Comparison of 3D graphics engines for particle track visualization in the ALICE Experiment

Piotr Nowakowski[1], Julian Myrcha[1], Tomasz Trzciński[1], Łukasz Graczykowski[2] and Przemyslaw Rokita[1] for the ALICE Collaboration

[1]Institute of Computer Science, [2]Faculty of Physics
Warsaw University of Technology, Poland,
pnowakow@mion.elka.pw.edu.pl, jmy@ii.pw.edu.pl, t.trzcinski@ii.pw.edu.pl,
lukasz.graczykowski@pw.edu.pl, p.rokita@ii.pw.edu.pl

Abstract. In this paper, we examine possible ways of upgrading the 3D graphics module in the Event Display, a standalone application used to visualize the processes occurring in the ALICE experiment at CERN. This application displays a graphical representation of tracks of elementary particles as measured with the detector and recorded during proton-proton, lead-lead or proton-lead collisions. These visualizations are crucial for monitoring the condition of data acquisition and event reconstruction processes for which they heavily rely on an outdated version of the OpenGL graphics engine. In this work, we analyze the advantages and disadvantages associated with upgrading the graphics engine to a new framework, be it a new version of the OpenGL engine or Vulkan. To that end, we present an extensive comparative evaluation between the new OpenGL and Vulkan graphics libraries and draw conclusions regarding their implementation within the frames of the Event Display application.

1 Introduction

ALICE (*A Large Ion Collider Experiment*) [1] is one of the four main experiments of the LHC (*Large Hadron Collider*) [2]. Its primary goal is to study the physics of ultra-relativistic heavy-ion collisions (lead–lead (Pb–Pb) in the case of LHC) in order to measure the properties of the Quark-Gluon Plasma [3,4]. ALICE is a complex detector consisting of 18 different sub-detectors which register signals left by traversing particles. Tracking (detection of particle trajectories, also referred to as "tracks") in ALICE is performed by three sub-systems, that is the Inner Tracking System (ITS) [5], the Time Projection Chamber (TPC) [6], and the Transition Radiation Detector (TRD) [7]. The TPC, a gaseous detector extending in azimuthal plane from 0.8 m to 2.5 m from the interaction point, is the main tracking device of the experiment.

In this work, we focus on one of the most crucial software applications used in the ALICE Experiment, the Event Display. This software is capable of rendering reconstructed particle tracks on a screen (see Figure 1). These tracks are computed from measurements done by the TPC alone or both the ITS and the

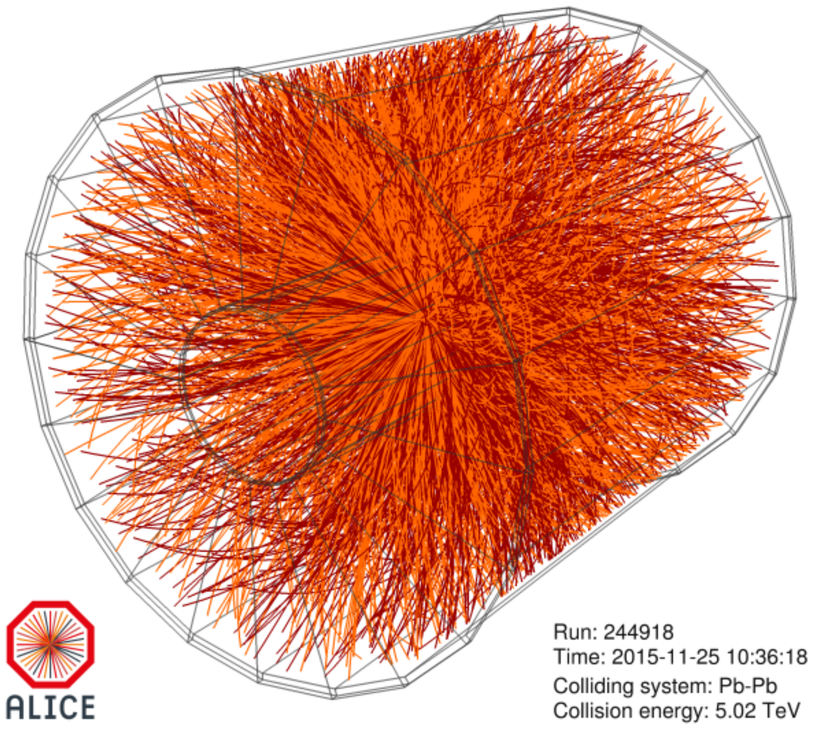


Fig. 1: Visualization of charged particle tracks in the ALICE TPC detector from the Pb–Pb collision at $\sqrt{s_{\mathrm{NN}}} = 5.02$ TeV [8].

TPC. Data for visualization can be sourced either from a database (past collisions) or from a "live" measurement. In this case data are gathered as soon as a partial reconstruction of the tracks collected during data taking is available. The second application is more important, because any hardware or software faults in the detector can lead to wrong, *i.e.* physically impossible, reconstruction results, which can easily be spotted by the monitoring team. Instantaneous error detection is crucial for the functioning of the entire system as, in case of any problems, the data collection process can be restarted thereby avoiding unnecessary corruption of the data. In this case, instead of loosing the data of an entire run of up to 38 hours, only a small portion of the corrupted data is lost.

The current implementation of the Event Display's 3D rendering module uses OpenGL 1.x API (*application programming interface*) which was released more than a decade ago and it is no longer supported by many of the recent visualization libraries. For this very reason an upgrade of the existing implementation to a new graphic engine is planned. In this paper, we investigate possible approaches of performing this upgrade. More precisely, we analyze the performances of the two most prevalent graphical APIs currently available on the market: OpenGL

4.x and Vulkan. Although alternative solutions exist *e.g.* DirectX or Metal, they are vendor specific, meaning that they offer support for a limited number of hardware configurations and we therefore do not consider them in this work. To evaluate the performances of the tested APIs, we developed a set of sample applications using both OpenGL and Vulkan, and compared their results in terms of efficiency and computational cost.

The remainder of this paper is organized in the following way. In the next section, we outline the graphics interfaces that we used in this work, highlighting their advantages and disadvantages. Next, we describe the sample applications we developed for performing experiments. Finally, we present our evaluation testbed and the results of the performed experiments. In the last section, we conclude this work by recommending one of the tested interfaces.

2 Description of Graphics Interfaces

2.1 OpenGL

OpenGL was created in 1992 by Silicon Graphics, Incorporated and is maintained to this day (in 2017 version 4.6 of the specification was released) by OpenGL ARB (*Architecture Review Board*), consisting of the biggest IT companies. In 2006 ARB was made part of a bigger organization called Khronos Group.

OpenGL [9] is based on a state machine called a *context* — global for the application list of settings which affects the way objects are displayed on screen. Ownership of the context is exclusive for a single thread of the application. This ownership can be passed on to other threads, but the context can not be accessed by two threads simultaneously. Because of that it is not possible to use capabilities of current multicore processors to speed up the object drawing itself. Additionally OpenGL functions are blocking, holding the thread execution until their task is completed — this thread will not be able to do any other computing in the meantime, while communication with the GPU is ongoing.

2.2 Vulkan

Vulkan [10] is a new graphic API released in 2016 by the Khronos Group. Vulkan is a low-level API, which provides more control of the graphics card to the programmer, but also requires from him to implement more code — he has to handle some tasks that are taken care of by the driver in OpenGL, such as memory management of the graphics card, synchronization or swapping display buffers for the operating system. Although this makes the application itself more complicated, it significantly simplifies the driver, which in turn can be more aggressively optimized by the graphics card manufacturer.

Vulkan was designed with multi-threading in mind, as it lacks a global context. Configuration is instead split into many Vulkan objects, which can be safely modified in different threads.

Rendering in Vulkan is realized by creating and filling ("recording") command buffers. Its contents are then placed in the task queue of the graphic card.

Command buffers can be rerecorded in every frame (similarly to how OpenGL operates). However, if it is not necessary (description of a particular object has not changed between frames) they can be reused (queued again), saving a lot of CPU time. Queueing of command buffers has to be performed on a single (usually main) thread, but the recording can be realized on multiple threads simultaneously. This design allows for participation of all processor cores in the drawing task, improving the overall performance.

3 Implementation

Graphics APIs provide a couple of ways that can be used to render an identical image on the screen, but varying in efficiency of GPU utilization, resulting in different performances.

Four graphics interface versions have been developed, starting from a naive implementation (for gauging performance when the code is not optimized) and ending with the most efficient implementation that we could come up with.

3.1 OpenGL

Version A treats reconstructed tracks as separate objects, thus allocating separate set of graphic card resources (vertex buffer, index buffer, color variable) for each one. Additionally, every track is drawn by a separate command. This implementation requires multiple OpenGL context alterations (mainly buffer binding) during rendering a single frame, which is costly — this should be reflected in poor performance.

Main feature of version B is a reduction of the amount of buffer binding while keeping separate drawing calls for each track. All vertex data are aggregated into a single buffer. Additionally, tracks are sorted according to the particle type, which allows configuration of the line color exactly once per group of tracks instead of for every individual track.

In previous two versions each track is drawn via an individual function call. Because a single collision is usually made of thousands of tracks, this adds up to a lot of avoidable, repetitive work for the graphic driver performed every frame. In version C a different drawing function was used, which can visualize the whole collision in a single call. It operates on an array of parameters (where every set of parameters represents a single drawing operation, like in previous versions).

Version C has reduced the number of drawing calls per frame to one, but only from the programmer's perspective — the driver still has to traverse the parameter array and dispatch drawing commands to GPU one by one. It is possible to store the parameters directly in memory buffer of the GPU and then just refer to it, avoiding most of the data transfer when a draw command is enqueued. This way of drawing is called *indirect*, which is a main feature of version D.

3.2 Vulkan

Vulkan sample programs allow to choose a track drawing strategy — with command buffers cleared and recorded for every frame (the *dynamic* version) or recorded only once (the *static* version). The *static* version takes full advantage of command buffering available in Vulkan, while the *dynamic* version tries to simulate rendering in the way of OpenGL in order to compare the two interfaces on a more equal basis.

Versions A, B and D use the same rendering techniques as their OpenGL counterparts, but implemented using the Vulkan API. Since there is no Vulkan equivalent of the drawing call used in OpenGL version C, an approach unique to Vulkan was tested here.

Version C is testing multithreading capabilities of Vulkan by utilizing *secondary* command buffers. *Secondary* command buffers can not be directly placed in the rendering queue of the graphics card, but can be executed as a part of a normal command buffer (which is called *primary*) and inherit some of the pipeline settings of its parent. Although a single command buffer (of any of the two types) can not be written to by more than one thread simultaneously, a piece of rendering work (*e.g.* an object in the virtual world) can be split into multiple secondary command buffers, recorded on multiple threads and then referenced in a single primary command buffer.

3.3 Track visualization

A single trajectory (in the ALICE track reconstruction system) is represented via a list of points describing positions in space where a particle was at a given moment. We use these data to construct composite Bézier curves that are displayed on the screen.

To calculate control points for the curves we have used two different algorithms, one created by Rob Spencer [11] (referred to as *Algorithm #1*) and the other created by John Hobby [12] (referred to as *Algorithm #2*). Curves produced by these algorithms on the same input data have a slightly different shape as shown in Figure 2.

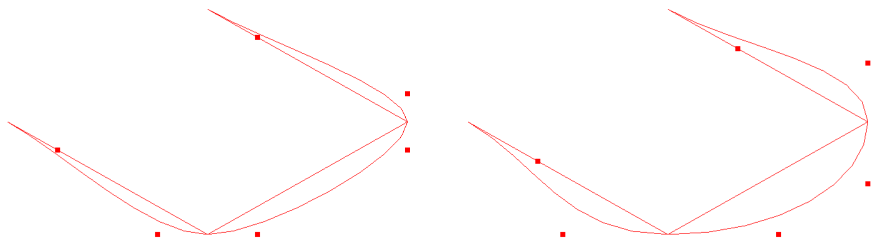
Additionally, control points can be calculated beforehand via the main processor and supplied to the graphic card or calculated on the graphic card directly. These options are labeled as *CPU* and *GPU* in the experimental results, respectively.

Taking both features mentioned above into account, there are four possible configurations of a single implementation variant. Experimental results have been grouped accordingly (see next section).

4 Experiments

4.1 Hardware

Sample programs were tested on Windows 10 with a NVIDIA 388.71 driver on two machines:



(a) Algorithm #1 (b) Algorithm #2

Fig. 2

- desktop computer — with quad-core Intel i7-4771 processor with 3.50 GHz clock, NVIDIA GeForce 780 GTX graphics card and 32 GB of RAM,
- notebook — with quad-core Intel i7-3610QM processor with 2.40 GHz clock, NVIDIA GeForce GTX 660M graphics card and 8 GB of RAM.

4.2 Performance tests

Performance of every implementation was measured by counting the number of frames rendered by the graphics card in a 10 seconds time period, determined by high-precision clock routines provided by the Windows operating system. In order to reduce the randomness of the results, each test was repeated 10 times and then averaged.

	Alg.#1, CPU	Alg.#2, CPU	Alg.#1, GPU	Alg.#2, GPU	**Average**
Variant A	63.26	64.85	64.96	61.23	**63.58**
Variant B	746.83	734.75	738.00	734.21	**738.45**
Variant C	1078.75	1059.32	1079.91	873.36	**1022.84**
Variant D	1251.56	1253.13	1233.05	997.00	**1183.69**

Table 1: Performance of OpenGL implementations on desktop computer in FPS (frames per second). “Alg. #1” is Algorithm #1, “Alg. #2” is Algorithm #2.

Table 1 presents OpenGL performance measurements gathered on the desktop computer. The biggest gain in performance (around twelve times increase in achieved frames per second) occurs between versions A and B, where the simplest optimization attempt was made (reduction of OpenGL context changes). In subsequent implementations the performance has also increased, although on a smaller scale — around 38% and 15% (if compared to the preceding version), respectively.

	Alg.#1, CPU	Alg.#2, CPU	Alg.#1, GPU	Alg.#2, GPU	**Average**
Variant A	29.21	28.95	28.36	27.88	**28.60**
Variant B	200.60	198.89	198.26	192.57	**197.58**
Variant C	224.82	226.04	225.33	213.49	**222.42**
Variant D	226.40	224.52	223.16	214.45	**222.13**

Table 2: Performance of OpenGL implementations on notebook computer in FPS (frames per second). "Alg. #1" is Algorithm #1, "Alg. #2" is Algorithm #2.

Table 2 presents OpenGL performance measurements gathered on the notebook computer. Like in the previous case, the biggest gain is achieved between implementation A and B (six fold increase). Subsequent implementations in this case, however, have not improved the performance very much — there is a small 12% increase between version B and C, while C and D are in a practical sense equal.

	Alg.#1, CPU	Alg.#2, CPU	Alg.#1, GPU	Alg.#2, GPU	**Average**
	dynamic				
Variant A	532.77	543.18	503.27	442.28	**505.38**
Variant B	1124.86	1116.07	1113.59	1108.65	**1115.79**
Variant C	1103.75	1131.22	1109.88	1101.32	**1111.54**
Variant D	1193.32	1169.59	1201.92	1182.03	**1186.72**
	static				
Variant A	1162.79	1196.17	1201.92	1204.82	**1191.43**
Variant B	1203.37	1203.37	1196.17	1184.83	**1196.94**
Variant C	1173.71	1203.37	1194.74	1191.90	**1190.93**
Variant D	1206.27	1209.19	1219.51	1197.60	**1208.15**

Table 3: Performance of Vulkan implementations on desktop computer in FPS (frames per second). "Alg. #1" is Algorithm #1, "Alg. #2" is Algorithm #2.

Table 3 presents Vulkan performance measurements gathered on the desktop computer.

Usage of the API in a style similar to OpenGL (the *dynamic* version) suffers from a similar bottleneck, as seen in the difference in performance between version A and B. However, the sample programs are running faster overall, especially the less optimized versions. As in OpenGL, a variant that uses the indirect rendering technique is the fastest. Surprisingly, the variant that uses multithreading achieve worse results than variant B.

When command buffers are not recorded for every frame (the *static* version), sample programs perform almost equally as well (with the exception of variant C, which also in this case runs slower).

Vulkan implementation, in the best case, is a little (around 2%) faster than the best implementation of OpenGL on the desktop computer.

	Alg.#1, CPU	Alg.#2, CPU	Alg.#1, GPU	Alg.#2, GPU	**Average**
	dynamic				
Variant A	157.80	158.86	156.08	156.42	**157.29**
Variant B	193.12	193.20	189.29	188.537	**191.04**
Variant C	193.65	193.46	189.00	188.75	**191.22**
Variant D	200.40	200.20	194.74	196.31	**197.91**
	static				
Variant A	201.33	201.65	196.81	196.19	**199.00**
Variant B	200.44	199.64	196.04	197.32	**198.36**
Variant C	201.94	201.41	196.81	196.58	**199.18**
Variant D	199.28	199.16	195.20	196.77	**197.60**

Table 4: Performance of Vulkan implementations on notebook computer in FPS (frames per second). "Alg. #1" is Algorithm #1, "Alg. #2" is Algorithm #2.

Table 4 presents Vulkan performance measurements gathered on the notebook computer. The results are similar to those found in Table 3. However, the difference in speed between variant A and B in the *dynamic* version is significantly smaller (on desktop machine performance gain is around 121%, while here it is only 21%) and the best implementation is not as fast as the best OpenGL implementation on the same machine.

Figure 3 presents the gathered data in a form of a graph. As can be seen, implementations that depend on repeated submission of drawing commands (*dynamic* Vulkan and all of OpenGL) achieve better performance at each optimization step. However, in the case of *static* Vulkan implementations the performance is almost constant, no matter if the code is optimized or not.

Additionally, the computer with a better GPU benefits from the optimization far more than the computer with an inferior version — every implementation with the latter (excluding "naive" version A) is running at around 180 frames per second, while with the former version the performance varies from around 60 frames per second to over 1200 frames per second.

4.3 Difficulty of usage

Difficulty of usage as a trait of programming interface is a subjective matter and as such is hard to measure, but it has an influence on the speed of application development. We tried to estimate this by counting the number of lines of code used in our implementations.

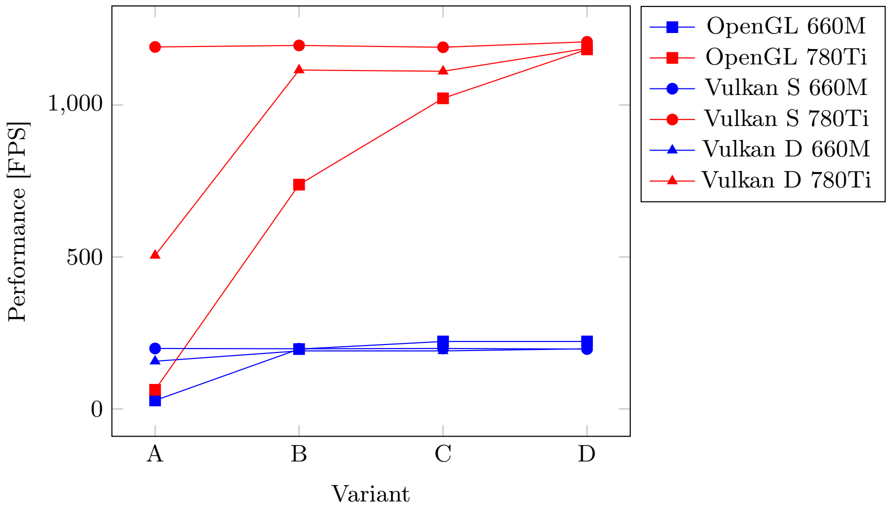


Fig. 3: Measured performance in FPS (frames per second). "Vulkan S" stands for *static* (*i.e.* command buffers recorded only once) while "Vulkan D" represents *dynamic* (*i.e.* command buffers recorded every frame).

Table 5 presents the gathered data. Only A, B and D variants are taken into account, because they use equivalent rendering techniques in both OpenGL and Vulkan. The biggest difference in length occurs in a piece of code that is shared

Module	OpenGL	Vulkan
Track loading	565	565
Common code	828	2272
RendererA	265	377
RendererB	301	441
RendererD	316	432
Sum	2275	4087

Table 5: Number of code lines.

between implementations using the same interface (which handles proper 3D initialization, shader loading, memory management, framebuffer swapping etc.). Sections containing only drawing code (labeled *RendererX*) are longer as well, but not nearly as much (around 40%). Taking everything into account, Vulkan implementation is 80% longer than OpenGL.

5 Conclusion

According to the experimental results the application performance is affected mainly by the way it was implemented and not by the chosen graphics API. The most advanced implementation (implementation D) in both OpenGL and Vulkan version on both computers achieves similar rendering speeds (difference is about 10%). Additionally neither API was found to be faster (OpenGL was faster on notebook while Vulkan on desktop). This means that there is a slight variance on how good the graphics driver performs on different hardware configurations. Therefore a simpler API should be selected, as this would reduce development time. OpenGL is in this case considered to be superior.

We have measured FPS in a scenario where only track data are displayed. If this was the only task of the Event Display, almost all compared implementations would be accepted, as rendering speeds are 3 times faster than the required 60 FPS. This is however not the case — the program has other data to display which also consumes time. Track rendering optimization provides more time for other computations.

Acknowledgements

The authors acknowledge the support from the Polish National Science Centre grant no. UMO-2016/21/D/ST6/01946.

References

1. K. Aamodt *et al.*, "The ALICE experiment at the CERN LHC," *JINST*, vol. 3, p. S08002, 2008.
2. L. Evans and P. Bryant, "LHC Machine," *JINST*, vol. 3, p. S08001, 2008.
3. E. V. Shuryak, "Quark-Gluon Plasma and Hadronic Production of Leptons, Photons and Psions," *Phys. Lett.*, vol. 78B, p. 150, 1978. [Yad. Fiz.28,796(1978)].
4. J. Adams *et al.*, "Experimental and theoretical challenges in the search for the quark gluon plasma: The STAR Collaboration's critical assessment of the evidence from RHIC collisions," *Nucl. Phys.*, vol. A757, pp. 102–183, 2005.
5. G. Dellacasa *et al.*, "ALICE: Technical Design Report of the Inner Tracking System (ITS)," 1999.
6. G. Dellacasa *et al.*, "ALICE: Technical Design Report of the Time Projection Chamber," *CERN-OPEN-2000-183, CERN-LHCC-2000-001*, 2000.
7. P. Cortese, "ALICE: Technical Design Report of the Transition Radiation Detector," 2001.
8. A. Collaboration, "ALICE event display of a Pb-Pb collision at 2.76A TeV." https://cds.cern.ch/record/2032743?ln=en, 2015.
9. G. Sellers, R. Wright, and N. Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 7th ed., 2015.
10. G. Sellers and J. Kessenich, *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Always learning, Addison Wesley, 2016.
11. R. Spencer, "Spline Interpolation." http://scaledinnovation.com/analytics/splines/aboutSplines.html, 2010.

12. J. Hobby, "Smooth, Easy to Compute Interpolating Splines," *Discrete & Computational Geometry*, pp. 123–140, June 1986.