Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

WebAssembly and Vulkan API in Image Processing Development

Robert Tornai, Péter Fürjes-Benke, László File, Dávid Miklós Nyitrai

University of Debrecen, Faculty of Informatics
tornai.robert@inf.unideb.hu
furjes.peter99@gmail.com
file.laszlo.n@gmail.com
dawnyitrai@gmail.com

Abstract

This paper will present an image processing program named BlackRoom. The software implements a lot of modern techniques like Vulkan, tile rendering, and WebAssembly.

The calculation of different effects on large size images induces huge memory consumption. With the usage of tile rendering, this memory allocation can be decreased at the cost of a slight calculation performance. The integration of the Vulkan API is also a way to increase the efficiency of applications. Beyond OpenGL, Vulkan provides low-level access to the hardware. In order to point out the difference among the applied technologies, our program has a built-in benchmark system to determine the performance of the CPU and GPU through the implemented executing branches.

A further aim is to make the program available for as many platforms as possible by paying special attention to WebAssembly. By generating WASM binary our application runs without installation in a browser.

Keywords: WebAssembly, Vulkan, tiling, image processing, benchmark.

MSC: 65D18, 68U10, 97R60

1. Introduction

BlackRoom is an image processing application developed in Qt [1, 2]. The goal was to use the most modern techniques, so we implemented the algorithms using Vulkan

API also beyond OpenGL, latter is slightly slower in compute shading. Judging by the fact that BlackRoom can be run from a browser thanks to WebAssembly, the user does not have to install it. We have checked the existing accelerated image processing libraries and found that one of the best is the progressive GPUCV library, unfortunately it was abandoned in 2010 [3]. This way, it was never developed in the direction to use it in web environment. Although, Allusse et al. enhanced the system with CUDA support, it is a proprietary technique and it is not web enabled either [4]. Furthermore, our program supports Linux, macOS, and Windows platforms because we would like to make it available for as wide user base as we can. The optimization of our image processing software yielded a better and customizable memory management regarding the memory usage of image modification algorithms. With the size of different kernels, the user can manage the memory allocation for the software by accordingly sized overlapping in tiling.

The structure of BlackRoom mainly follows the standard skeleton described in GPU Gems [5]. As a source operator, we have a load operator for processed and raw formats. Our system contains image filters. Some filters, as the Harris shutter, have additional load operators for the color channels, thus extending the simple demo structure mentioned in GPU Gems. In this way, not just linear processing paths can be accomplished. We have implemented both image view and save operator as sink operators. Similarly to the framework of Seiller et al. [6], the different GPU processing paths as GLSL and Vulkan are implemented in separate classes.

2. Categories of image processing algorithms

In image processing [7, 8], there are two kinds of algorithms, one approach does calculations based on the neighboring pixels and the other does not use the nearby pixel information, only use the current pixel.

2.1. Context-sensitive algorithms

One of the simplest context-sensitive algorithms is the median filter. This is used for noise reduction, mostly for damaged pictures. A few more context-sensitive algorithms are included in our software as edge detection, Gauss filter, or sharpening. We calculate the current pixel using the nearby pixels' information of a given size called window or kernel. In this kernel we can calculate e.g., the median of the neighboring data. However, there is a problem on the edge of the image with the pixels, where there is not enough nearby information to proceed. The simplest solution for this is to start with the first full kernel on the image and skip the ones on the edge and crop after we finish. We decided to implement a dynamic method that counts the neighbors and performs the calculations accordingly.

2.2. Context-free algorithms

A lot of context-free algorithms were implemented in BlackRoom as channel mixer, contrast, infrared conversion, negative effect, saturation, vignetting, and grayscale.

For example, grayscale is calculated by taking only just the current pixel's color information in a commonly used Red-Green-Blue set. The values have a range of [0–255]. The result will be independent of the neighbors. The weights for the RGB values can be set independently by a combined GUI element of a text field, a slider, and a value spinner. We calculate the weighted average of these values for the effect using the appropriate parameters, and this way we get a tone of gray color. It shall be used for the RGB components. If this final gray value is multiplied by vector (1.2, 1.0, 0.8) then we get a sepia brown picture.

3. Tile rendering

In the video game industry, generally in 2D games it is common to separate the game world into adjacent small portions, known as tiles. This approach helps with the memory management of the software. Apart from memory optimization, with the usage of this technique, we can describe a pathfinding method for game elements or even collision detection can be solved this way.

In image processing, we use tile rendering [9, 10] to optimize the usage of available system memory. With this approach, we can work with big file sizes with great image details. Using this method we can divide the processed image into these parts called tiles (see Figure 1). Since WebAssembly is only supported in 32 bit mode right now in Qt 5.14.0, it means that the usable memory for programs is maximum 4 GB. Having the original image and two auxiliary buffers beside the final picture, finally, they can be very huge using 16 bit color channels. By having just a tile for the auxiliary buffers, the memory consumption can be reduced a lot.

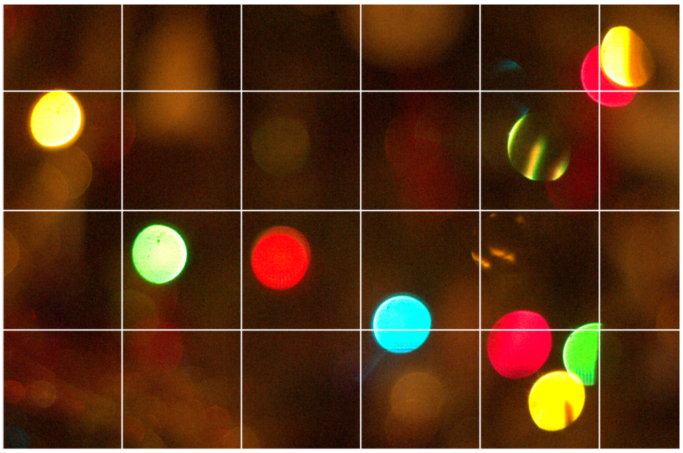


Figure 1: Basic partitioning for tile rendering

One of the advantages of tile rendering is the fact that we can calculate effects on images with bigger size than the available video or system memory. If the necessary resources do not fit in at once, the computer can use these rectangular parts of the image. The program will transport these parts of the image through the bus system of the mainboard one after another replacing the currently processed image part as the calculation goes on.

There is a problem with algorithms using the neighboring pixels on the edges of the tiles because here we also need the nearby tiles' information for the current tile's edge pixels' calculation. To solve this problem, we define an overlapping tile edge on the neighboring tiles (see Figure 2).

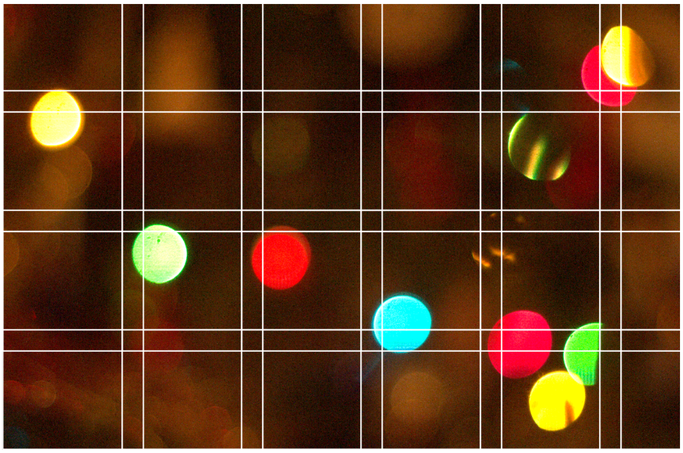


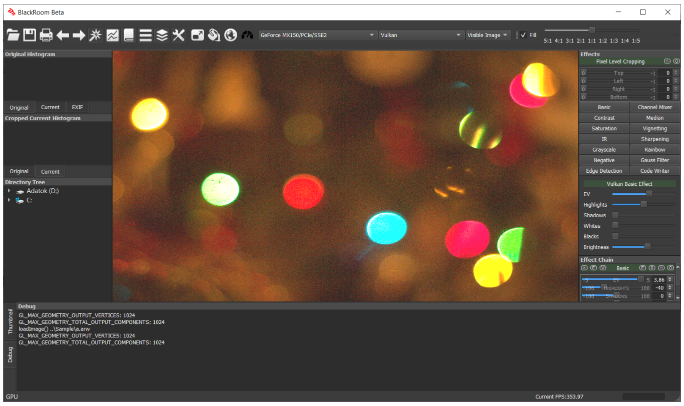
Figure 2: Overlapped partitioning for tile rendering

4. Vulkan API

Vulkan [11] is a cross-platform 3D graphics API that was released in 2016. It is based on the AMD Mantle API [12]. The main advantage of it is that it provides high-efficiency computing thanks to low-level access. What is to say that the developers have almost full control over the GPU's hardware. On the other hand, it makes this API [13] harder for beginners because we have to set up every detail from scratch. The memory management and initial frame buffer creation are fully the developer's responsibility. This feature of Vulkan gives less work to the graphics drivers.

In other APIs shaders have human-readable syntax. On the contrary, shaders in Vulkan have to be specified in bytecode, called SPIR-V. With this approach, Vulkan solves a really important problem. In the past, in order to compile OpenGL [14, 15] and Direct3D [16, 17] shaders, the GPU vendors had to write really complex compilers contained in the drivers, which resulted in that the shaders could work differently on different computers. Another great feature of Vulkan is the full support of multi-threading. This makes it possible to get better overall CPU usage. By this way, we can boost older PCs' 3D performance.

The computation in our software does not exploit the potential of Vulkan 3D capabilities. We use it only for 2D rendering to display pictures and for texturing. The Vulkan SDK provides tools for converting GLSL shaders [18] to SPIR-V format. Therefore, we can reuse our shaders written for OpenGL quite easily, but their usage is slightly different. Now our program can load in a picture and display it as a texture on the screen. The basic effects like exposure value or brightness can be set. According to our measurements with the new API, we reached approximately 20% performance increment based on the used nanosec timer. However, benchmarks of existing projects show that the difference could be up to 100%. In the future we will work on the optimization of our Vulkan implementation. On the other hand, all of our OpenGL shaders will be made available in Vulkan (see Figure 3).


BlackRoom Beta

GeForce MX150/PCIe/SSE2 · Vulkan · Visible Image · ✓ Fill

Effects: Pixel Level Cropping — Top -1 0, Left -1 0, Right -1 0, Bottom -1 0

Vulkan Basic Effect: EV, Highlights, Shadows, Whites, Blacks, Brightness

Effect Chain: Basic — EV -5 5 3.86; Highlights -100 100 -40; Shadows -100 100 0

```
GL_MAX_GEOMETRY_OUTPUT_VERTICES: 1024
GL_MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS: 1024
loadImage() ..\Sample\a.arw
GL_MAX_GEOMETRY_OUTPUT_VERTICES: 1024
GL_MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS: 1024
```

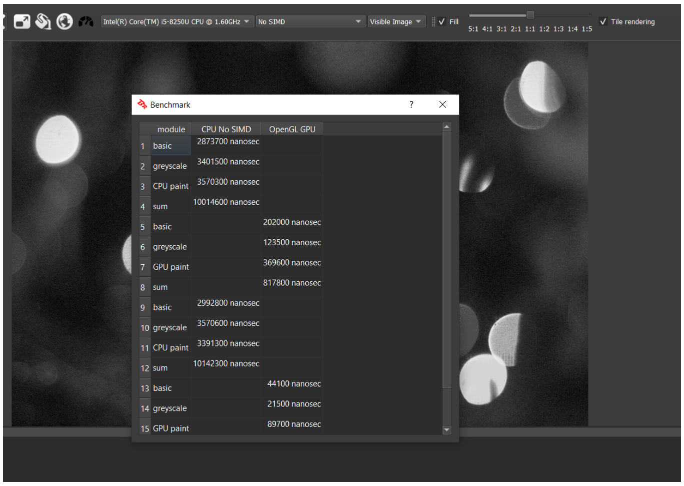
GPU · Current FPS:353.97

Figure 3: Vulkan texture

5. Benchmark system

The test was carried over a desktop machine equipped with Intel Core i5-8250U @ 1.60 GHz CPU and GeForce MX150 4 GB GPU.

A benchmark button was added to the graphical user interface. Our software implements a complex test, which can now be used to determine the performance of GPUs and CPUs per image processing modules [19]. Basically, during the benchmark the program renders the image multiple times applying the given effects using both CPU and GPU processing paths (see Figure 4). The times obtained were comparable in magnitude. Surprisingly, we have found that simple effects (maximum a few ten CPU cycle long) run faster on the CPU than on the GPU if they are alone in the effect chain. As an example, the infrared or the negative effect runs 40% faster on CPU because its calculation is quite quick. With GPU computing, transferring the image data through the buses takes more time than that we gain because of the better computation performance. From having at least two effects in the chain means that the GPU will be faster in all combinations because the transfer time become less significant in the whole process.



	module	CPU No SIMD	OpenGL GPU
1	basic	2873700 nanosec	
2	greyscale	3401500 nanosec	
3	CPU paint	3570300 nanosec	
4	sum	10014600 nanosec	
5	basic		202000 nanosec
6	greyscale		123500 nanosec
7	GPU paint		369600 nanosec
8	sum		817800 nanosec
9	basic	2992800 nanosec	
10	greyscale	3570600 nanosec	
11	CPU paint	3391300 nanosec	
12	sum	10142300 nanosec	
13	basic		44100 nanosec
14	greyscale		21500 nanosec
15	GPU paint		89700 nanosec

Figure 4: Benchmark System

Opposite to the results of Asano et al. [20], we found that even for one complex effect the GPU is faster. Maybe the reason for this, that the graphical cards evolved more during the last decade than the central processing units.

A local contrast effect is planned to be implemented without unwanted halo effects and gradient reversals along edges. A local laplacian filter suits these needs [21]. With its huge complexity, the GPU will undoubtedly have a high advantage over CPU. We plan to make comprehensive test against the results of Gkeka et al. [22].

6. WebAssembly

WebAssembly is a binary instruction format for stack-based virtual machines [23]. It is designed to be easily compiled from C/C++/Rust sources. Thanks to its binary format, it can be executed at almost native speed. To make it safe, WebAssembly enforces the browser security settings on the programs so it can protect us from malicious applications. With these properties, WebAssembly is a real alternative to the widespread JavaScript [24, 25]. It is faster and what is more, it is easier to understand. Since Qt's 5.13.0 [26] version it is fully supported, so our program can be built for this platform [27].

One of the hardest challenges was to make it possible to load pictures from local resources into the program. As our program runs on the WASM stacked-machine, originally, we could only reach the virtual machine's filesystem. We managed to solve this problem based on the solution of Morten Johan Sørvig[1]. The program basically uploads the selected picture into the virtual machine's memory, then displays it on the screen (see Figure 5). When the user finishes processing the image, he can download it onto his computer.

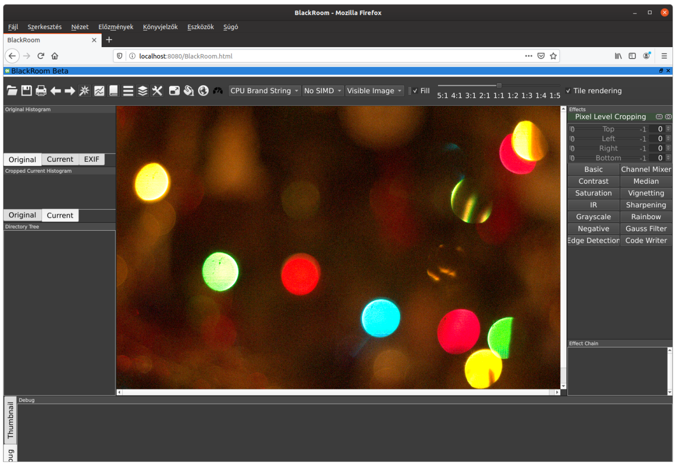


Figure 5: BlackRoom running in a browser

7. Results

We are able to build BlackRoom for WebAssembly. In this way it is possible now to use our program from a web browser. For Windows, Linux, and Android users, we incorporated the Vulkan API which is the most promising application programming interface with thread-friendliness and low-overhead. Users of macOS and iOS can enjoy the benefits through the MoltenVK layer. With the customizable memory usage, we can release our software on different devices, even having a limited amount of available memory. The controllable kernel size also makes a good test environment for the system's performance. Finally, we built a benchmark system into our application in order to illustrate the differences between the GPU and CPU rendering paths.

8. Future Work

We are planning to implement even more image effects besides more efficiency and memory tests. The Vulkan processing path can be enhanced to have almost 100% gain because real world applications suggest that. BlackRoom's multi-thread

[1]Morten Johan Sørvig solution is available here: `https://github.com/msorvig/qt-webassembly-examples/tree/master/emscripten_localfiles`, visited on 2019-10-12.

efficiency is also subpar presently in our minimum viable product, and it shall be increased to better reach the calculation capacity of the number of the threads of the CPU. The complex benchmark system will be enhanced to handle Vulkan API, as well.

Acknowledgements. This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

References

[1] Lazar, G., Penea, R., *Mastering Qt 5: Create stunning cross-platform applications*, Packt Publishing Ltd., Birmingham, England (December 15, 2016).

[2] Eng, L. Z., *Qt5 C++ GUI Programming Cookbook: Practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5*, Packt Publishing Ltd., Birmingham, England, 2nd edn. (March 27, 2019).

[3] Farrugia, J.-P., Horain, P., Guehenneux, E., Alusse, Y., GPUCV: A framework for image processing acceleration with graphics processors, in *IEEE International Conference on Multimedia and Expo*, Toronto (July, 2006), pp. 585 – 588.

[4] Allusse, Y., Horain, P., Agarwal, A., Saipriyadarshan, C., GpuCV: A GPU-Accelerated Framework for Image Processing and Computer Vision, in *Advances in Visual Computing. ISVC 2008. Lecture Notes in Computer Science*, Las Vegas (December, 2008), pp. 430–439.

[5] Jargstorff, F., A Framework for Image Processing, in R. Fernando, editor, *GPU Gems*, chap. 27, Addison-Wesley Professional, Boston, 1th edn. (April, 2004), pp. 445–467.

[6] Seiller, N., Singhal, N., Park, I. K., Object oriented framework for real-time image processing on GPU, in *Proceedings of 2010 IEEE 17th International Conference on Image Processing*, Hong Kong (September, 2010), pp. 4477–4480.

[7] Gonzalez, R. C., Woods, R. E., *Digital Image Processing*, Pearson Education Limited, London, England, 4th edn. (March 30, 2017).

[8] Nixon, M., Aguado, A., *Feature Extraction and Image Processing for Computer Vision*, Academic Press, Cambridge, Massachusetts, 4th edn. (November 18, 2019).

[9] Houston, M., Koh, W., Compression in the Graphics Pipeline:pp. 3–7, URL `http://graphics.stanford.edu/~mhouston/school/cs448a-01-fall/HoKo_compression_in_graphics_pipeline.pdf`, visited on 2020-01-22.

[10] Tile-Based Rendering, URL `https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/learn-the-basics/tile-based-rendering/single-page`, visited on 2020-01-22.

[11] Kenwright, *Introduction to Computer Graphics and the Vulkan API*, CreateSpace Independent Publishing Platform, Scotts Valley, California, 3rd edn. (October 21, 2018).

[12] Mantle (API), URL https://en.wikipedia.org/wiki/Mantle_(API), last modified on 2019-12-16, visited on 2020-01-22.

[13] Sellers, G., Kessenich, J., *Vulkan Programming Guide: The Official Guide to Learning Vulkan*, Addison-Wesley Professional, Boston, Massachusetts, 1st edn. (November 10, 2016).

[14] Kessenich, J., Sellers, G., Shreiner, D., *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, Addison-Wesley Professional, Crawfordsville, Indiana, 9th edn. (July 18, 2016).

[15] Sellers, G., Wright, R. S., Haemel, N., *OpenGL Superbible: Comprehensive Tutorial and Reference*, Addison-Wesley Professional, Crawfordsville, Indiana, 7th edn. (July 31, 2015).

[16] Luna, F., *Introduction to 3D Game Programming with DirectX 11*, Mercury Learning and Information LLC., Herdon, Virginia (February 28, 2012).

[17] Zink, J., Pettineo, M., Hoxley, J., *Practical Rendering and Computation with Direct3D 11*, A. K. Peters/CRC Press, Boca Raton, Florida, 1st edn. (July 27, 2011).

[18] Rost, R. J., Licea-Kane, B., Ginsburg, D., Kessenich, J., Lichtenbelt, B., Malan, H., Weiblen, M., *OpenGL Shading Language*, Addison-Wesley Professional, Ann Arbor, Michigan, 3rd edition edn. (July 30, 2009).

[19] Pizzini, J., GPU Rendering vs. CPU Rendering – A method to compare render times with empirical benchmarks (October 2, 2014), URL https://web.archive.org/web/20190809214146/https://blog.boxx.com/2014/10/02/gpu-rendering-vs-cpu-rendering-a-method-to-compare-render-times-with-empirical-benchmarks, visited on 2020-01-22.

[20] Asano, S., Maruyama, T., Yamaguchi, Y., Performance comparison of FPGA, GPU and CPU in image processing, in *International Conference on Field Programmable Logic and Applications*, Prague (Sept, 2009), pp. 126–131.

[21] Paris, S., Hasinoff, S. W., Kautz, J., Local Laplacian Filters: Edge-Aware Image Processing with a Laplacian Pyramid, in *Communications of the ACM*, vol. 58 (March, 2015), pp. 81–91.

[22] Gkeka, M. R., Bellas, N., Antonopoulos, C. D., Comparative Performance Analysis of Vulkan Implementations of Computational Applications, in *IWOCL'19: Proceedings of the International Workshop on OpenCL*, Association for Computing Machinery, Boston (May, 2019).

[23] Gallant, G., *WebAssembly in Action*, Manning Publications Co., Shelter Island, New York, 1st edn. (December 7, 2019).

[24] Flanagan, D., *JavaScript: The Definitive Guide: Activate Your Web Pages*, O'Reilly Media, Sebastopol, California, 6th edn. (May 13, 2011).

[25] Haverbeke, M., *Eloquent JavaScript: A Modern Introduction to Programming*, No Starch Press Inc., San Francisco, California, 3rd edn. (December 4, 2018).

[26] Qt for WebAssembly (December 16, 2019), URL https://wiki.qt.io/Qt_for_WebAssembly.

[27] Rourke, M., *Learn WebAssembly: Build web applications with native performance*

using Wasm and C/C++, Packt Publishing Ltd., Birmingham, England, 1st edn. (September 24, 2018).