EG UK Computer Graphics & Visual Computing (2022)
M. Turner and P. Vangorp (Editors)

Parallelizing Rendering on Devices with Multi-Core CPUs - Implementation Suggestion for Education

R. Porath

Lucerne University of Applied Sciences and Arts, Switzerland

Abstract

It is a well-known fact that parallelizing rendering calculations in ray tracing programs is possible and useful in many use cases, because the calculation for each pixel is often independent of the calculation of other pixels. This is also one main reason for the massive performance gain on GPUs and allows real-time rendering. However, it is often too difficult to teach students at schools and universities on how to program GPUs and parallelized rendering or it goes beyond the scope of the course. In order to still provide them a feasible way to make use of parallel rendering on their devices, be it mobile phones, tablets or PCs, we describe in this paper an implementation method, which does not require a deep IT knowledge and can be taught and applied easily. The implementation method is based on JavaScript, which became one of the easiest languages to learn programming, and is therefore often used as a great educational tool to teach and learn the basics of 3D Graphics and Rendering as well as physics, mathematics and programming. The method described in this article allows the distribution of computations to all CPU cores in modern devices, and demonstrates shorter rendering calculation times up to 70-85%.

CCS Concepts

• ***Software and its engineering*** → *Data flow architectures;* • ***Theory of computation*** → ***Parallel computing models;***

1. Introduction

JavaScript (often called JS) has undergone an incredible evolution since it was introduced in 1995 [Net95] and has become not only a professionally used programming language, but also one of the easiest languages to learn programming. Therefore, it is a great educational tool to teach and learn the basics of 3D and image rendering and thus to help students and interested other persons to get into the fascinating area of 3D Graphics, with all the physics, mathematics and programming behind it. An impressive JavaScript library called "three.js" [Thr] is a good example to demonstrate the possibilities of rendering of 3D worlds in browsers. It is based on graphics libraries like WebGL and WebGPU and can be used by students and teachers, who are less interested in learning the ray tracing basics from scratch (which is the basis for the article in hand), but prefer to develop programs for 3D objects and 3D worlds and to render them fast.

About 40 years ago, physically-based rendering as used in [PJH16] was only possible on large workstations ([Whi79], [CWVB83], [Ama84], [GP89]), if at all. Nowadays, software written in JavaScript, like the one used in this study, can demonstrate that modern internet browsers, even on a smartphone, are capable to calculate physically-based rendering images with hundreds of thousands of triangles and vertices, in a reasonable time frame of seconds to minutes. This is only possible because the modern JS engines make use of a built-in Just-In-Time (JIT) compiler, which recognizes repeatedly used code and compiles it, even though JavaScript is an interpreter programming language in general.

However, there is more potential to accelerate the rendering computation on consumer devices by either applying conceptual methods like pre-determination of bounding boxes and geometry compression for complex scenes as in [Dee95], or by applying the so-called "JavaScript WebWorkers" as described in this article at hand. JS WebWorkers are a fairly new functionality in modern JavaScript engines ([WHA10]), and give the programmers an easy way to parallelize calculations to make use of the device's Multi-Core CPU. The idea behind it is, that one main JS code executes and orchestrates other JS programs and exchanges data packages with them via a defined protocol. By this, each of those additional JS programs can run in parallel and independently and thus the whole calculation can occupy more than one CPU core.

Because of the fact that rendering an image can often be split into independent calculations without the need for large coordination or communication activities, JS WebWorkers can be applied to image rendering with great benefit.

2. Implementation to use JS WebWorkers

For the analysis in this paper, a JavaScript program was used, which is usually part of a course to teach students in developing a 3D edit-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

DOI: 10.2312/cgvc.20221166

ing and Ray Tracing app [Por21] and allows the students to learn the basics of rendering, like vector geometry, reflection, shadowing, refraction, surface gloss and brightness.

2.1. Concept to introduce WebWorkers into the JS code

The main differences between a single standalone JS program compared to a main JS program, which coordinates WebWorkers is depicted in Figure 1.

Instead of using one main JavaScript program, which may include additional functions from another file (upper part of Figure 1), it uses one main JavaScript program to start and orchestrate $1 - n$ other WebWorker JavaScript programs, which themselves may include the same additional functions file (lower part of Figure 1). Those JS WebWorkers programs are basically a copy of the original standalone JS program, with some additional lines for the data transfer.

This concept of how to introduce WebWorker as depicted in Figure 1 can easily be understood by students, who have some programming knowledge and the level to understand the basics of rendering.

2.2. Data transfer

To start the data transfer, the main JS program sends all variables and configurations to the WebWorkers, right after they were created (arrow 'a' in Figure 1). Then, the main JS program gets prepared to receive results from the WebWorkers (arrow 'b' in Figure 1) and to display these received pixel colour results.

This means, the WebWorkers get the variable and config data from the main JS code, as well as instructions on which part of the image to calculate. Calculated pixel colours are then sent back to the main JS program as intermediate or final results.

The time lag due to the data transfer between the main program and the WebWorkers was determined by taking time measurements from start of the calculations until the time when the first pixel line calculation was completed. It turned out that the difference in calculation time for the system with JS WebWorkers compared to a standalone JS program was always below 0.1s. One can therefore state, that the data transfer does not add a significant latency to the calculations, but transferring data many times can still make an impact to the overall calculation time. Therefore, the number of data transfers has to be balanced with the level of data complexity. For this reason, the WebWorkers used for the measurements presented here received their instructions only once and have returned their results only once per image line rather than for each pixel separately. From an educational point of view, to make the programming of the data transfer most understandable for students, all variables and configurations were sent to each WebWorker, even if not all variables were necessary.

2.3. Distributing the calculations among WebWorkers

When calling the WebWorkers, we divided the full image height into n parts with equal length, where n is the number of WebWorkers used. To minimize the number of data transfers, as described in previous section, the WebWorkers in our implementation calculated the pixel colours of a full image line before sending the resulting pixel colours to the main JS program to display it. Then they continued calculating the next line, see Figure 2 where four different random subsequent calculation situations are shown.

2.4. 3D rendering scene

For the measurements, we used a standard 3D object, namely a rotational ellipsoid with about 650 triangles, with index of refraction of 1.5, with part of direct light equal to 60%, part of reflection equal to 25% and part of refraction equal to 15%. Three lights were shining on this object from three different locations and with different colour, and the Ray Tracing recursion depth was set to 10. The resulting image can be seen in the right image of Figure 2. The resolution of the image was 860 x 353 px on the PC and 818 x 476 px on the Mobile Phone.

2.5. Systems used to measure the impact of the WebWorkers

Measurements were taken both on a Windows 11 PC 8-Core-CPU with Firefox v99.x and Chrome v100.x as well as on a Mobile Phone 8-Core-CPU with Firefox v99.x.

Detailed specs are:

- HP EliteDesk 800 G6 TWR with Octa-Core Intel Core i7-9700 at 3.60 GHz
- Samsung A52s 5G with Octa-Core (4 Kryo 670 at 2.4 GHz, 4 Kryo 670 at 1.9 GHz)

On those systems, the calculations were repeated at least twice to limit the effect of other processes running on the same system in the background.

3. Results and discussion

The measured times for calculating the image in Figure 2 are listed in the following Tables 1 and 2. It is visible and expected that the more JS WebWorkers are involved the shorter the time to calculate the image.

Configuration	PC Firefox v99.x		PC Chrome v100.x	
Without WeWo	495s ± 1s	≙ 1.0x	339s ± 19s	≙ 1.0x
With 1 WeWo	363s ± 3s	≙ 1.4x	230s ± 5s	≙ 1.5x
With 2 WeWo	185s ± 1s	≙ 2.7x	117s ± 3s	≙ 2.9x
With 3 WeWo	170s ± 1s	≙ 2.9x	105s ± 1s	≙ 3.2x
With 4 WeWo	130s ± 1s	≙ 3.8x	80s ± 1s	≙ 4.2x
With 5 WeWo	110s ± 1s	≙ 4.5x	70s ± 1s	≙ 4.8x
With 6 WeWo	90s ± 1s	≙ 5.5x	57s ± 3s	≙ 5.9x
With 7 WeWo	80s ± 1s	≙ 6.2x	52s ± 3s	≙ 6.5x

Table 1: *Measured calculation times and related speeding factors for rendering the image with different configurations on a PC. All measurements were repeated at least twice. (Abbr.: WeWo = WebWorker)*

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

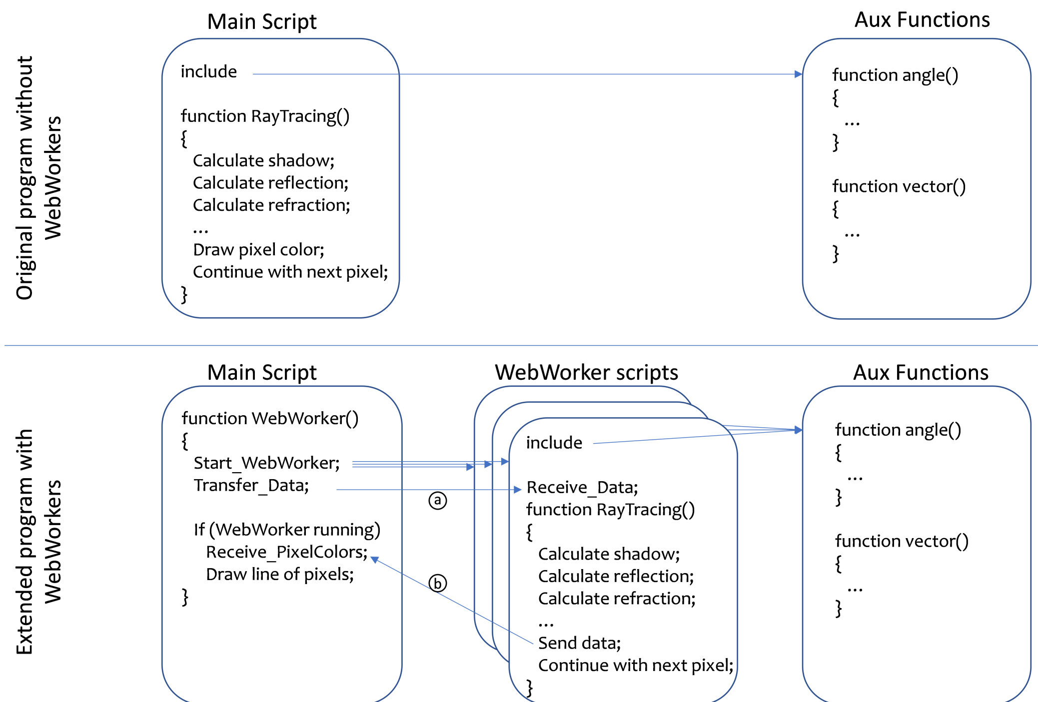

Figure 1: *Schematic overview of involved scripts. Upper part shows a standalone program. Lower part shows the main program orchestrating the additional programs called JS WebWorkers. Note: As mentioned in the text, instead of sending one pixel from the WebWorker scripts to the main script only, a whole line of pixels can be sent to reduce the number of data transfers and thus improve performance even more.*

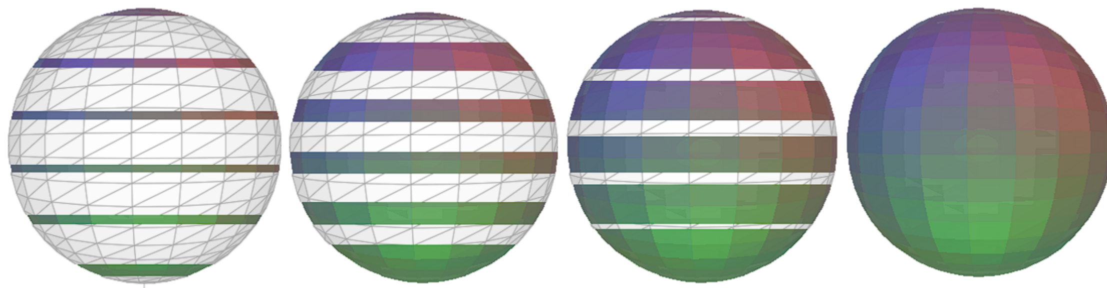

Figure 2: *WebWorkers calculate their part of the image independently. These 4 images show the calculation after different times from left to right, with the full image to the right*

3.1. Detailed results from calculations executed on the PC

If we focus first on the measured time for Chrome on the PC, we can see that the reduction in rendering calculation time when using one versus seven WebWorkers was about 77% (from 339s±19s to 52s±3s). For calculations in Firefox, we measured a reduction by a similar amount, namely about 78% (from 495s±1s to 80s±1s).

If we compare the calculation time used by one JS WebWorker with the calculation time used by a standard JS program without any WebWorker, it turns out that using even only one WebWorker is already faster by about 32% (from 339s±19s to 230s±5s) on Chrome and by about 27% (from 495s±1s to 363s±3s) for Firefox. The reason is obviously that the use of WebWorkers enables the browser and the operating system to run the calculations in parallel to other browser activities. Using all possible WebWorkers compared to the standalone JS program reduces the calculation time totally by up to 80-85%.

3.2. Detailed results from calculations executed on the Mobile Phone

For Firefox on the Mobile Phone, the reduction in rendering calculation time was about 71% (from 963s±1s to 275s±15s), which is comparable to the results from the PC measurements.

As a takeaway, one can learn that involving at least one Web-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Configuration	Mobile Phone Firefox v99.x	
Without WebWorker	963s ± 1s	≙ 1.0x
With 1 WebWorker	670s ± 90s	≙ 1.4x
With 2 WebWorkers	340s ± 1s	≙ 2.8x
With 3 WebWorkers	535s ± 15s	≙ 1.8x
With 4 WebWorkers	310s ± 1s	≙ 3.1x
With 5 WebWorkers	343s ± 3s	≙ 2.8x
With 6 WebWorkers	275s ± 5s	≙ 3.5x
With 7 WebWorkers	275s ± 15s	≙ 3.5x

Table 2: *Measured calculation times and related speeding factors for rendering the image with different configurations on a Mobile Phone. All measurements were repeated at least twice.*

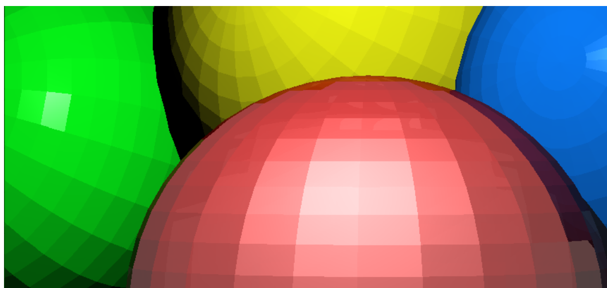

Figure 3: *3D object, which fills the whole image*

Worker is faster than running the calculations in one main JS program only. Also, the use of all possible WebWorkers, aligned to the number of available CPU cores, can make the calculations faster by a large percentage, in our cases by up to 80-85%.

3.3. Further results

Tables 1 and 2 indicate also that the calculation of images with large areas of background (like the white areas around the sphere in Figure 2), where no ray hits any object, may not always be faster with more WebWorkers (see the small differences in measured times with 2, 3, 4, 5 WebWorkers for the Mobile Phone in Table 2). The reason is that in such cases one or more WebWorkers get an area to render, which is only or mainly background and therefore, other WebWorkers have to calculate more calculation-intensive areas where objects are present.

When rendering other images however, in which less background areas are present (see an example depicted in Figure 3), all WebWorkers can contribute to the calculations of the image almost equally and the reduction of calculation time is maximized.

4. Conclusion

The education of 3D Computer Graphics and Rendering technics is possible by means of the easy to learn JavaScript programming language, and we have shown in this paper that this can be effortlessly and understandably extended by the JavaScript WebWorkers without becoming much more complicated. Basically, the same scripts can be used with only a small number of amendments to include the data transfer. This enables the students and interested persons to run their rendering calculations on their Mobile Phones, tablets or PCs with much higher performance, in our cases up to 70-85% faster, which means the waiting time shrinks from minutes or hours to seconds or few minutes.

References

[Ama84] AMANATIDES J.: Ray tracing with cones. In *Proc. SIGGRAPH '84* (1984), vol. 18, pp. 129–135. doi:10.1145/800031.808589. 1

[CWVB83] CLEARY J., WYVILL B., VATTI R., BIRTWISTLE G.: Design and analysis of a parallel ray tracing computer. In *Proc. of Graphics Interface '83* (1983), pp. 33–38. Annual Conference Series, 13-20. 1

[Dee95] DEERING M. F.: Geometry compression. In *Proc. SIGGRAPH '95* (1995). Annual Conference Series, 13-20. 1

[GP89] GREEN S., PADDON D.: Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications 9(6)* (1989), 12–26. 1

[Net95] NETSCAPE: Netscape and sun announce javascript. *PR Newswire* (Dec. 1995). https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html. 1

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016. 1

[Por21] PORATH R.: Software. https://github.com/ScienceBoy/Parallelization-of-RayTracing-with-JS-WebWorkers. 2

[Thr] Three.js 3d library. https://threejs.org. 1

[WHA10] WHATWG: Web workers. *PR Newswire* (June 2010). (Apple, Google, Mozilla, Microsoft) https://www.whatwg.org/specs/web-workers/current-work. 1

[Whi79] WHITTED T.: An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques* (1979), p. 14. 1

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.