Implementing an HTML5 mobile simulation utilizing web workers

Submitted on 04/12/2020
authored by
Lari Alakukku (528362), Miika Rouvinen (356770), Ilkka Malassu (430463)

***Index Terms*—HTML5, web, workers**

***Abstract*—HTML5 introduced web workers as a way to utilize concurrent computation in the domain of web-based JavaScript applications. This opened up the possibility to host computationally demanding applications with the web browser while attempting to maintain a high quality of experience.**

This paper follows the implementation of a computer physics simulation and compares the architectural decisions that can be applied to it. Different architectures with and without web workers are analysed and compared in terms of their performance metrics such as FPS, memory usage and message transfer time. The perceived quality of the simulation is also compared between the different versions.

I. Introduction

The internet has progressed from a distributed document sharing system to a platform that can host computationally demanding applications. This evolution makes performance a crucial web-client requirement. The modern browser provides a possibility to develop intricate web applications utilizing only HTML, CSS and JavaScript. However, the developers of a web-based game, for example, need to optimize their code taking the processing capability of different browsers into account. Current hardware mostly focuses on concurrent execution while many web applications still only use the main browser thread for all the computation tasks. HTML5 introduced web workers as a first step towards concurrent execution in web applications. [1]

Web workers operate by offloading processing tasks from the main browser thread to background worker threads, requiring the platform to have basic support for concurrency. The communication between the threads is facilitated with message sending. The process of main thread web worker communication is illustrated in Figure 1. The main thread uses the ”postMessage” function to send a message to a worker thread. The worker thread can handle this message using the ”onMessage” function. The parameters x and y of Figure 1 represent the data, that can be shared via the messaging interface. [1], [2]

Web workers also have some limitations. They can not use DOM operations or access window objects and parent objects. Also, direct data sharing between the main thread and the worker threads is not possible. JavaScript is also unable to access the underlying hardware information of a computer to, for example, fetch the number of CPU cores. [2], [4]

This paper follows and compares the architecturally different implementations of a computer physics simulation. These architectures apply different ways to implement a web-based application with and without web workers. The comparison is done with respect to performance metrics such as FPS, memory usage, message transfer time and perceived quality. After this introduction, section II of this paper covers the literature and studies related to web workers. Section III describes all the physics simulation implementations made for the purposes of this research paper. Section V evaluates and compares the performance of these implementations and Section VI concludes this paper.

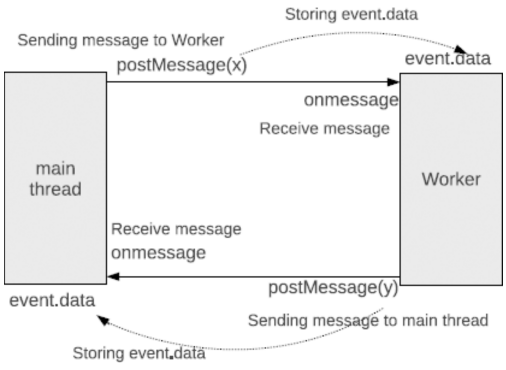

Fig. 1. Communication between the main thread and worker threads [2].

II. Related Work

For web-based games, Erbad et al. [1] propose a concurrent processing solution called DOHA, which aims to provide better control over quality by unlocking the multi-core processing capabilities. This solution consists of game event-loops running in worker threads and MultiProc, which is the module for scheduling, state management and other concurrent execution related tasks. Their game implementation had three main components: simulation, graphics rendering and AI. The main thread handles rendering and offloads game event processing to web workers. This communication also requires the sending of state information. In general, DOHA offered better scalability and responsiveness across different platforms. However, thread communication required replicating state across workers, which increased jitter.

Zhang et al. [3] introduce WWOF, which is a framework for seamlessly offloading web workers to the cloud. Their study investigated applying the framework to an interactive animation application with a substantial amount of animation events and moving objects rendered on the screen. The main thread of the application would send state data to the worker threads, which would then, after sending and receiving an acknowledgement message proceed to update this data according to the simulation rules. All this data would then be sent back to the main thread for merging and rendering. On average, the framework achieved energy savings of 85% on devices such as mobile phones, desktop computers and pads. The performance of the devices improved by a factor of 2-4.

Verdú et al. [4] examine how utilizing web workers scales the performance of a JavaScript application. Their research investigated using varying amounts of workers in a synchronous ray tracing application and in an asynchronous hash calculation task. They found that the optimal number of workers depends on various factors such as the CPU architecture, the worker execution model and the browser. Using a large number of web workers did not prove to be beneficial compared to using only a few.

III. Implementation

This section covers the different architectures of the physics simulation application implemented in this paper. The main idea of the simulation is to generate a given amount of spherical objects and simulate their interactions via a graphical interface. These objects have attributes such as position, radius, velocity and acceleration. The physics of the application includes simulating explosions and collisions between the objects. The number of these objects is changed to investigate the behaviour of the different architectures of the simulation.

Figure 2 shows the default user interface of the application running the simulation.

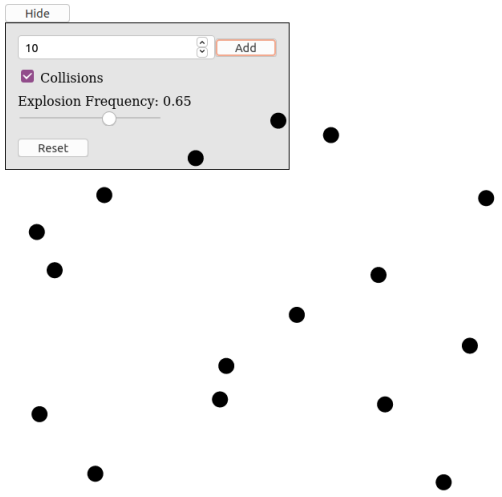

Fig. 2. User interface of the simulation application.

A. Default simulation

The first implementation of the physics simulation only utilized the main DOM thread in computation. This default architecture of the application implemented classes for the spherical objects, their interactions and the physics stage. A class utilizing the PIXI JavaScript library was implemented for managing the game loop, rendering and physics simulation objects.

B. Spatial partitioning

The first architectural decision only optimized the simulation by partitioning the physics space and it did not yet include web workers. This implementation added a grid object used in constructing the physics space in a tree data structure.

C. One worker solution

After the spatial partitioning solution, an implementation with a single web worker communicating with the main thread was made. This version initialized the

objects used in physics simulation only in the worker thread. The worker thread executed the simulation and communicated the state information to the main thread via the messaging interface. The only responsibility of the main thread was then to read the received information and render the graphics accordingly.

D. Multiple workers solution

After re-factoring the application architecture to support calculating the simulation steps in a worker thread, the number of web workers was increased. In this solution, the main DOM thread communicated with a worker planner class, that organized and divided the computational tasks to be passed to multiple workers. After all the worker threads have finished calculating their respective simulation steps, the changes to the data have to be assembled, merged and rendered in the main thread.

E. Data sharing

As stated in Section I, web workers communicate with the main thread and other threads via message sending. If data needs to be passed from one thread to another, this data needs to be serialized to JSON. This paper also investigates how message sending affects the performance of the simulation by implementing a version where the data is shared using a SharedArrayBuffer JavaScript object. This tool only supports sharing the data in a binary format, but it makes the data accessible from all the worker threads.

IV. Methods

This section describes the methods used for evaluating the different architectures of the simulation application.

V. Evaluation

Here we present our evaluation of the implementation.

VI. Conclusions

We conclude our research paper here.

References

[1] A. Erbad, N. Hutchinson, and C. Krasic, "Doha: Scalable real-time web applications through adaptive concurrent execution," Apr. 2012. DOI: 10.1145/2187836.2187859.

[2] Y. Watanabe, S. Okamoto, M. Kohana, M. Kamada, and T. Yonekura, "A parallelization of interactive animation software with web workers," in *2013 16th International Conference on Network-Based Information Systems*, 2013, pp. 448–452. DOI: 10.1109/NBiS.2013.74.

[3] J. Zhang, W. Liu, W. Zhao, X. Ma, H. Xu, X. Gong, C. Liu, and H. Yu, "A webpage offloading framework for smart devices," *Mobile Networks and Applications*, vol. 23, Jan. 2018. DOI: 10.1007/s11036-018-1009-z.

[4] J. Verdú and A. Pajuelo, "Performance scalability analysis of javascript applications with web workers," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 105–108, 2016. DOI: 10.1109/LCA.2015.2494585.