MÄLARDALEN UNIVERSITY
SCHOOL OF INNOVATION, DESIGN AND ENGINEERING
VÄSTERÅS, SWEDEN

Thesis for Degree of Bachelor in Computer Science (15 HP)

RAY TRACING IN WEBASSEMBLY, A COMPARATIVE BENCHMARK

Ludwig Johansson
Ljn18012@student.mdh.se

Examiner: Gabriele Capannini
Mälardalen University, Västerås, Sweden

Supervisor: Daniel Hedin
Mälardalen University, Västerås, Sweden

2022-02-13

Abstract

WebAssembly is a new low-level language built into the web. It's considerably faster than the currently standardly used JavaScript language. But, how fast is WebAssembly really and can it compete with native code? Previous works have tried to answer these questions. A. Haas et al. executed benchmarks comparing WebAssembly's performance to native, and found that several of them executed within 10% of the speed of native. A. Jangda, et al. did their own benchmarks, and found that WebAssembly was 55% slower than native in Chrome. In this work we want to complement these previous measurements, with measurements a previously unexplored area – graphical applications.

To carry out these measurements, we implemented our own ray tracer program. Ray tracing is a simple algorithm used to achieve photorealistic rendering, by tracing the paths of rays of light through a scene. We chose to implement our own program, rather than using an existing solution, to have full insight into the code, which would make the following parts of the work easier. The program was compiled to native code as well as WebAssembly, and measured.

Our results should be regarded as a complement to previous experiments. It is clear that WebAssembly performs differently in different types of applications. As such, it is important to test these applications separately, to see if there is room for improvement in WebAssembly, and if so, what those improvements would be.

Our results showed that the WebAssembly program reached a frame rate within 15% of the frame rate of the native program. We also started a low-level analysis of the results, using performance counters. With these, we concluded that the decrease in performance in the WebAssembly program was due to an increased number of instructions as well as branches. Due to time constraints, we have not been able to conclude the root cause of this increase in events.

Table of Contents

1. Introduction

The web has grown tremendously in recent years. As web browsers become the most common platform for serving user applications, the demands of the web platform grow. The web is no longer just being used to exchange simple documents over a network. Today we see the web being used for all kinds of sophisticated applications, such as video software, 3D vizualisations and games. JavaScript, as the only language built-in to the web, was never intended to meet these demands, and it shows. Applications written or compiled to JavaScript, perform a lot worse than their native counterpart. WebAssembly is a new low-level language built into all major web browsers, with the explicit goal of enabling safe, fast and portable code on the web. One of the goals of WebAssembly is to match the performance of native code. But how fast is WebAssembly really and can it compete with native code? Previous works have tried to answer these questions. A. Haas et al. executed benchmarks comparing WebAssembly's performance to native and found that several of them executed within 10% of the speed of native. A. Jangda, et al. did their own benchmarks, and found that WebAssembly was 55% slower than native in Chrome.

In this work we want to complement these previous measurements, with measurements in a previously unexplored area – graphical applications. We don't aim to cover the entire area of graphical applications, but rather one part of it. We still think this is a good first step, and we hope that our work inspires and enables more work in the area, by providing a simple framework for carrying out these experiments.

To carry out these measurements, we implemented our own ray tracer program. Ray tracing is a simple algorithm used to achieve photorealistic rendering, by tracing the paths of rays of light through a scene. We chose to implement our own program, rather than using an existing solution, to have full insight into the code, which would make the following parts of the work easier. The program was compiled to native code as well as WebAssembly, and measured.

Our results should be regarded as a complement to previous experiments. It is clear that WebAssembly performs differently in different types of applications. As such, it is important to test these applications separately, to see if there is room for improvement in WebAssembly, and if so, what those improvements would be.

Our results showed that the WebAssembly program reached a frame rate within 15% of the frame rate of the native program. We also started a low-level analysis of the results, using performance counters. With these, we concluded that the decrease in performance in the WebAssembly program was due to an increased number of instructions as well as branches. Due to time constraints, we have not been able to conclude the root cause of this increase in events.

1.1. Overview of the report

In the background section, we offer a more detailed description of WebAssembly, ray tracing as well as performance counters. Under related work, we have gathered a number of reports, including benchmarks of WebAssembly, works that have used performnce counters, as well as works regarding graphical applications on the web. In the problem formulation, we motivate and specify our research questions. In the merhod we describe what scientific methods we use, as well as what one needs to be cautious of when executing these methods. The method also contains an overview of the different parts of the work; the implementation, the performance measurements, and the low-level analysis, as well as what role each of these play in the work. The method is followed by a walkthrough of how we implemented the codebase. After that, we describe how we carried out the benchmarks. Finally, we talk about how we carried out the low-level analysis. Under results, we present the results from our benchmarks as well as the results of the low-level analysis. The discussion tries to interpret these results and fit them into a broader context. The most important results have been gather under *conclusions*. Under future work, we talk about all the things we would like to accomplish, but sadly could not fit into the work.

2. Background

In this section, we present the fundamental background knowledge necessary for this work, starting with WebAssembly – a low-level language for the web, a significantly faster alternative to the currently standardly used JavaScript language [1]. This is followed by an introduction to the Emscripten compiler – a compiler toolchain that enables us to compile native C/C++ code to fast WebAssembly code and run it directly in the web browser [2]. We offer a high-level explanation of the ray tracing algorithm [3], used to achieve photorealistic rendering, which will be at the core of this work. We introduce what performance counters are, and how they can serve as a useful tool to analyze the results of performance benchmarks. And finally, we introduce our performance counter toolchain of choice for this work - Perf.

2.1. WebAssembly

WebAssembly was introduced as a low-level bytecode for the web, by engineers from the four major browser vendors [1]. Before WebAssembly, JavaScript was the only language built into the web. Other technologies, such as Java applets, ActiveX and Flash, were only available through the use of plugins. Native client is another attempt of achieving near-native performance on the web, without compromising safety [4], that has been around since before WebAssembly. Native client provides sandboxed execution of x86 binary code modules and portability across operating systems, through the web browser. Asm.js is an efficient, low-level subset of JavaScript [5], that was also available at the time. One can compile code from memory-unsafe languages, such as C and C++, into asm.js, and execute them in what works like a sandboxed virtual machine. Yet the people behind WebAssembly argued that (prior to their work) safe, fast, portable and compact code on the web had yet to be achieved [1]. JavaScript (and by extension asm.js) has problems with inconsistent performance and other pitfalls. Meanwhile, Native client has its own set of problems, three prominent issues being: 1) It's inability to synchronously access JavaScript API:s, due to constraints of the chrome browser. 2) It has isssues with portability, as it builds on a particular architecture's machine code. 3) Native client is exclusive to the Chrome browser.

2.2. Emscripten

A. Zakai (ex-Mozilla employee) presented Emscripten - a compiler from LLVM (Low Level Virtual Machine) assembly to JavaScript [6]. LLVM is a compiler framework which enables analysis and transformation through all stages of a program [7]. Source code is compiled through a *frontend*, into the LLVM IR (intermediary representation), which is then compiled again through a *backend*, that generates the actual machine code [6]. Emscripten serves as a backend, compiling the IR code into JavaScript. Emscripten has since been extended with support for compiling to asm.js, and later WebAssembly [2]. Emscripten works on practically any C/C++ codebase, assuming it is portable. It has good support for the C- and C++ standard libraries and other API:s. Many times, it is sufficient to modify the definition of the main loop of the program, as well as the file handling, to be in line with the limitations of the web browser.

2.3. Ray tracing

Ray tracing is an algorithm used to achieve photorealistic rendering [3]. It is a simple algorithm that works by tracing the paths of rays of light and seeing how they interact with objects in a scene. A ray tracing implementation includes at least the following base components; a camera, ray-object intersections, light distribution, visibility, surface scattering, recursive ray tracing and ray propagation. The *camera* determines how and where from the scene is being viewed. It is responsible for generating the rays cast from our "eye" into the scene. The camera is positioned in the world and is looking in a certain direction. It has a Field of view variable, which specifies the viewing angle. *Ray-object intersection tests* tell us where exactly a given ray intersects an object. We also have to determine certain geometric properties at the intersection point, such as the surface normal. *Light distribution* is a model of the lights, in terms of their location and how they are distributed throughout space. Figure 1 shows the camera generating rays and how they interact with the objects and lights in the scene. *Visibility tests* determine if a given point is hit by any of the lights. This is trivial to do in ray tracing, as you can just generate the ray from the surface to each light and see if there any objects in the way. If no lights hit the point, it is in shadow, as demonstrated in figure 2. *Surface scattering* gives us information about how the lights interact with the objects' surfaces. This is usually expressed as how the light is scattered directly towards the camera. *Recursive ray tracing* is the process of generating additional rays starting at a surface. This is necessary because light can reach a surface by reflecting off of other objects, as seen in figure 3. Finally, *ray propagation* determines how light changes as it

travels along a ray. When the scene is rendered in a vacuum, the light remains constant. This is the assumption made by most ray tracers.

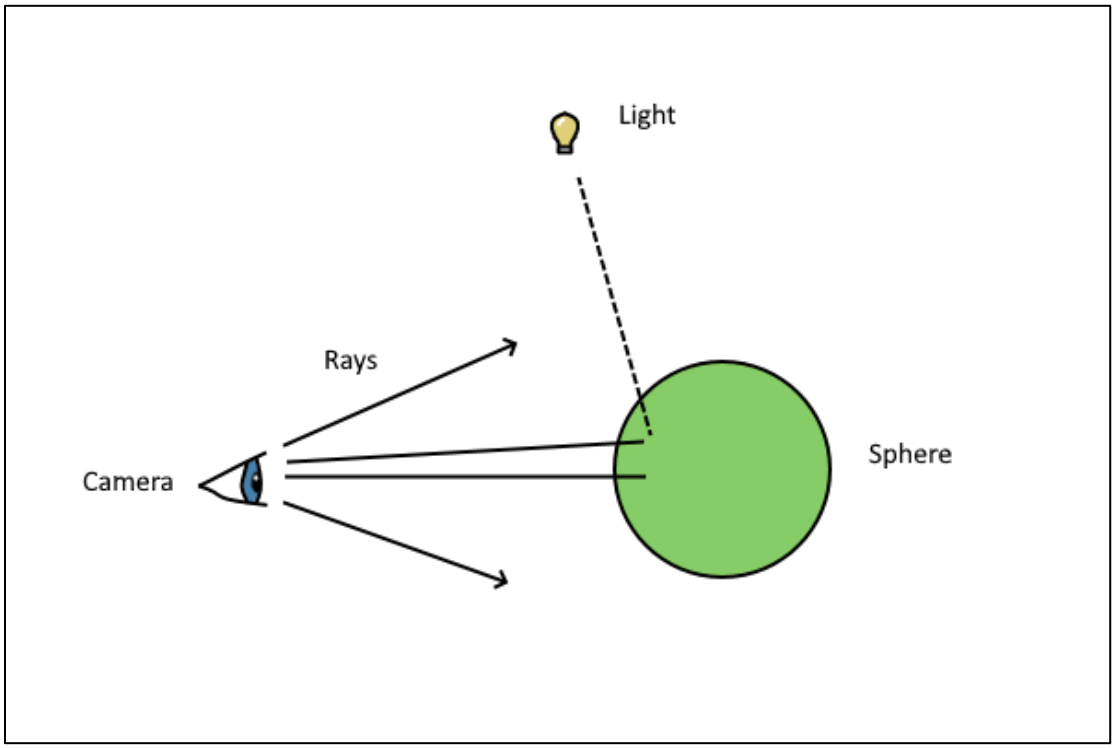

Figure 1: an overview of the main components of a ray tracer. Shows how the camera samples colors by generating rays and seeing how they interact with the scene. If a ray intersects an object we generate additional rays to the light points, to see whether the surface reflects any light.

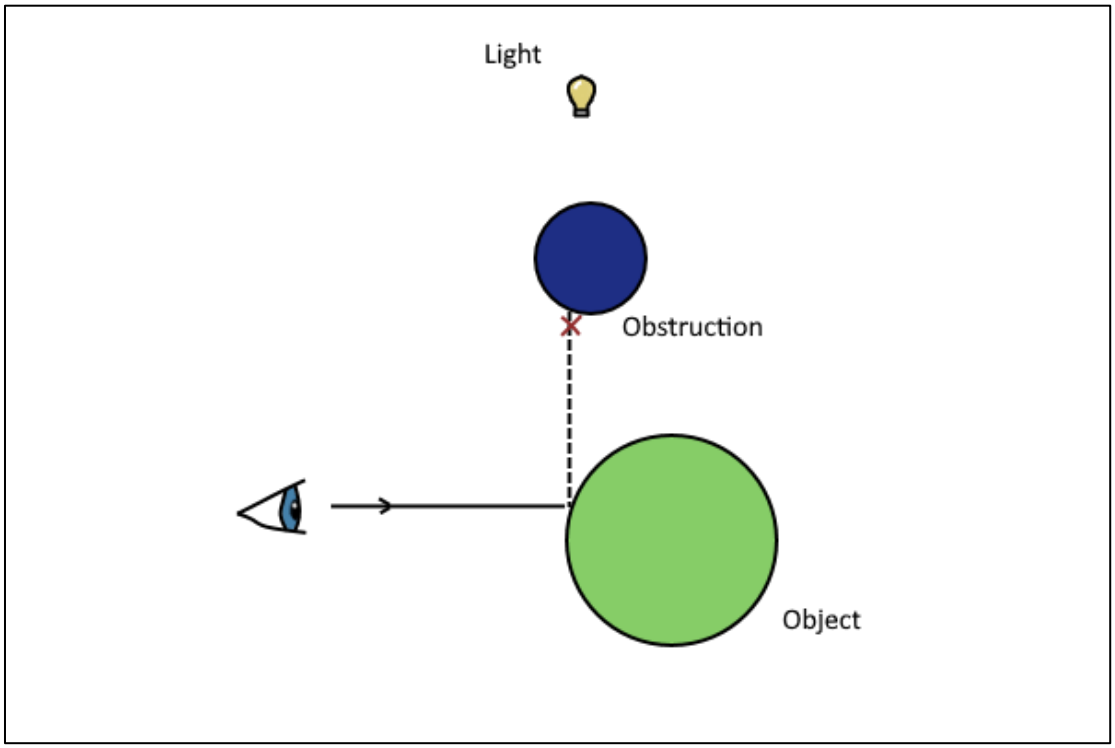

Figure 2: a visibility test in a ray tracer. A ray is generated from a point on the surface, sampled by the camera, to the only light in the scene. Because this ray is obstructed by another object in the way, the point sampled by the camera must be in shadow.

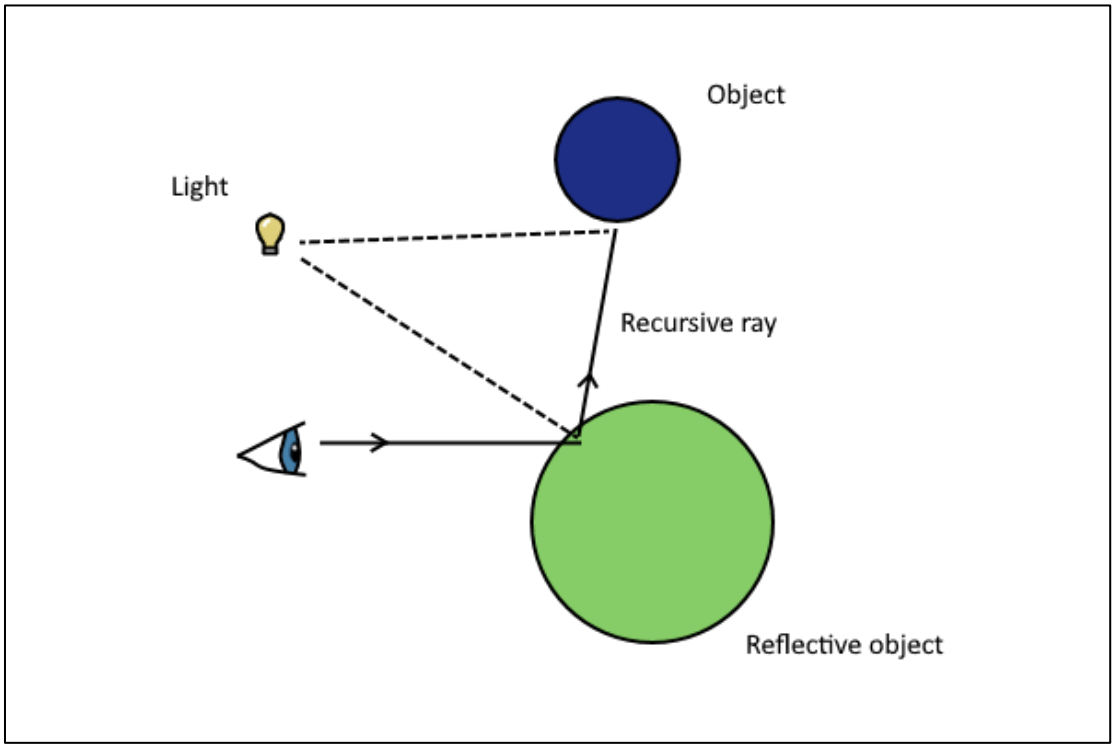

Figure 3: shows the recursive nature of a ray tracer. If a ray hits a reflective surface, such as a shiny metal, an additional ray is generated, to sample other objects being reflected.

A more technical explanation of ray tracing is beyond the scope of this work. We urge anyone who may be interested to refer to the source material [3], or any of the multiple resources available online.

2.4. Performance counters

Hardware performance counters are a set of special-purpose registers residing on the CPU [8]. These can be configured to measure performance parameters, such as instruction cycles, cache misses and branch misses. As these measurements are done at the hardware level, the overhead is low. For these counters to be of any use, we must use software to control them. The software is responsible for configuring the monitoring, starting the measurements, aggregating the counter values and finally displaying the results. A commonly used program to handle this is *Perf*.

2.5. Perf

Perf is a tool included in the Linux kernel, with powerful profiling capabilities. It can among other use cases, be used to run performance counters [9]. Perf provides a uniform command line interface, by abstracting away any CPU hardware differences [10]. Perf offers commands such as *perf stat*, which runs a provided command and gathers performance counter statistics from it. By default, perf will count the occurrences of the most common events, such as context-switches or cycles, and present them to standard output once the command finishes executing.

3. Related Work

The article that introduced WebAssembly, concluded with some performance benchmarks [1]. To this end, they ran the Poly-BenchC benchmark suite on both Chromium's V8 engine and Mozilla's SpiderMonkey engine. The results showing that WebAssembly is competitive with native code. Seven of the benchmarks executed within 10% of the native execution time, and nearly all of them within 2 times native. Furthermore, WebAssembly was 33.7% faster than asm.js on average.

The paper, *Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code* [11], argues that previous benchmarks of WebAssembly are insufficient. Previous works, such as the paper that introduced WebAssembly [1], utilize the Poly-BenchC suite of benchmarks. The problem with these benchmarks is that they only measure the performance of smaller programs (~100 LOC) rather than real applications [11]. Their aim is to instead measure the performance of WebAssembly using the SPEC CPU benchmark package. The challenge becomes compiling these benchmarks to Wasm, without modifying the code, as doing so would be a threat to validity. They accomplish this by developing an extension to BROWSIX - a JavaScript framework that enables Unix features in browser applications - called BROWSIX-WASM. BROWSIX-WASM allows them to compile Unix programs and run them in the web browser without code modifications. Their experiments conclude that, while WebAssembly is faster than JavaScript (on average 1.3 times faster), there is a considerable gap between the performance of native code and WebAssembly (1.55 times slower in Chrome and 1.45x slow in Firefox). They did a forensic analysis in order to identify the root cause of the gap in performance between native and WebAssembly. To this end, they used program counter data. They compared the data of the benchmarks running in WebAssembly, to the data of the benchmarks running natively. Attaching the performance counter to a native program is trivial but attaching it to a program running in a web browser gets a bit more complicated. In the article they accomplish this by using web workers to spawn a new process for the benchmark. They then inform another process on the computer, using HTTP requests, to attach the program counter. The benchmark program then resumes in the browser, with the program counter recording the data. They found that WebAssembly produces more loads and stores than native code (due to less available registers), and more branches (since WebAssembly requires more dynamic safety control). WebAssembly generates more instructions, which leads to more L1 instruction cache misses.

There are several other papers utilizing hardware performance counters to analyze the performance of software. One such example is the work by D. Matthew, et. Al [8]. They developed a modified version of the *perf* profiling tool, enabling access to the hardware counters from a virtual environment, running on an embedded system. They then use this tool to analyze the performance of applications running in a microkernel-based virtual environment, compared to a non-virtual environment. Furthermore, they use the tool to profile the scaling of virtual applications, by analyzing the variations in performance when an application is running in parallel with several other virtual machines on the system.

In an article S. Kang and J. Lee document how they converted a C/C++ code base for rendering 3D geospatial data into asm.js (a predecessor to WebAssembly), using Emscripten [12]. They then evaluate the FPS when rendering different areas of the map in the Emscripten approach and compare it to the FPS when using standard JavaScript. They concluded that the Emscripten approach resulted in faster rendering speeds in general.

A. Jangda, et. Al [11] made a good point about how we need more benchmarks of real applications running in WebAssembly. Although their work provided some more thoughrough evaluations, one could still argue that there is a lack of experiments on graphical applications, such as ray tracers, running in WebAssembly. S. Kang and J. Lee's work [12] shows promising results but was done at a time before Emscripten supported WebAssembly, and it doesn't really tell us anything about how the web application performs in comparison to the native application.

4. Problem Formulation

As mentioned, there is a lack of experiments on graphical applications running in WebAssembly. Therefore, the purpose of our work is to start filling these gaps by evaluating the performance of a ray tracer running in WebAssembly, in comparison to the same application running natively. Naturally, the ray tracer isn't representative of the entire area of graphical applications, instead we regard our work as a steppingstone to more thorough evaluations. If the WebAssembly application is significantly slower, we would like to see if we can identify the cause of this, much like A. Jangda, et. Al. did [11] and see what can be done to minimize said difference.

The questions we hope to answer are as follows:

- How does a ray tracer in WebAssembly perform compared to the equivalent program running natively?
- Assuming a significant difference in speed, what is missing in WebAssembly to cause this?

5. Method

To answer our first question about how a WebAssembly implementation of a ray tracer performs compared to a native implementation, we carried out an empirical study. To measure the performance in a valid way, we had to acquire two functionally equivalent versions of the same program, one running natively and one running in WebAssembly. To make sure we measured exactly what we were interested in, we had to make sure both versions of the program were doing the exact same work, on the same system. This study would form the first two tasks of our work: 1. *implementing the codebase*, and 2. the *performance measurements.*

The second question about what WebAssembly is missing to close the gap in performance, had to be broken up into smaller problems. Firstly, we would have to measure detailed data from our programs, in another empirical study. This data would tell us what events the programs were triggering at a low level, which would allow us to reason about what was causing the WebAssembly version to take more time. Ideally, we would then be able to confirm our hypotheses, but that is beyond the scope of our work. This study makes the third task of our work, the *low-level analysis*.

5.1. Implementing the codebase

We settled on writing our own implementation, as opposed to using an existing open-source codebase, to make the subsequent parts easier. We wanted to have a full insight into the code, in case we ran into any problems in the subsequent parts of the work. Say, for example, there were errors when compiling the code to WebAssembly (due to, perhaps features incompatible with our compiler of choice), it would be easier to identify these errors and correct them. Having full insight into the program would also make it easier to analyze the results.

5.2. Performance measurements

For the second task of measuring the performance of both programs, several preparations had to be made. First, necessary changes had to be made to the code, to allow us to compile it to WebAssembly. We practiced caution to only make the minimal changes necessary for compilation and nothing else, as not to threaten the validity of the results. We then wrote our benchmarks, which simply measured the time to render each frame of the 3D scene. The code base was compiled to a native program and a WebAssembly application respectively, with proper configuration. Finally, we could run both programs and compile the results of the benchmarks into graphs, to get a clear idea of the performance differences. To account for temporary variations in performance when running the programs, we sampled the render time of several frames, before calculating an average.

5.3. Low-level analysis

We used performance counters for the low-level analysis of the program. The performance counters were used to count the total occurrences of common low-level events. Attaching the performance counter to the native version of the program was trivial, but the WebAssembly version had us take a few extra steps. Due to the nature of WebAssembly, the program is constrained to a web browser, and as such, not its own process that we can control. To get around this, we implemented a simple server named *pf-server*, to handle the performance measurements of the WebAssembly application. Pf-server would listen to a signal from the web application, that tells it to start the measurements. The server would identify the id of the isolated thread the web application was running in and attach the performance counter to it.

6. Implementing the codebase

To be able to perform the benchmarks, we began by implementing our own simple ray tracer program. Figure 4 shows what the finished program looked like.

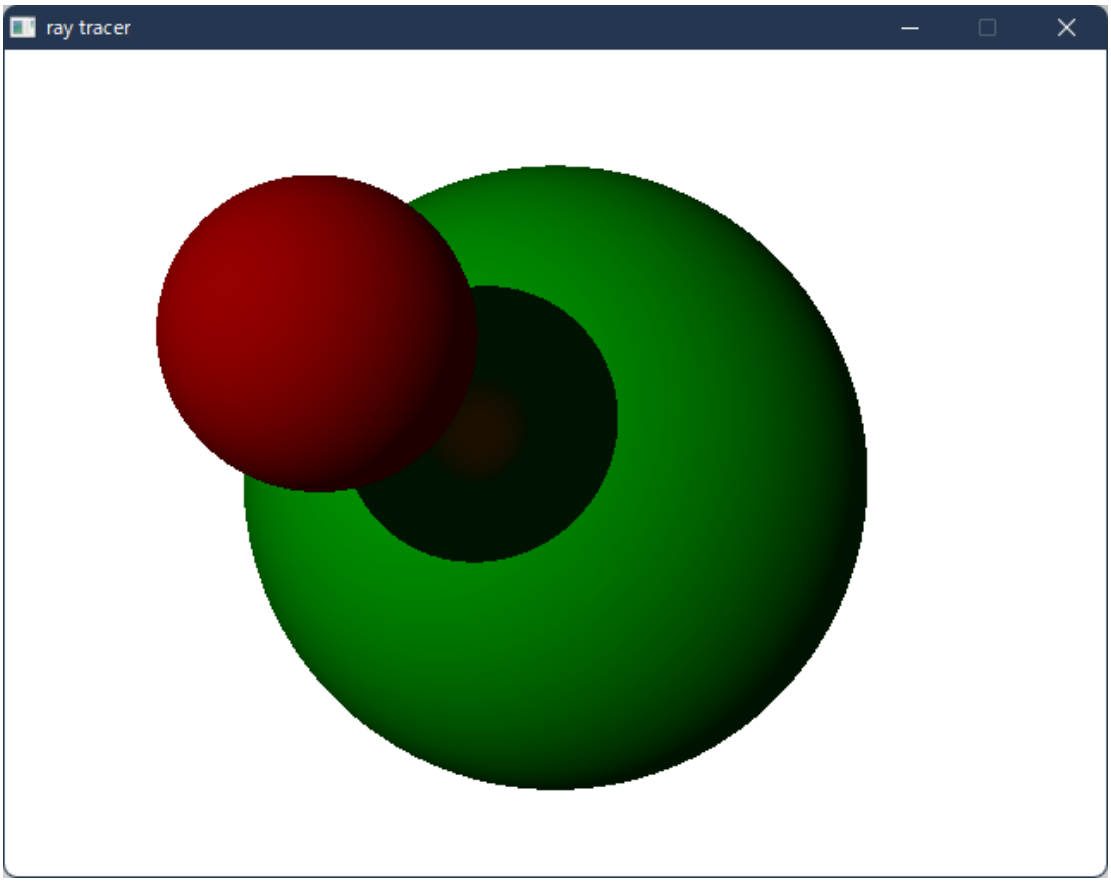

Figure 4: the fnished ray tracing program running natively. Shows two spheres being rendered, with light cast on them. The front-most sphere can be seen reflected in the back most-sphere, as well as casting a shadow on it.

The program implements a camera, ray-object intersections, light distribution, visibility tests, surface scattering and recursive ray tracing. The code was written in C++. The SDL2 library [13] was utilized to manage the window, and for facilitating manual configuration of the color of each pixel on the screen. The Eigen library [14] was utilized for vector algebraic calculations.

At the start of the program, a window is created using SDL. An array of three bytes per pixel is allocated. These three bytes represent the red, green and blue color contributions of that pixel respectively. From this array, we then create an SDL surface.

```
auto *pixels = new Uint8[SCREEN_WIDTH*SCALE * SCREEN_HEIGHT*SCALE * 3];


surface = SDL_CreateRGBSurfaceFrom(pixels, SCREEN_WIDTH*SCALE, SCREEN_HEIGHT*SCALE,
                                   24,
                                   SCREEN_WIDTH*SCALE * 3,
                                   0x0000FF,
                                   0x00FF00,
                                   0xFF0000,
                                   0);
```

We then create the scene structure, including the camera, two spheres, and a light point;

```
scene = new Scene(
        Camera(SCREEN_WIDTH, SCREEN_HEIGHT, 45,
            Vector3f(0.0, 3.0, 0.0),
            Vector3f(0.0, 1.8, 10)),
        vector<Sphere> {
            Sphere(
                    Vector3f(0, 3.5, -3),
                    Vector3f(0, 150, 0),
                    0.2, 0.7, 0.1, 3),
            Sphere(
                    Vector3f(-1.5, 2.0, 1.5),
                    Vector3f(150, 0, 0),
                    0.05, 0.75, 0.2, 1) },
        vector<Light> {
            Light(Vector3f(-10, -5, 20))
        });
```

At this point, we start the main loop of the progam, which renders the scene at an uncapped frame rate. The backing array of the surface is passed to the scene object's render method, responsible for calculating the color of each pixel. A texture is created from the surface, which is then drawn to the screen.

```
auto *pixels = (Uint8 *) surface->pixels;
scene->render(pixels);

SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);

SDL_RenderClear(renderer);
SDL_RenderCopy(renderer, texture, nullptr, nullptr);
SDL_RenderPresent(renderer);

SDL_DestroyTexture(texture);
```

The scene's render method is largely composed of vector operations. We calculate a vector pointing straight up from the camera, and another vector perpendicular to it;

```
Vector3f eye = camera.getDirection() - camera.getPoint();
eye.normalize();

Vector3f vpRight = eye.cross(Vector3f(0, 1.0, 0)).normalized();
Vector3f vpUp = vpRight.cross(eye).normalized();
```

We then loop over each pixel on the screen and calculate a vector pointing out of that pixel.

```
for (int x = 0; x < camera.getWidth(); x++) {
    for (int y = 0; y < camera.getHeight(); y++) {
        Vector3f xcomp = vpRight * (x * pixelWidth - halfWidth);
        Vector3f ycomp = vpUp * (y * pixelHeight - halfHeight);

        Vector3f rayDirection = eye + xcomp + ycomp;
        rayDirection.normalize();

        ...
    }
}
```

This vector is then passed to our trace function, which is responsible for sampling the color for that pixel. This part was largely inspired by the open-source literate-raytracer project [15]. We urge anyone who is interested in how the trace function was implemented to refer to that.

The full source code of our ray tracer implementation is available on our GitHub repository [16]. In the same repository, you can also find the build scripts we used to build the program binaries.

7. Performance measurements

The performance was measured in two distinct stages. In the first stage the program binaries were built as-is, and their resulting execution speeds were recorded for the sake of comparison. In the second stage, we increased the workload of the program and re-did the measurements. The aim was to compare the results of the two, to see if we could establish a ratio between the execution speed of the native program and the WebAssembly program, regardless of the workload. I.e., whether the execution speed grew linearly with the workload.

7.1. Initial performance measurements

The program was built and measured natively, to have a reference point to compare the performance of the WebAssembly program to. The native program was built using the CMake build system, and the Clang compiler. The Emscripten compiler was used to compile the code to WebAssembly. Both Clang and Emscripten work in tandem with LLVM, making them a suitable choice for our experiments. The code was compiled with optimization level 2, which includes most optimizations, [17], including vectorization [18]. Optimization level 2 in Emscripten is similar to that of Clang, but optimizing WebAssembly includes some additional types of optimizations. We also compiled the code to Javascript as another point of reference. The web application was run in the latest version of Google chrome at the time of writing this report, Version 96.0.4664.45 (Official Build) (64-bit). All measurements were run on Linux Mint 20.2 Cinnamon on an *Intel Core i5-8250U* processor.

The performance was measured in terms of the time to render each frame, running over 100 frames. The render time of the first 100 frames of the program running natively, in WebAssembly as well as in Javascript, is seen in figure 5.

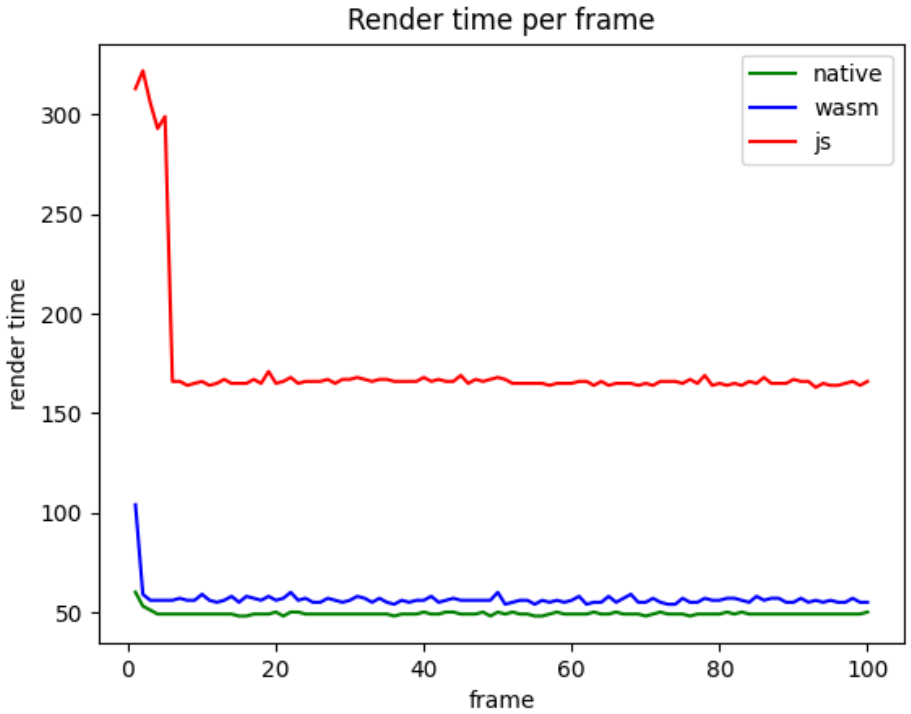

Figure 5: a comparison of the render time of the first 100 frames in the native, Wasm and JS versions of the program.

The results show that the WebAssembly version takes roughly 57 ms to render each frame, whereas the native version does it in roughly 49 ms. The Javascript version is significantly slower than the other two, taking nearly 173 ms per frame. At first glance this may suggest that the WebAssembly version is nearly competitive with the native version, taking merely 8 ms longer per frame, than the native version. How ever, if we think in terms of percentages, this is almost 15% slower than the native version, which depending on the workload could be a significant amount of time.

7.2. Complementary performance measurements

To see how the performance was affected when increasing the workload, further development was done to the codebase. We figured that a trivial feature to implement that would increase the workload almost linearly would be anti-aliasing. Rather than generating one ray per pixel, 4 rays were generated per pixel, and the sampled colors were interpolated to get the final color of the pixel. In theory this would increase the workload roughly fourfold. The resulting program is seen in figure 6. The edges of the spheres are considerably smoother than in the previous version of the program.

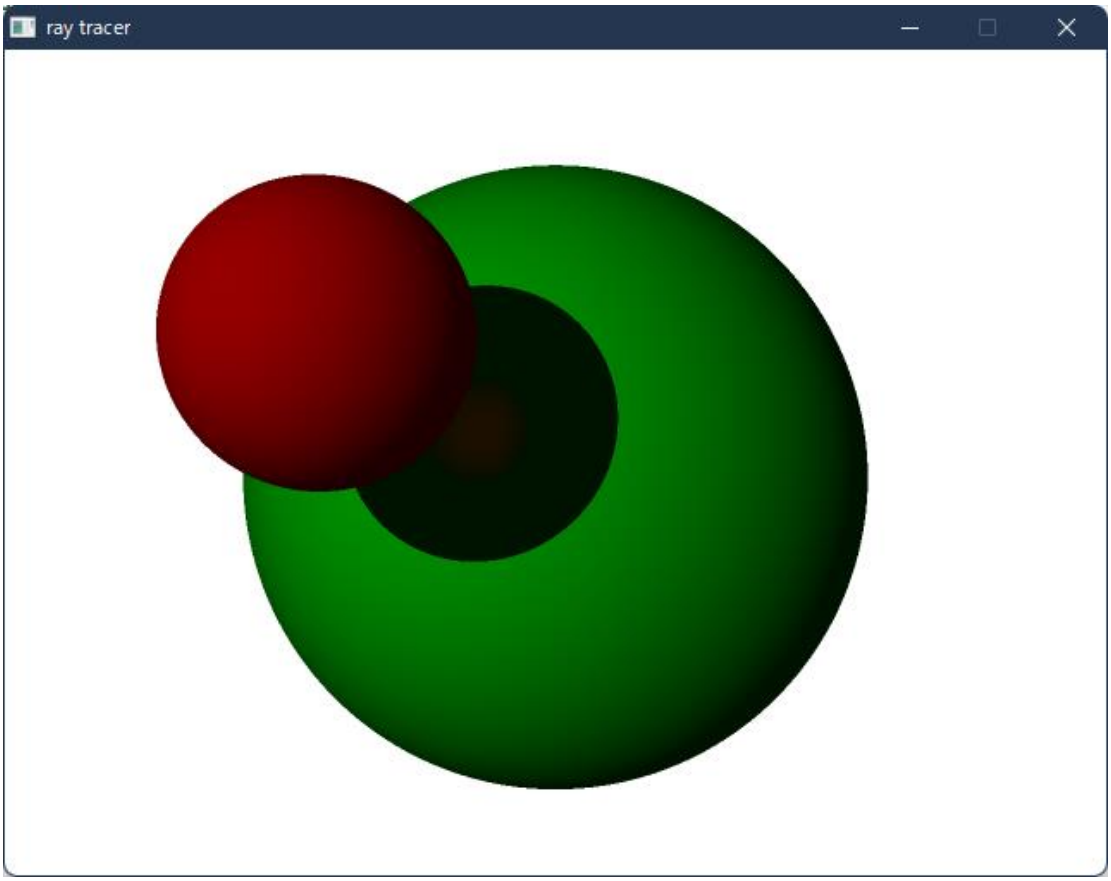

Figure 6: the ray tracing program with anti-aliasing implemented. The edges of the spheres are significantly smoother than in the previous version of the program.

Just like before, we measured the performance in terms of the time to render each frame. The results can be seen in figure 7.

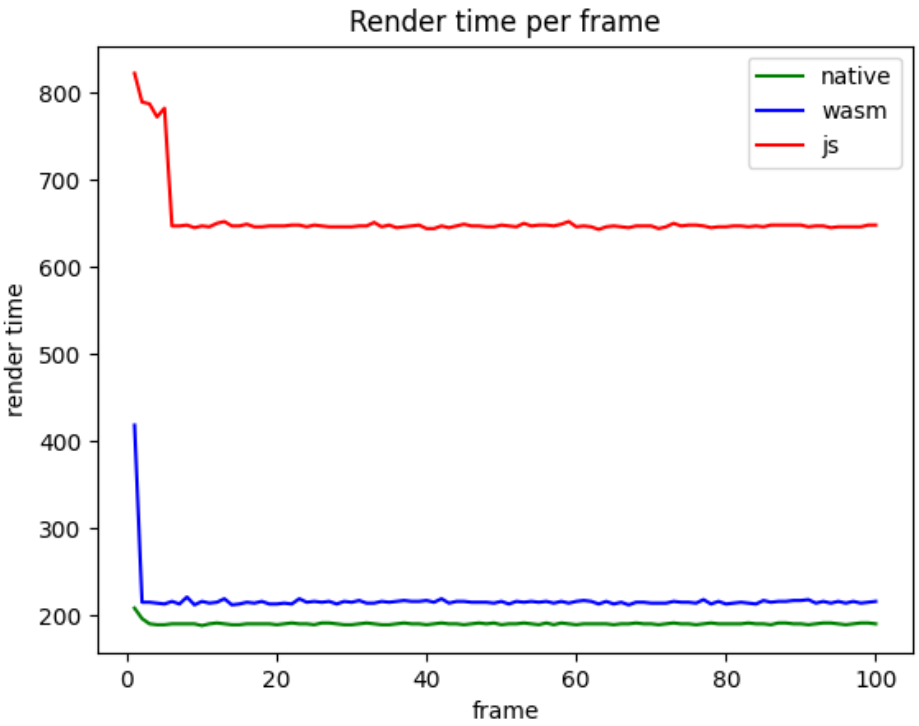

Figure 7: a comparison of the render time of the first 100 frames in the native, Wasm and JS versions of the program, after anti-aliasing was implemented.

This time, the results show that the WebAssembly version takes roughly 218 ms to render each frame, while the native version takes roughly 191 ms. Like we predicted, this version of the program is almost four times slower. Just like before, the WebAssembly version is about 15% slower than the native version. As such, we concluded that the WebAssembly version of the program is significantly slower than the native version, and we followed up with further experiments to be able to diagnose why.

8. Low level analysis

To perform a low-level analysis of the program, we wanted to attach performance counters to the program binaries. We chose to use the Perf performance counter, available on Linux systems, due to its popularity and flexibility. We used the defaults settings of Perf, which measure the exact count of some of the most common events. Attaching Perf to the native program was as simple as telling Perf where to find the executable. Perf would start the program and record the events. Attaching Perf to the web application proved more challenging.

We made use of Perf's ability to attach to a running thread. We supplied the *proxy-to-worker* flag to Emscripten, making the main application code run in a web worker, on an isolated thread. The ray tracer program would signal another processing running on the system, acting as a server, when it was time to attach the performance counter. The server would identify the id of the web worker thread and attach Perf to it. The server would then let the client know the performance counter was successfully attached. This in turn caused the client to start rendering the scene, while Perf was collecting the data.

The server was written in JavaScript for Node.js [19]. It was implemented as a web API, using the Express framework [20]. The API exposes two endpoints; '*/start*' as well as '*/stop*'. Calling '/start' would search for the thread with the name 'DedicatedWorker', on the same machine. Assuming we found such a thread, it would then attach Perf to it and send a message back to the client, letting it know it was a success.

```
app.get('/start', async (req, res) => {
    const tid = await findThread('Dedicat').catch(e => {
        console.log(e);
    });

    if(tid) {
        console.log(tid);
        startPerf(tid);
        res.send(`Start performance counter on thread ${tid}`);
    } else {
        res.statusCode = 500;
        res.send('Error starting performance counter');
    }
});
```

The find thread function would try to find an isolated thread with the specified name, by executing the '*top*' command in Linux. If it found more than one possible candidate, it rejected the caller.

```
exports.findThread = (searchString) => {
    const query = `top -H -b -n 1 | grep ${searchString}`;

    const promise = new Promise((resolve, reject) => {
        exec(query, (error, stdout, stderr) => {
            const rows = stdout.split('¥n');
            rows.pop();

            if(rows.length == 0) {
                reject("Can't find thread")
            }

            if(rows.length > 1) {
                reject(`Can't isolate thread, found ${rows.length} candidates¥n${rows}`)
            }

            const row = rows[0].trim().split(' ');
            resolve(row[0]);
        });
    });
    return promise;
}
```

We won't go into the details of the '*/stop*' endpoint, or the methods to start/stop Perf as they are quite trivial. The full source code of the server is again available on our GitHub repository [16].

9. Results

This section presents the results of our studies. First, we will present the results of the performance benchmarks. Later, we will present the results of the performance counters.

9.1. Performance benchmarks

After measuring the render time of our program compiled to native and WebAssembly respectively, we calculated an average of our samples and compiled a graph of the render time expressed as FPS (frames per second). Figure 8 shows the graph of the program with anti-aliasing disabled, and figure 9 shows it with anti-aliasing enabled (close to 4x the workload).

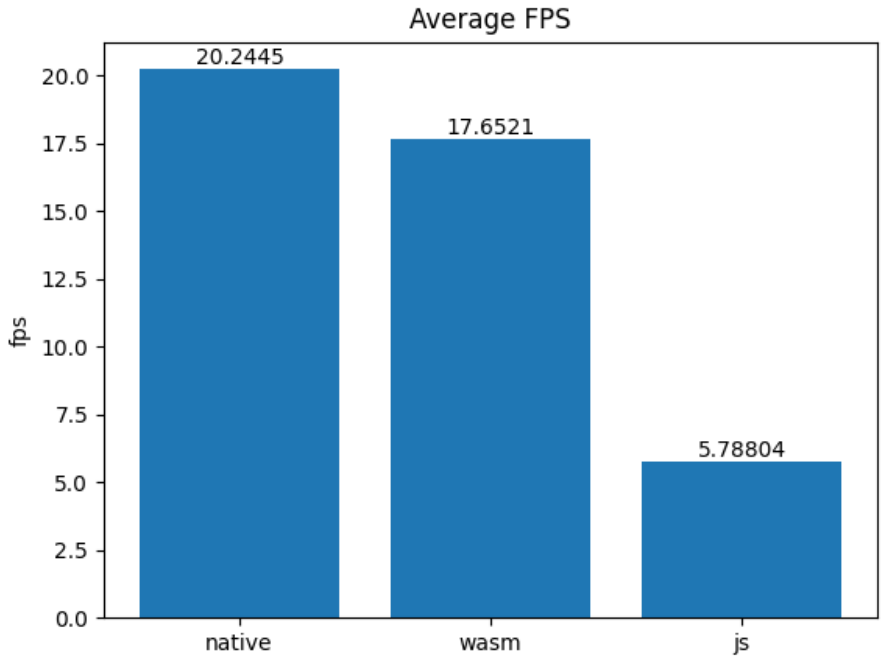

Figure 8: the average FPS of the program with anti-aliasing disabled, running natively, in WebAssembly, as well as JavaScript.

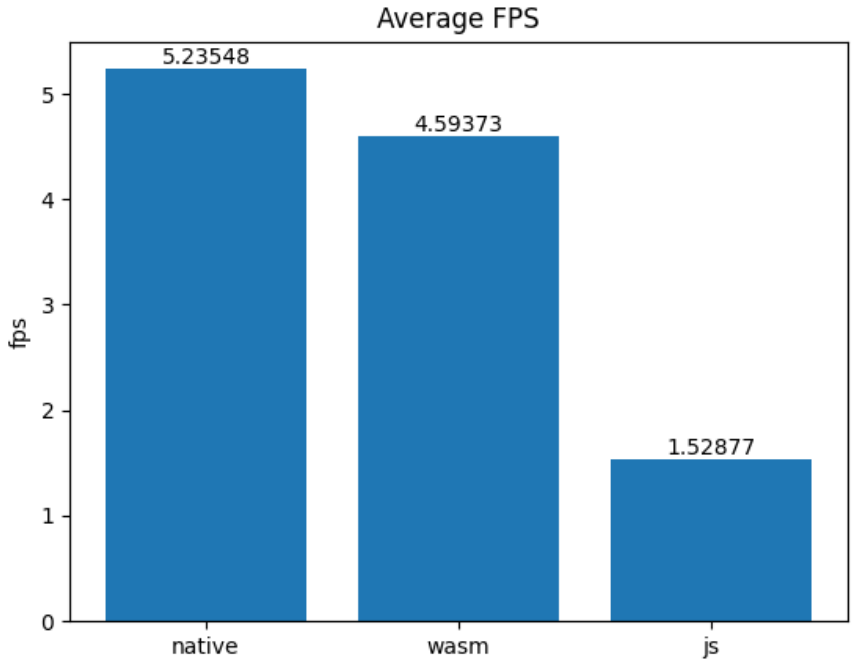

Figure 9: the average FPS of the program with anti-aliasing enabled, running natively, in WebAssembly, as well as JavaScript.

Although it is not of direct relevance to this work, we decided to include the performance of the same program compiled to JavaScript as well. What we can clearly tell is that the native program performs the best of the three, with a frame rate of more than 20 FPS with anti-aliasing disabled, and more than 5 FPS with it enabled. The WebAssembly program ends up with a near 15% slower frame rate. The JavaScript program is by far the slowest of the three, being more than 3 times slower than the native program. We can see that this ratio between the three holds even as we increased the workload of the program by enabling the anti-aliasing. This suggests that there is a linear correlation.

9.2. Performance counters

After the programs finished executing, the Perf tool told us the exact task-clock as well as the exact count of some of the most common events. Included in these counts were the number of context switches, cpu-migrations, page-faults, cycles, instructions, branches and branch-misses. In this context, page-faults refer to both minor page-faults (those that do not require disk I/O) and major page-faults (those that do require disk I/O), as described in the man page [21]. We compiled the results into a table, table 1.

Events	linux native	wasm
task-clock (ms)	18 041,5385	20 881,81
context-switches	389	860
cpu-migrations	9	3
page-faults	11 649	242 263
cycles	57 388 350 723	70 158 411 619
instructions	129 302 984 924	169 009 682 056
branches	13 362 357 502	16 133 334 206
branch-misses	43 370 251	154 566 410

Table 1: the exact count of the measured low-level events in WebAssembly as well as natively.

To make it easier to compare the results of the native and the WebAssembly benchmarks, we compiled these performance counts to a normalized graph, as seen in figure 10.

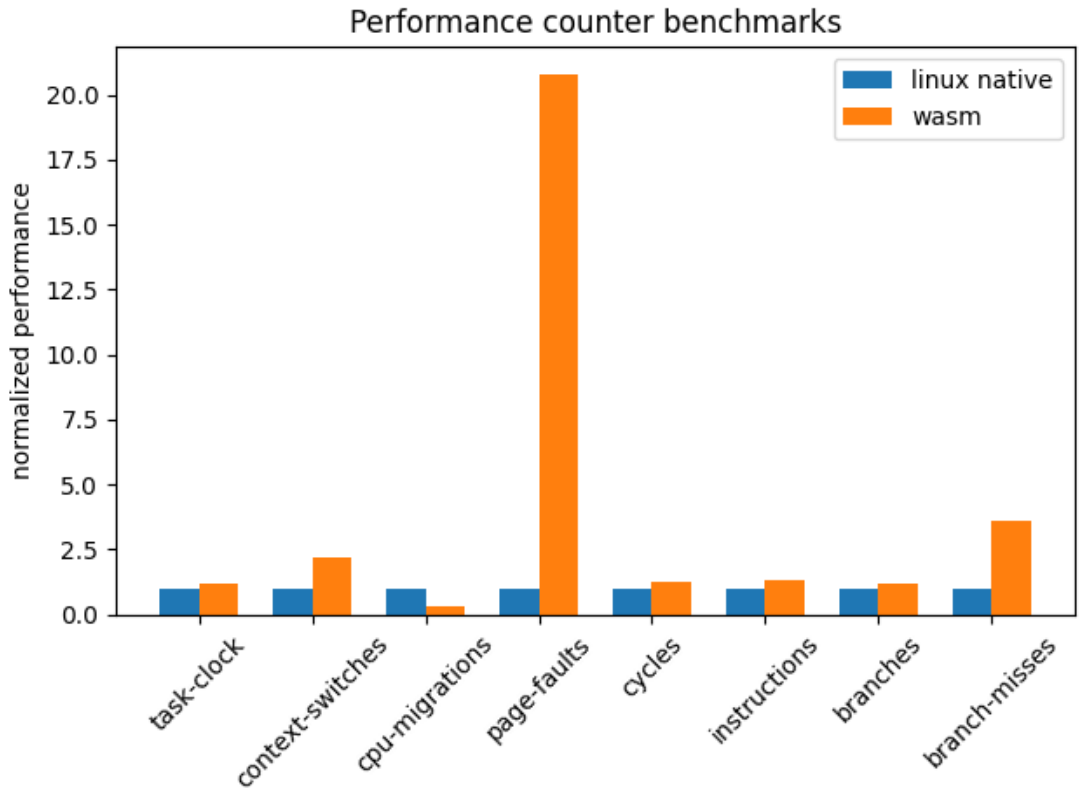

Figure 10: the normalized count (native = 1) of the common low-level events executed natively and in the WebAssembly program.

Unsurprisingly, we can see that the WebAssembly version of the program generates considerably more cycles, instructions, as well as branches. We can also see that the WebAssembly version produces significantly more branch-misses. What's really striking, how ever, is the huge spike in page-faults.

10. Discussion

Our results concluded that the WebAssembly program executed almost 15% slower than the equivalent native program. If the goal is for WebAssembly to compete with native programs, this difference can be seen as quite significant. The results of the second part of the study, using program counters, explains this difference. We can see that the WebAssembly program generates considerably more instructions and branches as well as far more page-faults and branch-misses. WebAssembly does not Unfortunately, due to time constraints, we will not be able to pin down the exact cause of these events, but one can still make some reasonable speculations.

The increased amount of instructions could be attributed to the same findings of A. Jangda, et al., that WebAssembly produces more stores and loads. I believe another possible cause of the increase in instructions could be due to limited support of SIMD in WebAssembly. Due to portability concerns, WebAssembly SIMD does not expose the full native instruction sets [22]. Ray tracing is highly parallelizable since all rays can be traversed in isolation, making SIMD highly applicable [23]. Limited SIMD capabilities could therefore negatively impact the performance. The increased amount of branches is likely the result of the browser's need for more dynamic safety checks. With more instructions, and more branches, it would make sense that we see more cache-misses and branch-misses, but the sheer scale of the increase is still surprising, and is something that should be investigated further.

In the work of A. Haas et al. several of the benchmarks executed within 10% of the speed of native. Unfortunately, our benchmarks were slightly slower. Because of this, I don't think that WebAssembly is competitive with native on the point of performance. How ever, our benchmarks were considerably faster than those of A. Jangda, et al., where WebAssembly was 55% slower than native in Chrome. What can be concluded when looking at previous works, as well as my own, is that the difference in performance is heavily dependent on the type of application. Because of this reason, I do not think we can make general statements about WebAssembly's performance, but rather have to test and evaluate different types of applications separately.

I do not think WebAssembly will ever be able to compete with native for every type of application. But, I do think WebAssembly is *fast enough* for many types of applications, and it is clearly an improvement over JavaScript and other previous solutions. With sufficient performance thanks to WebAssembly, and due to all the other benefits developing for the web brings, I think we will see an increase in the number of web applications.

11. Conclusions

In this work, we investigated how WebAssembly performed in one application in a previously unexplored context – that of graphical applications. More specifically we wanted to see how well a given ray tracer performs in WebAssembly, and whether it could compete with the equivalent native application. We implemented a ray tracer and compiled it to a native executable, as well as WebAssembly. Rather than just seeing how WebAssembly differed in performance, we wanted to open the possibility to answer what causes said difference. For this reason, we carried out the necessary work to attach performance counters to the web application.

Our results showed that the WebAssembly application reached a frame rate almost 15% slower than that of the native application. Which is a significant slowdown. Previous measurements of other applications show smaller differences than ours, while others show greater differences. Clearly the performance of WebAssembly, compared to native, is dependent on what type of application is measured.

The results of the performance counters show that the WebAssembly program generates more instructions, more branches, more cache misses and more branch misses, than the native application. The increase in these events is a clear signifier of the decrease in performance. We speculate that the increase in instructions is the result of WebAssembly generating more load and store operations, as well as limited support for SIMD instructions in WebAssembly, and that the increase in branches is the result of WebAssembly having a higher need of dynamic safety control.

12. Future Work

While we think we accomplished most of what we set out for with this work, there are still a few things left to be desired. Our work laid the groundwork to be able to perform a more thorough low-level analysis of the resulting programs. We concluded that the decreased performance in the WebAssembly program is due to more instructions as well as more branches being generated. How ever, we could only speculate on where these extra instructions and branches came from. If we could assess what caused these extra events, perhaps something could be done to close the gap between WebAssembly and native performance further. We would also like to see experiments done on a larger range of graphical applications. We believe the work we have done here can serve as a framework for more work in the area.

References

[1] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, 2017, doi: 10.1145/3140587.3062363.

[2] "About Emscripten." https://emscripten.org/docs/introducing_emscripten/about_emscripten.html (accessed August 19, 2021, 2021).

[3] M. Pharr, *Physically based rendering from theory to implementation*, 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010.

[4] B. Yee *et al.*, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *2009 30th IEEE Symposium on Security and Privacy*, 17-20 May 2009 2009, pp. 79-93, doi: 10.1109/SP.2009.25.

[5] D. Herman, L. Wagner, and A. Zakai. "asm.js." http://asmjs.org/spec/latest/ (accessed September 20, 2021, 2021).

[6] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," presented at the Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, Portland, Oregon, USA, 2011. [Online]. Available: https://doi.org/10.1145/2048147.2048224.

[7] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 20-24 March 2004 2004, pp. 75-86, doi: 10.1109/CGO.2004.1281665.

[8] D. Mathew, B. A. Jose, J. Mathew, and P. Patra, "Enabling Hardware Performance Counters for Microkernel-Based Virtualization on Embedded Systems," *IEEE Access*, vol. 8, pp. 110550-110564, 2020, doi: 10.1109/ACCESS.2020.3002106.

[9] "Perf Wiki." https://perf.wiki.kernel.org/index.php/Main_Page (accessed December 12, 2021, 2021).

[10] "Tutorial - Perf Wiki." https://perf.wiki.kernel.org/index.php/Tutorial (accessed December 12, 2021, 2021).

[11] A. Jangda, B. Powers, E. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," p. arXiv:1901.09056. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2019arXiv190109056J

[12] S. Kang and J. Lee, "Improving rendering speed of 3D geospatial data based on HTML5/WebGL using improved arithmetic operation speed," *International Journal of Urban Sciences*, vol. 23, no. 3, pp. 303-317, 2019/07/03 2019, doi: 10.1080/12265934.2018.1476175.

[13] "About SDL." https://www.libsdl.org/ (accessed October 27, 2021, 2021).

[14] "Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms." https://eigen.tuxfamily.org/ (accessed October 27, 2021, 2021).

[15] T. Macwright. "literate-raytracer." https://tmcw.github.io/literate-raytracer/ (accessed October 27, 2021, 2021).

[16] L. Johansson. "ray-tracer." https://github.com/ludwigdev/ray-tracer (accessed January 2, 2022, 2022).

[17] "clang - the Clang C, C++, and Objective-C compiler." https://clang.llvm.org/docs/CommandGuide/clang.html (accessed February 13, 2022.

[18] "Clang optimization levels." https://stackoverflow.com/a/15548189 (accessed February 13, 2022.

[19] "About Node.js®." https://nodejs.org/en/about/ (accessed January 12, 2022, 2022).

[20] "Express." https://expressjs.com/ (accessed January 12, 2022, 2022).

[21] "perf_event_open(2) — Linux manual page." https://man7.org/linux/man-pages/man2/perf_event_open.2.html (accessed Feburary 13, 2022, 2022).

[22] "Porting SIMD code targeting WebAssembly." https://emscripten.org/docs/porting/simd.html (accessed February 13, 2022.

[23] G. Marmitt, H. Friedrich, and P. Slusallek, "Efficient CPU-based Volume Ray Tracing Techniques," *Computer graphics forum*, vol. 27, no. 6, pp. 1687-1709, 2008, doi: 10.1111/j.1467-8659.2008.01179.x.