be programmed by small programs called shaders. These shaders define which operations are performed on vertices

[1] developer.nvidia.com

[2] developer.amd.com



(a) Before version 4.3

(b) Since version 4.3

Figure 1: Setting vertex array parameters before OpenGL version 4.3 and since version 4.3

and fragments[3] and manage their order. This is usually referred to as a programmable pipeline.

When we render simple scenes with simple geometries, simple lights and simple materials, we do not need functionality of the programmable pipeline, as the fixed pipeline fulfills our needs. For this reason, we do not need to write and optimize complex programs of the programmable pipeline and thus we save some time and effort. On the other hand, well written programs may save some costly state changes done in the fixed pipeline.

It is questionable which of these pipelines leads to a better performance in given situations. Most of modern graphics cards are programmable, and OpenGL's fixed functionality is programmed in the driver after all. Many modern programs (especially computer games) utilize shaders, which may lead driver programmers to dedicate less effort to optimizing drivers for programmable pipeline in comparison to fixed pipeline. On the other hand, fixed pipeline exists since the first version of OpenGL and had

[3] There are also geometry shaders which operate on primitives and tessellation shaders which subdivide them, however, our application uses only vertex and fragment shaders.

Figure 2: Scene with the Fabia car used when measuring rendering speed

more years to get optimized for applications that uses it.

3.3 Buffer setup

OpenGL 1.1 came with vertex arrays and allowed to process multiple primitives with a single call. It provided a few functions to set parameters of these arrays, that is the size and the type of vertex attributes, a buffer with these data and the stride between them, but all of them had to be set when an attribute or a buffer changed, as illustrated in Figure 1a.

For many years, this was the only method to set up buffers. In 2012, OpenGL 4.3 reviewed when vertex array parameters are set and designed a technique that decreased the amount of data set each time. It introduced binding points, which are places where buffers can be bound, and allowed us to bind buffers and set attribute parameters separately, as illustrated in Figure 1b.

In addition to this, vertex array objects (VAOs), presented in OpenGL 3.0, allow to create objects holding all information about the setup of vertex arrays. They are similar to buffer objects or texture objects, which have both existed in OpenGL for many years. VAOs do not change the way vertex arrays are set up. They only make switching between them easier.

We tested and compared both methods of setting vertex arrays and VAOs. Moreover, we also decided to test an extension *NV_vertex_buffer_unified_memory* developed by NVIDIA, available on their graphics cards. This extension is described in [6], and its main idea lies in querying the address of memory allocated by vertex buffer objects. Using this address in plane of a vertex buffer allows us to save the driver some work which speeds up the rendering.

3.4 Rendering context

The last issue we focused on is aimed at the OpenGL context. There are many parameters that are set when creating this context. We tested two of them, the type of profile and the presence of debug features, and measured their influence on the speed of rendering.

OpenGL profiles were introduced in version 3.2 as a form of removing deprecated functions. OpenGL defined two of them: core and compatibility. The core profile contains only the most modern features, while the compatibility profile includes all functions since the first version of OpenGL. As the core profile could be simpler to implement and optimise by drivers, we decided to test, whether it leads to an increase in the number of rendered frames per second.

Like many other libraries, OpenGL comes with new methods to ease debugging and development of new applications. In addition to querying OpenGL for simple errors, some implementations allow us to create a debug context, giving us an option to set up callbacks that are called every time an error occurs. Using this debug context must obviously lead to a decrease in speed of rendering, as it must handle not only these errors, but also all attached callbacks. For this reason, we decided to measure its impact on the actual speed of rendering.

4 Measurement

We measured the time our application needed to render a single frame depending on several settings.

The measured scene contained a static model of the Fabia car containing 4.6 million triangles, illustrated in Figure 2. This model was represented by a hierarchy tree

Configuration		GeForce		QUADRO	
		CPU [ms]	**GPU** [ms]	**CPU** [ms]	**GPU** [ms]
1)	**B/E**	262.662	274.890	208.861	337.591
2)	**VA**	199.759	261.961	15.776	77.987
3)	**VBO**	25.876	243.702	6.624	26.397
4)	**DL + B/E**	327.224	333.550	4.154	8.599
5)	**DL + VA**	341.850	343.046	7.368	11.481
6)	**DL + VBO**	340.475	342.296	7.403	11.480
7)	**VBO + Shaders**	27.233	26.416	27.932	27.591
8)	**DL + VBO + Shaders**	487.704	485.719	10.083	22.440
9)	7) + **VAO**	27.711	27.281	28.644	27.816
10)	7) + **MDE**	11.568	16.916	11.238	21.172
11)	7) + **Bindless**	13.499	16.717	13.973	21.260
12)	7) + **MDE + Bindless**	9.616	16.964	9.828	21.148
13)	7) + **VAO + MDE**	11.170	16.649	10.488	21.183
14)	7) + **VAO + Bindless**	14.659	16.764	14.776	21.301
15)	7) + **VAO + MDE + Bindless**	10.630	16.754	10.584	21.206
16)	7) + **MDE + Format43**	11.456	16.742	11.210	21.186
17)	7) + **VAO + MDE + Format43**	10.746	16.723	10.798	21.212
18)	**Core**	27.953	27.445	28.369	28.001
19)	12) + **Debug**	33.244	31.384	—	—
	4) + **Debug**	—	—	5.043	8.775

Table 1: Rendering times of different configurations on both tested machines

with 1344 geometry nodes, containing 1342 triangle lists and 145830 triangle strips. The model's appearance was described by 86 materials; one of them implemented a car paint effect and used its own shaders, the rest were simple enough to be rendered with the fixed pipeline. The scene was lit by a simple directional light centered at the camera (headlight).

We ran all tests on two machines. The first machine, in the rest of the paper labelled as **GeForce**, contained Intel i7 2600 processor with 8 GB of main memory and GeForce GTX 580 with the display driver version 310.70. It was chosen to represent consumer machines.

The other machine, labelled as **QUADRO**, contained two Intel Xeon X5680 processors and 24 GB of main memory. It had two QUADRO 6000 graphics cards with the display driver version 310.70. It was chosen to represent professional workstations. Although this machine contained two graphics cards, only one of them was active so that the results could be compared with the results of the first machine.

We measured a time these machines needed to render the scene in configurations described below. Since the actual rendering runs asynchronically on graphics cards, we separately measured the time needed to call all OpenGL functions (labelled as **CPU**) and the time the driver needed to execute and complete all commands (labelled as **GPU**). We measured groups of 50 frames and chose three groups with the smallest deviation. Since we focused on the maximum speed of rendering, we took the minima of measured values (in milliseconds) and presented them in Table 1.

4.1 Configurations and Results

We tested the following configurations. First, we tested the influence of draw commands used to render the geometry. We compared rendering with *glBegin* and *glEnd* functions (in the table labelled as **B/E**), rendering with vertex arrays stored at the client site (**VA**), and rendering with vertex arrays stored at the server site (**VBO**). Since all these draw commands can be stored in display lists, we made the same measurement again, this time using display lists (**DL**). The results are shown in Table 1 with configurations numbered 1 – 6.

Next, we tested how shaders influence the speed of rendering (configuration labelled as **Shaders**). The largest part of the scene could be rendered using the fixed pipeline, therefore, we compared the rendering speed when using the fixed pipeline and the programmable pipeline with shaders. These shaders simulated operations of the fixed pipeline, however, we must note that they implement per-pixel lighting. Despite the fact they are more computationally demanding than the fixed pipeline, we believe the measured values are still comparable. The results are numbered as 7 and 8.

We also measured the influence of vertex array objects (labelled as **VAO**), *glMultiDrawElements* (**MDE**), *NV_vertex_buffer_unified_memory* extension (**Bindless**), OpenGL 4.3 technique of setting vertex arrays (**Format43**) and their combinations. We chose the configuration 7), that is **VBO + Shaders**, as a starting configuration for this group of configurations. In the Table, these configurations are numbered as 9 – 17.

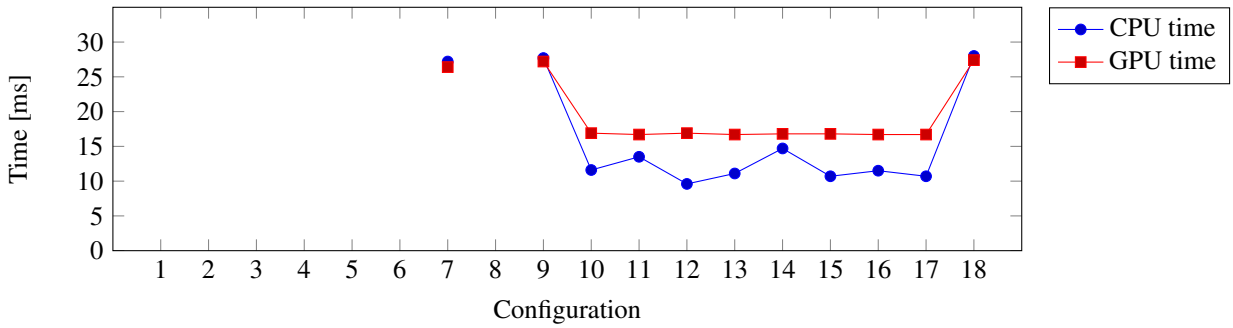


Figure 3: Rendering times of configurations achieved on **GeForce**. Times of configurations 1 – 6 and 8 are too large to display

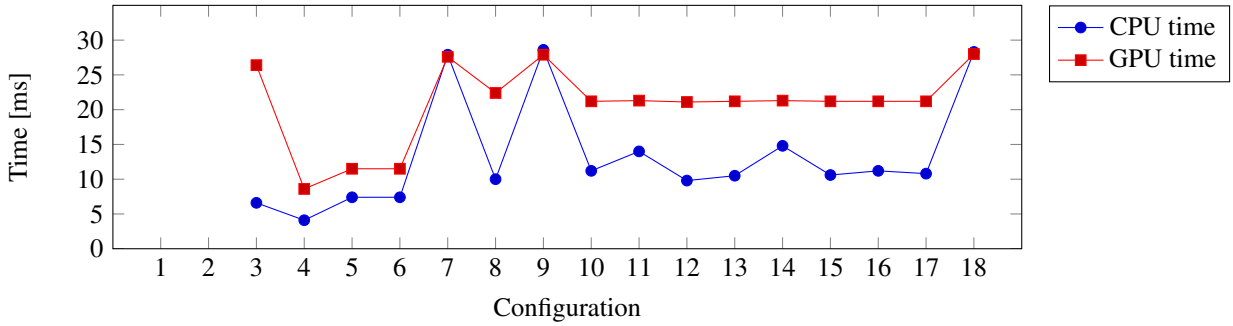


Figure 4: Rendering times of configurations achieved on **QUADRO**. Times of configurations 1 and 2 are too large to display

Finally, we tested the influence of the rendering context parameters. Rendering with the core profile (labelled as **Core**) is numbered as 18, and automatically implies **VBO + Shaders + VAO** are active. For the debug context test (labelled as **Debug** and numbered as 19), we chose as the starting configurations the best configuration on each machines, that is the configuration 12 on **GeForce** and 4 on **QUADRO**.

Figures 3 and 4 shows rendering times as line graphs for the **GeForce** machine and the **QUADRO** machine respectively. Rendering times greater than 50 ms are not shown, so that the difference in time of other configurations could be better visible. Also, configuration 19 is not present, because it differs between both machines, and the figure could lead to a misinterpretation of results.

4.2 Discussion

The measurement revealed several interesting facts. Measured rendering times in configurations 1 – 8 show, that using shaders in cooperation with vertex arrays stored at the server side in vertex buffer objects is essential for fast rendering on the **GeForce** machine. On the other hand, activating display lists led to a severe performance hit. We believe this happened due to the fact that GeForce graphics cards (as well as other consumer graphics cards) are optimized for computer games that usually do not contain static geometries and use modern features of graphics libraries, especially the programmable pipeline.

However, this was not true for the **QUADRO** machine, where display lists were the most effective way of rendering geometry. This is probably the result of driver optimizations, since display lists are a perfect solution for static geometries present in many professional applications.

Configurations 9 – 17 give us more information about the contribution of other modern techniques to the speed of rendering. Using vertex array objects showed a slight slow down when compared to configuration 7. This could have been caused by misunderstanding the role of these objects leading to an improper implementation in our rendering module, or by insufficient optimization in the driver.

Using extension *NV_vertex_buffer_unified_memory* and *glMultiDrawElements* led to an increase in speed of rendering. It is obvious that setting buffers and calling draw commands were probably the bottlenecks in our application, and these features effectively reduced their impact on the final speed of rendering. OpenGL 4.3 technique to set up vertex arrays did not result in any significant speedup. We should also mention an interesting fact, that all configurations 10 – 17 show approximately the same GPU time, but they differ in the time the application needed to call all functions.

The last two configurations show that using the core profile does not lead to any significant difference in performance (compare configurations 9 and 18). Obviously,

debugging with debug context causes a performance loss. This loss was much smaller on the **QUADRO** machine. We, therefore, assume that professional graphics cards are more optimized and therefore more suitable for debugging than consumer graphics cards.

5 Conclusion

We measured and compared the speed of rendering of a static scene in several configurations depending on techniques used and the type of hardware. We found that using modern features of OpenGL such as shaders and vertex buffer objects led to an increase in the rendering speed on NVIDIA's consumer graphics card GeForce. On the other hand, professional graphics card NVIDIA QUADRO achieved the best rendering times when we used display lists and the fixed function pipeline. Given these results, we believe neither old nor modern features are the absolute choice for better rendering performance, as this highly depends on the hardware the application runs on.

Acknowledgments

We would like to thank ŠKODA Auto a.s. for the model of the Fabia car, and also Antonín Míšek for his ideas and measuring on the **QUADRO** machine.

References

[1] Samuel Gateau. Batching for the masses: One glCall to draw them all. SIGGRAPH, 2013.

[2] Mark J. Kilgard. NVIDIA's OpenGL functionality. GPU Technology Conference, 2010.

[3] Barthold Lichtenbelt. Announcing OpenGL 4.4. SIGGRAPH, 2013.

[4] John McDonald and Cass Everitt. Beyond porting. Steam Dev Days, 2014.

[5] NVIDIA. *NVIDIA Quadro vs. GeForce GPUs: Features and Benefits*, 2003.

[6] NVIDIA. *OpenGL Bindless Extensions*, 2009.

[7] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System: A Specification*, 2013.