Utrecht University

Master Thesis

Supervisors: dr. ing. Jacco Bikker, dr. ir. A. Frank van der Stappen

---

Porting a CUDA Path Tracer to WebGL

(a case study in porting large graphics applications to the web)

Andrius Jaškauskas

ICA-6070981

Faculty of Science

Department of Information and Computing Sciences

21 February 2020

Contents

1. Introduction

Path tracing is one of the most popular ways to render high-quality 2D images for a given 3D scene from an arbitrary camera position. Current path tracers tend to be developed as local applications. It is critical to make path tracers easily accessible to a broader audience and to bring them to the next level. A straightforward way to do that is to create a path tracer which would be accessible via the web browser from external computers without installing any additional software.

The overall goal of this project is to develop a WebGL path tracer running on a web server. The main technical issue is to port already implemented NVIDIA CUDA path tracer (later called the CUDA path tracer) named LightHouse to WebGL. Besides, investigations into different technologies that are capable of running native GPU code on the web will be conducted; this will ensure that WebGL is the optimal choice for this given case study.

Despite it being a mostly technical issue, scientific questions also emerge from this case study. One scientific problem arises from the main critical issue of computing time needed to render an image. Path tracers that are implemented on technologies such as NVIDIA CUDA and run on local machines with powerful hardware can already achieve acceptable performance. Implementing computing time-sensitive applications for the web can be a demanding task. Practical experiments will be performed to test if a web path tracer can achieve comparable performance to the CUDA path tracer.

The 3D scene typically contains textured geometry and a sky-dome, both of which are encoded in specific file formats. These files may contain a large amount of data. In order to render the image, the path tracer program has to retrieve the scene data. Data retrieval while path tracing on the local machines typically does not cause critical issues since the data files are stored together with the application. For web path tracing, this is not a trivial task; in order to start a path tracing process, a user has to download data files from a server first. Downloading large data files may result in long waiting times. In this project, we will perform literature research on how to manage caching of complex data for graphics since that can affect our application development process.

By the end of this project, it is expected to have a WebGL path tracer (which is replicating the CUDA path tracer functionality) running on a web server. Additionally, we will conduct practical experiments to evaluate newly developed application performance. Finally, all scientific research and case study results will be presented.

1.1. Research Questions

The following research questions are expected to be answered throughout the master thesis project:

1. Can an industry-strength path tracer run on a website? Is WebGL suitable for such a task?
2. How does the performance of a real-world medium-scale rendering application implemented in WebGL compare to native performance?

2. Preliminaries

This chapter provides background knowledge for the project. Basic concepts of the path tracing algorithm will be described, case study application introduced, and some background about web-based applications given.

2.1. Path Tracing

2.1.1. Physically Based Rendering

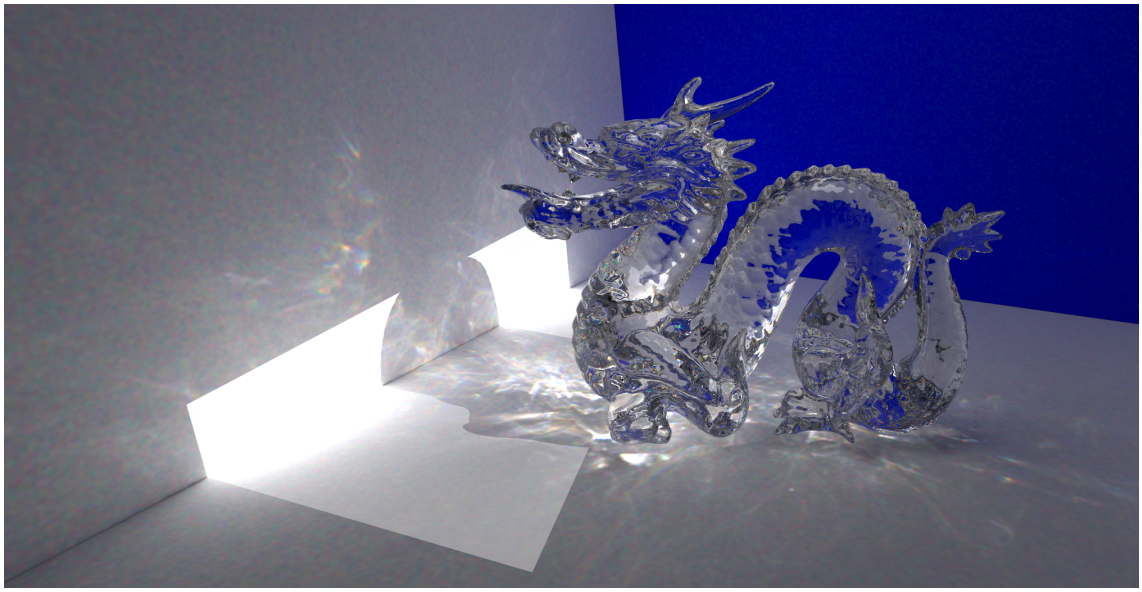

Figure 1: Image rendered with path tracer, showing complex light interaction with glass and diffuse materials, yielding accurate shadows and caustics.

Path tracing is a method to simulate real-world light transport in computer graphics by evaluating a large number of photon paths. The first researcher who came up with the idea of how to generalise a variety of known rendering algorithms was James T. Kajiya in 1986. The author has introduced the rendering equation (Equation 1), which mathematically describes real-world light transport [Kaj86]. It takes into account direct and indirect illumination, which is light that gets reflected more than once in the scene before reaching the camera. The proposed technique has also extended the range of optical phenomena which can be effectively simulated (an example is given in Figure 1).

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos\theta_i \, d\omega_i \tag{1}$$

The given equation evaluates the light a point emits, plus the light the point reflects (incoming over the hemisphere). Here ω_i and ω_o stands for incoming and outgoing direction respectively, and p is a point:

- $L_o(p, \omega_o)$ - outgoing radiance from a point p on a surface towards direction ω_o.

- $L_e(p, \omega_o)$ - emitted radiance from a point p towards direction ω_o (direct illumination).
- $f_r(p, \omega_i, \omega_o)$ - Bidirectional Reflectance Distribution Function (BRDF). It describes what proportion of the light coming from ω_i is reflected towards ω_o.
- $L_i(p, \omega_i) \cos \theta_i$ - amount of irradiance from the direction ω_i at a point p (indirect illumination), $\cos \theta_i$ converts incoming radiance to irradiance.

Monte Carlo integration is a method for solving an integral by applying stochastic sampling. This method is typically used for those integrals which are not feasible to solve analytically. Equation 2 generally describes the Monte Carlo method. It relies on the fact that the integral of a function equals the expected value, which is calculated as an average of N random samples calculated for the function.

$$\int_{\Omega} f(x) = E(f) \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \tag{2}$$

The rendering equation contains an integral that does not have a *closed* form (it can not be calculated by a finite number of operations). The best we can do is approximate it using random sampling. Kajiya has proposed to apply Monte Carlo integration for rendering equation approximation [Kaj86]. Rendering equation expression by applying Monte Carlo integration is provided in Equation 3. Replacing the infinite integral by the average of multiple random samples transforms the equation into a *closed* form. This approach allows us to implement the path tracing algorithm practically.

$$L_o(p, \omega_o) \approx L_e(p, \omega_o) + \frac{1}{N} \sum_{i=1}^{N} f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos \theta_i \tag{3}$$

The light an arbitrary point reflects can be calculated by a recursive integral. The path tracing algorithm randomly continues the light path and evaluates the emitted and reflected light for each point hit by a random light path. $L_i(p, \omega_i)$ term causes a recursion in the given integral (Equation 3) to calculate the radiance for each arbitrary point by integrating over its hemisphere. Note that the recursive integral yields a random light path rather than a single sample over the hemisphere (Figure 2).

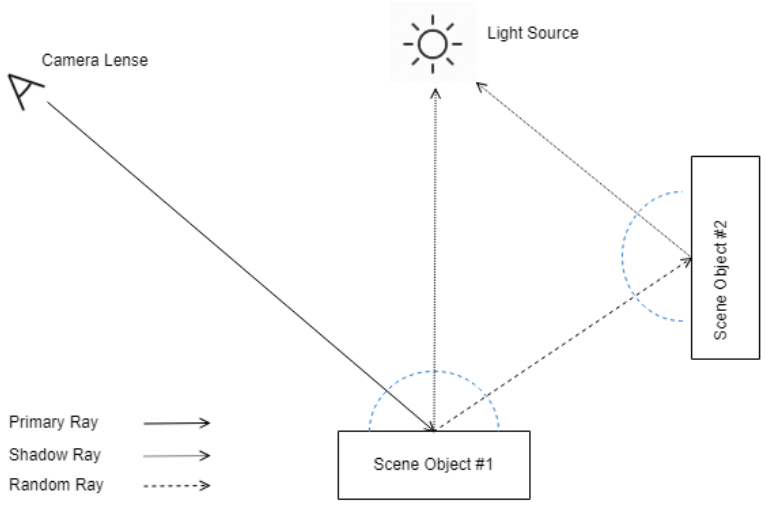

Figure 2: General scheme of path tracing with Next Event Estimation.

2.1.2. Image Rendering

A 2D image is rendered by evaluating Equation 1 for each pixel on the screen. We calculate a radiance value of all visible points from the camera. The path tracing process is computationally expensive because it traces a large number of rays against the scene geometry to render a single image. We need to sample incoming radiance over the hemisphere of a given point. To accomplish this, there is a *primary* ray shot from a camera through each pixel on a virtual image plane towards the scene. Whenever it hits the scene geometry, we cast two random rays: the first to a random direction over the hemisphere (to estimate indirect illumination) and the second to a random light source (to estimate the direct light portion of the energy).

Calculating more samples for the same pixel reduces the noise (variance) of the final image. Although calculating multiple samples is computationally expensive, path tracing researchers have proposed multiple variance reduction techniques that have become state-of-the-art in this area. Commonly applied techniques are Russian Roulette (RR), which introduces ray survival probability to avoid infinite ray bounces; Importance Sampling (IS), which samples only those areas of the hemisphere which are likely to contribute more light; and Next Event Estimation (NEE), which splits sampling process into direct and indirect light sampling. Description of mentioned and other variance reduction techniques can be found in Humphreys et al. book "Physically based rendering: From theory to implementation" with more detailed explanation [MPH16].

2.2. LightHouse

This project is a case study in which a CUDA path tracer named LightHouse will be ported to WebGL. The CUDA path tracer, provided by Utrecht University, is a native application which has to be installed on a client machine. LightHouse was built as a commercial application for rendering realistic images of given products such as baby strollers, furniture, and clothing. The main application architecture contains different rendering cores which

are all controlled via rendering API. It is implemented in an object-oriented manner; all components are encapsulated in different classes.

Here is an abstract list of the key functional features implemented in the given path tracer:

- Render a photorealistic high-quality 2D image from a textured 3D scene (Figure 3).
  - It can produce a noisy image of the scene in a real-time.
  - The noise can be significantly reduced, and a high-quality image can be rendered in less than one second.
- Load complex scene from popular 3D geometry files (*.gltf*, *.obj*, *.fbx*).
- Construct or rebuild acceleration structure (BVH) for the scene at run-time.

At a low level, LightHouse deploys the wavefront algorithm [LKA13], which splits a single monolithic path tracer into several smaller kernels. For our research, it is not clear whether this is a suitable approach.

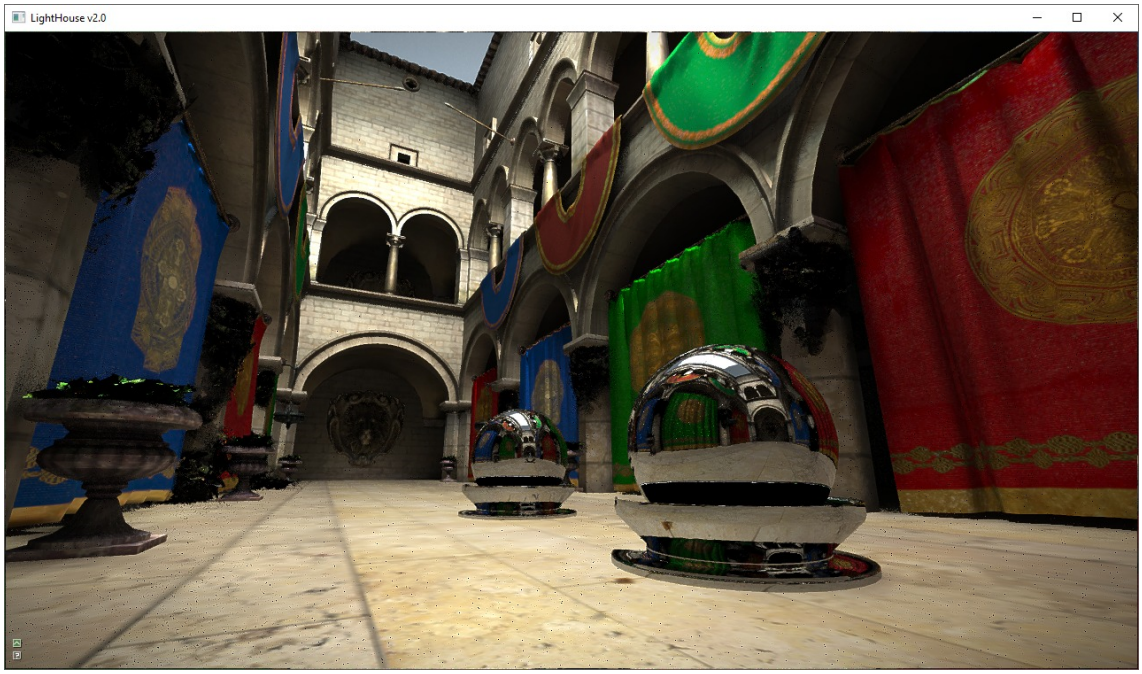

Figure 3: Example scene rendered with LightHouse path tracer.

2.3. Web-Based Applications

A web application is a type of software that we can refer to as client-server software. The application itself is deployed on a server, and a user accesses it via client application (typically a web browser). Typical control-flow of a web application involves:

1. The user submitting a request to the server via a web client.

2. The server processing it and sending a response back.
3. A web client then processes the received response and provides the final output for the end-user (rendered visual content is standard output, obtained by interpreting HTML code by a web browser).

In this project, the main focus is on HTML5 with JavaScript web applications because WebGL is a native JavaScript API. WebGL requires a *canvas* element (which was introduced in HTML5) to draw on it. It is important to note that JavaScript code is executed on a client machine. This execution method allows us to exploit the computational power of a client machine without too much load for a web server. The JavaScript web applications, however, still have to be accessed via a web client.

The main difference between native and web path tracers is how they retrieve scene data into the application. A native renderer is typically deployed on a computer together with all required data. When running this application, data is read directly from local storage. Implementing a web path tracer requires the user to retrieve data from a server to be able to render an image. Scene data for path tracing usually contains multiple different files: 3D models, sky-dome, textures, and settings. These files typically store a large amount of data (from tens of megabytes to gigabytes). Their content also may change over time because the scene can be edited. This causes a complex problem that involves data caching into the web path tracing process; downloading a large amount of data every time a web path tracer is accessed is costly for the end-user.

There are two different options on how to store data for a web application: temporary and persistent. Persistent data is supposed never to be deleted unless the user explicitly takes action. Temporary data is typically maintained by applying LRU (the Least Recently Used) eviction policy [Net]. Currently, there are no globally accepted limits for the local storage available on the client for running the web application. These limits may differ between different web clients. JavaScript has native APIs to check available quota for storage. For example, Mozilla Developer Network provides such limits for web applications - the global limit is 50% of free disk space, and the limit per domain is defined as 20% of the global limit (with a minimum of 10 MB and a maximum of 2 GB) [Net].

The guidelines for web storage limits seem to be sufficient for this project, and it is expected that scene data storage on a client machine can be implemented successfully. However, even though we have enough space to store the scene data, caching it efficiently requires further research.

3. Related Literature Overview

Porting a CUDA path tracer to WebGL is the main technical task, which requires theoretical knowledge about several different areas presented below.

- **Acceleration Data Structures for Path Tracing**. We will provide a detailed overview of acceleration data structures, particularly about a BVH. Considering that this study is not aimed to improve the path tracing algorithm itself, we limit the literature study to acceleration data structures; acceleration data structure will be implemented separately instead of being ported from LightHouse. Section 3.1 explains why the acceleration data structure will be implemented from scratch, instead of being ported.

- **Native GPU Code for Web**. The comparison of different technologies to run native GPU code for the web is necessary to ensure that WebGL, together with GLSL, is an appropriate choice for the task at hand.

- **Data Caching in Web Applications**. Research about caching for the web has to be conducted in order to perceive current limitations.

3.1. Acceleration Data Structures for Path Tracing

Scene geometry typically consists of many primitives. A primitive is the most basic geometric shape: triangle, sphere, cube, cylinder or even torus. For each generated ray, a ray-primitive intersection has to be performed to find out if a given ray hits the scene geometry or not. Considering the vast number of intersection tests involved in producing a single image, the intersection check is one of the costliest parts of the path tracing algorithm. Acceleration data structures in path tracing have a significant role and are used for ray-primitive intersection performance improvement. There are many different structures, such as Bounding Volumes Hierarchies (BVH), grids, kd-trees, and oct-trees.

Since acceleration structure maintenance and traversal in LightHouse is handled by Optix API [PBD+10], it cannot be ported to the WebGL path tracer. This means that this part has to be developed independently from LightHouse; therefore, the WebGL does not have a native tool to handle acceleration structure for path tracing. We will thus research how to replace the Optix functionality. For this, we will use the BVH. We consider the research of other acceleration structures to be outside the scope of this project. The BVH is a bounding volume hierarchy which means that all primitives in the scene are surrounded by bounding volumes, and they are grouped and connected into one structure (usually a binary tree) where each tree node encloses its descendants (see Figure 4) [RW80].

In the following subsections, we provide an overview of state-of-the-art methods for BVH construction and traversal.

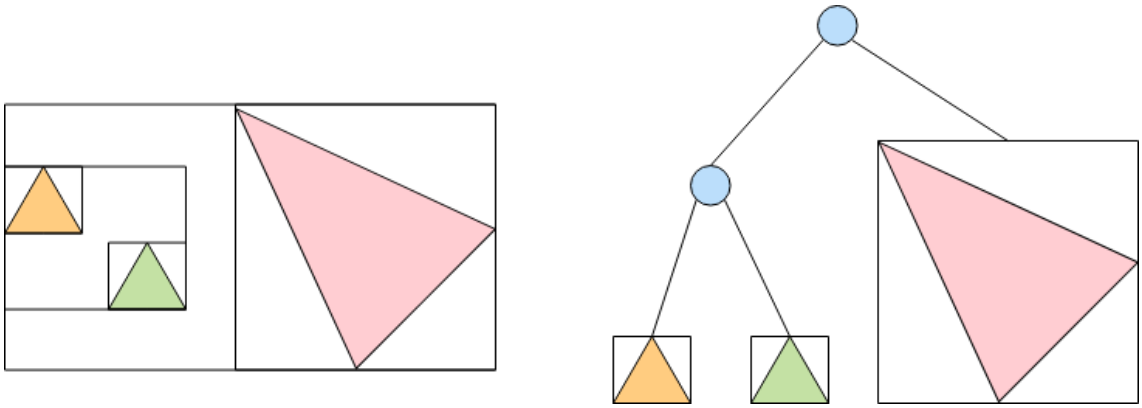

Figure 4: General BVH scheme. Scene geometry grouped into bounding volumes - on the left, binary BVH tree - on the right.

3.1.1. Construction

Each primitive in the scene has a bounding volume. The BVH is typically constructed using a top-down approach. The input consists of all primitives surrounded by one global volume. Initially, it is subdivided into two volumes, and each sub-volume is further recursively partitioned until a termination criterion is met. A BVH leaf node typically contains 2-3 primitives. Wald et al. [WBS07] provide an algorithm for BVH building which uses Surface Area Heuristic (SAH) method introduced by Macdonald and Booth [MB90]. Typically SAH method is used to increase the probability of a ray hitting a primitive. This technique increases the quality of the BVH tree when considering a split plane of a volume. It is important to find minimum intersection cost - the lower splitting cost is the bigger probability of ray hitting geometry in the particular bounding volume. The splitting cost calculation formula is provided below:

$$C_{split} = C_T + A_{left}N_{left}C_I + A_{right}N_{right}C_I \tag{4}$$

- C_{split} - splitting cost.
- C_T - traversal cost (time), specific number varying depending on implementation.
- A_{left} and A_{right} - total surface area of all primitives in left and right subnode.
- N_{left} and N_{right} - total amount of primitives in left and right subnode.
- C_I - intersection cost of one specific primitive (e.g. different cost to intersect triangle or sphere).

3.1.2. Traversal

BVH traversal refers to a method on how to perform intersection between the BVH and a ray with as few operations as possible. Generally, the BVH is traversed using a stack. After starting from a root node, we test which children nodes are hit by the ray and continue to

traverse those until a leaf is reached. Typically a BVH node has two children, and a ray always visits the *near* child first. An index of the *far* child is pushed to the stack and is traversed afterwards if no intersection is found. A leaf is traversed by simply performing a ray-primitive intersection for each primitive inside it. Assuming that the BVH is typically a binary tree, its traversal complexity is logarithmic - $O(log_n)$.

The BVH structure can be altered and converted to an MBVH (Multi-branching BVH) where one node can hold more than two children. These types of structures are aimed to speed up traversal. They strongly benefit from SIMD (Single Instruction Multiple Data), which allows for a single ray to be tested against multiple nodes in parallel. Wald et al. described BVH traversal with 16-way branching [WBB08]. Their method performance has been stated to be significantly better for less coherent rays (e.g. for soft shadows) but performs worse for *primary* rays. Similarly, Dammertz et al. described *n-way* branching BVH traversal algorithm [DHK08]. Tree structure was defined as QBVH (Quad-BVH), where n is 4. They claimed that their method outperforms current high-performance single ray-tracing algorithms and also reduces memory bandwidth.

A GPU-based ray traversal algorithm has been proposed by Ylitie et al. from NVIDIA research [YKL17]. The proposed algorithm significantly reduces memory traffic and results in roughly two times faster incoherent ray traversal compared to current state-of-the-art techniques. They also propose a technique to compress a binary BVH into MBVH (authors call it wide BVH) in a SAH-optimal fashion.

3.2. Native GPU code for Web

Path tracing is a costly algorithm in terms of computational power. It is, however, a massively parallel algorithm and thus particularly suitable to run on GPU. Purcell et al. performed early research on how a ray-tracing algorithm could be efficiently implemented on GPU [PBMH02]. Currently, there are already accepted and well-performing technologies to run code on the GPU such as CUDA and OpenCL. Despite the good performance, those technologies are suited to run GPU applications only locally, without a straightforward way to make an application accessible via the web. Since the project aim is to bring path tracing to the web, it is necessary to explore existing options to run a native GPU code for the web.

3.2.1. WebAssembly

Recently, WebAssembly was introduced for web development (abbreviated Wasm). Haas et al. describe WebAssembly as the first low-level technology for the web, which meets all design requirements [HRS+17]. These requirements are stated to be: *safe*, *fast*, *portable*, *compact*. The most relevant requirement for this study is portability. The authors state that WebAssembly is hardware and platform-independent, which make applications runnable across different browser and hardware platforms. Even though this technology pretends to be highly innovative and replace JavaScript, it still does not have a preferred way to run code for GPU by implementing massive parallelism and achieve significant speed-ups. Conrad Watt has presented a mechanised specification for WebAssembly called Isabelle [Wat18]. The author states that WebAssembly currently focuses more on formal "pen and paper" specification rather than something practical that is ready to use.

3.2.2. Emscripten

Emscripten is an LLVM (Low-Level Virtual Machine) assembly to JavaScript compiler. Alon Zakai from Mozilla research has presented this tool [Zak11]. Emscripten provides two approaches on how to run the application on the web written in different languages than JavaScript.

- Compile code directly into LLVM assembly, and then compile it into JavaScript. This approach can be applied for running C/C++ code.
- Compile a language's entire runtime into LLVM and then into JavaScript, as in the previous approach, and then use the compiled runtime to run code written in that language. This approach can be applied to run, e.g. Python code.

These approaches might be useful for source code porting even though Emscripten is just a tool to get non-JavaScript code into JavaScript (which still uses WebGL together with GLSL shaders to run it). Emscripten itself does not provide any new technology on how to run a native GPU code on the web.

3.2.3. WebGL with GLSL shaders

WebGL [DJAI15] is a native JavaScript API for rendering 2D and 3D graphics on the web. It is already supported by all the most popular browsers. WebGL produces output to HTML canvas element. Its program consists of two major parts: JavaScript calls to the WebGL API and shaders written with OpenGL Shading Language (GLSL), which syntax is similar to C-family languages. WebGL is developed and maintained by the Khronos Group[1].

While developing with WebGL, it is required to implement shaders in order to get graphics on the screen. Shaders are small programs which compile into assembly code which is executed on GPU. Typically a shader is composed of two parts: the *vertex shader* and the *fragment shader*. The vertex shader is responsible for converting (typically 3D) object coordinates into 2D display space. The fragment shader is responsible for determining the colour for each pixel on display depending on the different input scene settings such as textures, lighting and material properties. [Par12, p. 14].

Currently, there are no other feasible technologies to run native GPU code for web except WebGL. Other technologies either are very limited or still in the development stage, which makes WebGL the only viable choice for path tracer implementation on the web. For this project, we will use WebGL version 2.0 because this is the last stable version with a note that all applications will be portable to the new versions of WebGL.

For this project, we will rely mostly on the shader functionality of WebGL (GLSL programming language). Since the path tracing algorithm final result is the colour for each pixel on the screen, the main functionality will be implemented in the fragment shader. Shaders will be used to fill a texture that is applied to a screen-filling rectangle. LightHouse is implemented by combining C++ and CUDA code. Thus in this project, we have to encapsulate a significant amount of native GPU code in the shaders. These shaders implementation is

[1] https://www.khronos.org/webgl

a challenging technical task which will help to answer the question if WebGL is suitable to implement an industry-strength path tracer.

3.3. Data Caching in Web Applications

Data caching refers to the computing issue where data exists that has to be accessed multiple times; ideally, this data would be fetched only once. When considering web caching, fetching refers to downloading files from a remote web server. Downloading large files results in internet traffic load, which in turn results in increased waiting times. An efficient cache reduces the impact of repeatedly accessing the same data. This significantly improves web application performance by reducing waiting times and internet bandwidth consumption. Jia Wang has addressed caching as an essential technique to improve web-applications performance in early research about caching schemes for the web (mostly focusing on the most beneficial cache proxy server placement in the global system) [Wan99].

In order to have an efficient cache implementation, we must have adequate space to store the data (local storage in our case) and effective eviction policy. Free cache storage is always limited, so we need to find a smart way to replace out-dated and save new/fresh data. Data can be cached at various points in the network (e.g. proxy server, client machine). Regardless of the caching implementation point, the critical concept is storage replacement strategy (namely *eviction policy*) - it describes a strategy on how cached data storage will be maintained (based on various rules to determine which cached objects can be removed to free up space for the new data).

Aggarwal et al. in early research about eviction policies addressed non-homogeneous file size as one of the critical complications when implementing caching for the web [AWY00]. The cache hit ratio can increase when a larger amount of smaller files are favoured over a fewer amount of larger files. Even so, authors also considered that the hit ratio might not be the optimal measure for the web caching because receiving one large file may cost more time than a small one. Also, the authors describe an issue that some web files have an expiration date that can affect caching efficiency.

3.3.1. Eviction Policies for Web

The research about eviction policies by Aggarwal et al. [AWY00] was refined after a few years by Podlipnig et al. [PB03]. The recent research provides the classification of the most common eviction policies based on previous work. The classes they use to represent different policies take most important web factors (such as the size of an object, last request time, time since the last request, number of past requests, access latency, fetching cost) into account to determine candidate objects for replacement:

- *Recency-based strategies.* These strategies use recency as the main factor. Most of them are derivations of LRU (Least Recently Used) policy - it implies that recently accessed objects will be likely reaccessed in the future, so least recently accessed objects should be pruned.

- *Frequency-based strategies.* These strategies use frequency as the main factor (we need to track object access times). Typically, they are derivations of LFU (Least Frequently

Used) policy, which implies that different web objects have different popularity and the most popular objects are likely to be reaccessed in the future and least frequently used objects are replacement candidates.

- *Recency/frequency-based strategies.* These eviction policies are the combination of the first two strategies, that can help to overcome limitations of mentioned techniques (e.g. when using LFU multiple objects can have similar frequency rate, so additionally considering access recency could reduce choice ambiguity). Although they can introduce undesired additional complexity when implementing.

- *Function-based strategies.* These strategies use potentially general function to determine object value which will serve as a key argument to determine replacement candidate (depending on the function, the largest or lowest value indicates that the particular object should be pruned). Typically different factors serve as the weighted parameters. Most of the function-based strategies consider the size of the file because it has a significant role in caching for the web.

- *Randomized strategies.* Non-deterministic techniques are used to determine replacement candidates. Although the implementation is typically straightforward, it is complicated to measure if these strategies are viable.

There is no perfect eviction policy, and each of them is suitable for different circumstances. A typical measure of cache replacement strategy is *hit-rate*. It implies how often we hit an object in the cache without fetching it from the server. Podlipnig et al. have also described *byte-hit-rate* (mostly affected by size of a file factor) and *delay-savings-ratio* (mostly affected by latency and fetching cost factors) metrics for performance evaluation. Following metrics can be more beneficial when considering caching strategies for the web because web-specific factors have a more significant impact.

4. Related Industry Work

Before starting the development of the WebGL path tracer, it is important to grasp current progress in this area. Because of this, we briefly researched current available public WebGL path tracers. Below some interesting web path tracers found online are discussed:

- WebGL Path Tracer made by Evan Wallace in 2010 [Wal]. Even though it has several scenes, this implementation is basic and does not support material texturing. Scene geometry is hard-coded into shaders before compilation. This path tracer was one of the earliest attempts to implement the algorithm with WebGL.

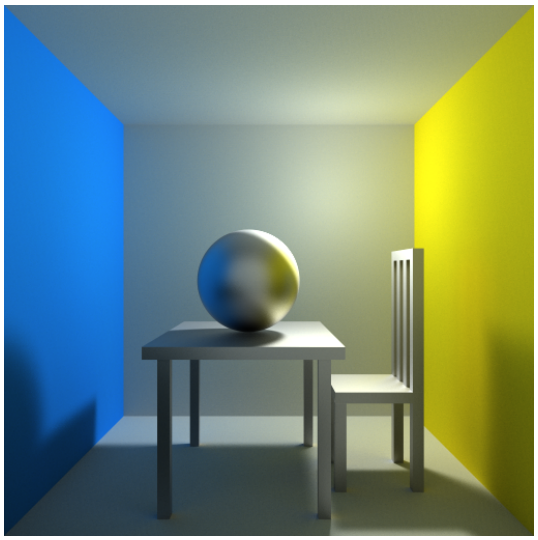

Figure 5: Early WebGL path tracing by by Evan Wallace.

- Caffeine path tracing demo & tutorial created by Rye Terrell [Ter]. WebGL path tracer supports only spheres. The Caffeine molecule scene is hard-coded inside the GLSL fragment shader. The demo implements glossy reflections, but no textures for the materials.

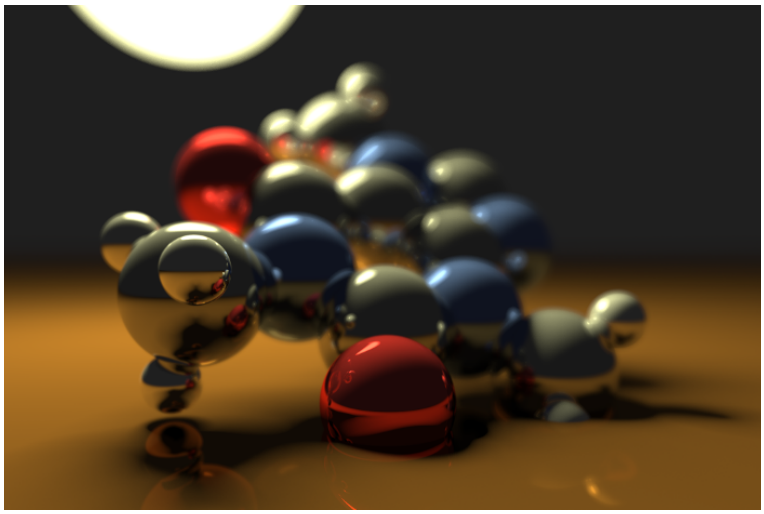

Figure 6: Caffeine molecule rendered by Rye Terrell.

- The WebGL path tracer implemented on THREE.js framework by Erich Loftis [Lof]. This one has the most important features working such as a BVH acceleration data structure, 3D model loading from *.gltf* files, texturing, and even depth of field. It also supports metallic, refractive and diffuse materials together with many different primitives such as spheres, discs, quads, triangles, and cylinders. This implementation seems to be the most functional web path tracer, even though this application is still in progress. Despite that, the path tracer works as a demo application and does not provide functionality for the end-user to set up its scene. User interface currently is minimal. Also, its architecture is fundamental because each demo scene involves the creation of separate GLSL shaders. It does also not support most common 3D file formats such as *.obj* or *.fbx*.

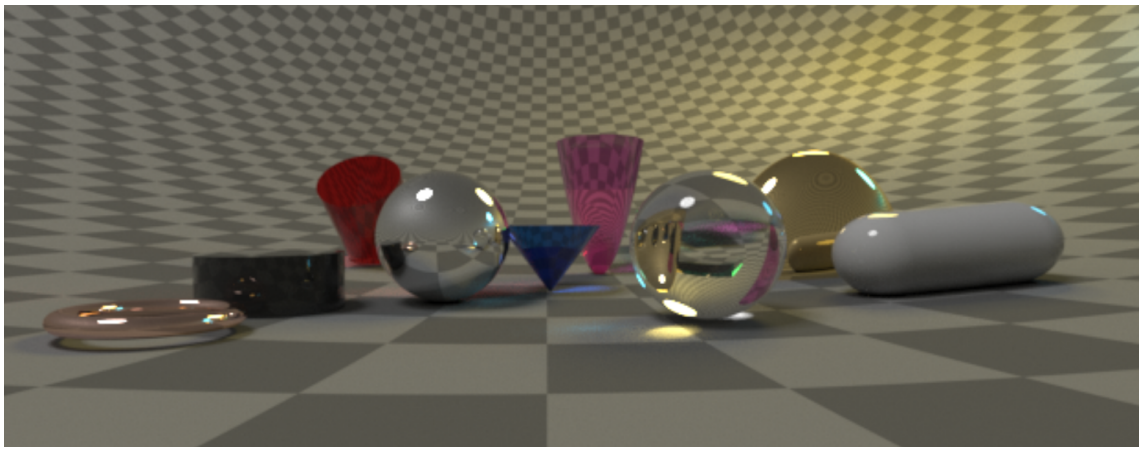

Figure 7: Geometry Showcase Demo by Erich Loftis.

- WebAssembly Ray-Tracer by Simon Niklaus [Nik]. Several demo ray-tracers imple-

mented with JavaScript, asm.js, WebAssembly and GLSL shader. All applications have the same scene with few spheres and textured floor. It can be seen that WebGL ray-tracer performance is the best and its output is the most photorealistic.

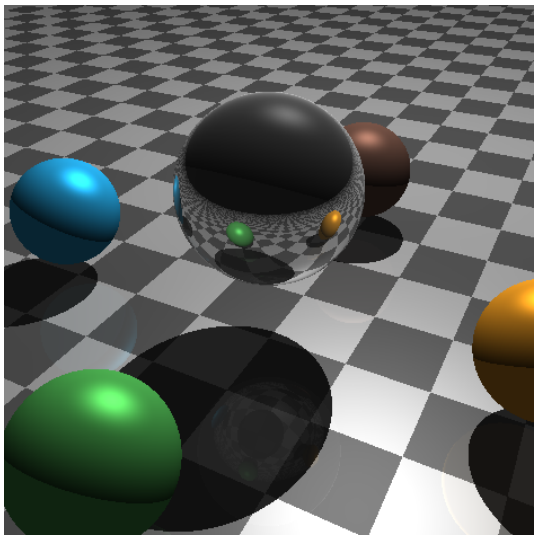

Figure 8: WebAssembly Ray-Tracer by Simon Niklaus

- *LightTracer.org* path tracer [BU19]. Recent real-time industrial path tracer implementation appeared on the web. It has support for base path tracing features, can render photorealistic images real-time. Although the first frames are not stable and deviate a lot in between, image stabilizes only after quite some rendering.

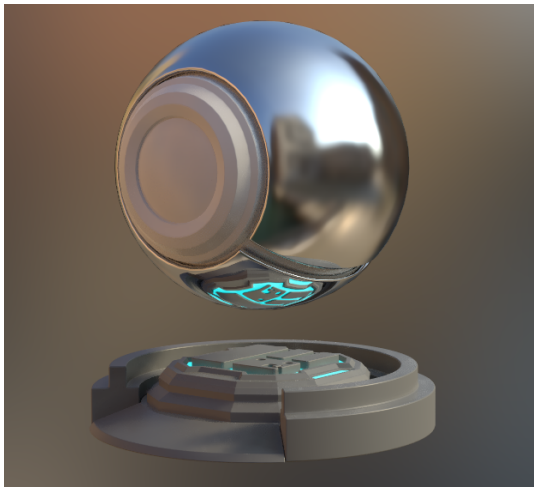

Figure 9: *LightTracer.org* by Denis Bogolepov and Danila Ulyanov

Currently, there is no scientific research being conducted about WebGL path tracing, and it is tough to generalise the current progress because there is also a limited amount of

sophisticated path tracers (that can load geometry from any 3D file formats or construct acceleration data structures). On the other hand, it is important to note that features such as shading, glossy reflections, and mirrors appear to be rendered in a photorealistic manner. Besides, any critical performance issues were not noticed while rendering small photorealistic scenes.

A Web path tracer feature comparison is provided in Table 1. Work by Erich Loftis seems to be very promising while other path-tracers are more basic applications. Loftis path tracer currently supports most of the features and is still in progress. Although his work is mostly done to demonstrate separate features and each scene is implemented as a standalone web page (only using some common dependencies), the application itself can not be seen as a complete industry-strength path tracer. It is not clear if it is possible to combine all features into one application that every scene would be rendered by applying all the mentioned features.

Another interesting work to mention is *LightTracer.org* developed by Denis Bogolepov and Danila Ulyanov. They have created the first industrial web path tracer. Their application can render each frame at a real-time, although first frames have too many deviations amongst each other which makes image change drastically at the beginning. This leads to an uncomfortable experience for the user. It remains unclear why this flaw exists. On the other hand, this web path tracer seems to be promising because its performance remains decent while having essential features in place.

In the current industry work research, it remains unanswered if it is possible to port complex industry-strength CUDA path tracer to WebGL while having all technical aspects covered: achieve comparable performance; maintain complex architecture; have smooth image rendering (not much deviation between frames); being able to render a photorealistic image.

Path Tracer	**Multiple Primitives**	**Camera Controls**	**Reflection**	**Refraction**	**Mesh Loading**	**BVH**	**Depth of Field**	**Multiple Lights**	**Texturing**	**Skydome**
by Evan Wallace [Wal]	+	+	+	-	-	-	-	-	-	-
by Rye Terrell [Ter]	-	+	+	-	-	-	+	-	-	-
by Erich Loftis [Lof]	+	+	+	+	+	+	+	+	+	+
by Simon Niklaus [Nik]	-	-	+	-	-	-	-	-	+	-
by Denis Bogolepov and Danila Ulyanov [BU19]	+	+	+	+	+	+	+	+	+	+

Table 1: Web path tracers feature comparison.

5. Research Methodology

Throughout this project, we research whether WebGL is a suitable technology for complex graphics applications development. In order to examine that, we will implement an industry-strength path tracer on the web using WebGL technology. This particular case study also contributes to the broader question in our research on how mature is this technology for such a task and what performance penalty can be expected for the final application. Finally, we will evaluate our WebGL path tracer under different circumstances in order to perceive how its performance compares amongst different platforms and with the native application.

5.1. Planned Experiments and Results

During this master thesis project, in order to answer research questions from Section 1.1, we will conduct several experiments. To begin with, we will perform a case study by porting LightHouse to WebGL to see if complex graphics applications can run on the web. After porting is completed, we plan to evaluate the performance of WebGL path tracer on multiple different platforms (PC, mobile) to examine WebGL feasibility for such a task. The last step of our experiments is to evaluate LightHouse's performance and compare it with our newly developed web path tracer.

Each before-mentioned experiment will yield particular results that will contribute significantly to our research. The first expected result is the industry-strength WebGL path tracer which runs on a website. This will answer the first part of our first research question. The second expected result is WebGL path tracer performance analysis in two different aspects. By analyzing performance on different platforms, we expect to answer the second part of the first research question; and then, a performance comparison between LightHouse and web path tracer will help to answer our last research question.

5.2. Porting and Implementation

In order to answer emerged questions, we develop a WebGL path tracer. The development process consists of two major tasks. The first one is minimalistic WebGL framework implementation which is responsible for geometry retrieval from graphics files, rendering on canvas, scene creation, acceleration structure building, and other path tracing sub-tasks. The second step is to port code responsible for path tracing from LightHouse to our WebGL application. We port CUDA code to GLSL. In the following section, we present the chosen technologies for WebGL path tracer development.

5.2.1. Chosen Technologies

WebGL path tracer contains two layers: native (JavaScript side) and shader (GLSL side). Although we develop the JavaScript application, source code is written with TypeScript and then translated to JavaScript. Developing with TypeScript provides more flexibility in organizing entities and dependencies between them. Furthermore, TypeScript allows us to implement the system in a fully object-oriented manner and simplifies the development process. Also, that helps to create more complex but scalable architecture which is relevant for our research.

According to our literature research, WebGL is the most suitable technology to implement graphics applications for the web. We use WebGL 2.0 (last stable version). For shaders implementation, we use GLSL ES 3.00 because that is the latest version of graphics language, which is supported by WebGL 2.0. Some of the essential WebGL 2.0 features for us include floating-point and non-power of 2 textures, bit-wise operators (bit shifts), direct texel lookup by pixel id. In WebGL 2.0 loop restrictions in shaders were removed (in WebGL 1.0 a loop had to use a constant integer expression).

The final application version will be deployed on a web server provided by Utrecht University. Web server operating system is irrelevant for our research because the most crucial part of the application will be running locally on a user machine. It is expected that our application will be accessible on all web browsers (including mobile) which have support for WebGL 2.0.

There are multiple general tasks involved in web application development process such as translating TypeScript code to JavaScript or reading binary files. For such tasks, we will use existing external libraries and tools. To handle external dependencies in our framework conveniently, we will use the Node Package Manager (*npm*).

5.2.2. Limitation

It is essential to mention that the technical aim of the case study is to port CUDA path tracer to WebGL. However, we will port only the path tracing code which is responsible for the pixel shading. This limitation is necessary since other parts of the application (such as geometry retrieval, BVH building and traversal, rendered image representation) have to be developed from scratch and are fundamentally distinct due to dissimilarities between technologies (C++, CUDA, Optix versus HTML, JavaScript, WebGL, GLSL). On the other hand, focusing only on porting the most crucial part which affects the final rendered image, we expect to deliver a visually similar output. This limitation also helps to focus more on the WebGL path tracer performance rather than superfluous software engineering issues.

6. Implementation

In this chapter, the implementation of WebGL path tracer is presented. The primary goal is to implement an industry-strength path tracer which can run on a website. To be able to prove it, findings and decisions throughout the application development process will be described in a low-detail level.

6.1. System Architecture

Typical WebGL application architecture contains two major parts: JavaScript application and GLSL shaders. JS part is responsible for all typical web application tasks (such as handling user input, processing data) and the shader performs computations on GPU (outputs a texture). We also follow this architecture design. In our case, the JS part is mainly responsible for scene data loading and preprocessing for the shader input; it also handles user input (camera controls, scene selection). The application has to deal with external dependencies, and it does via *npm*. The actual path tracing algorithm is implemented inside the GLSL shader. Since WebGL draws exceptionally on HTML *canvas* element, there is minimalistic HTML web page created. It serves as an entry point for our application and also provides the end-user interface. General application architecture is presented in Figure 10.

The most pivotal entity involved in path tracing process is a scene containing 3D geometry. In every step over image rendering process, we deal with particular scene objects. In our application, the scene is compound from triangles and their (sometimes textured) materials, light sources, skydome. It also has a virtual pinhole camera which can be controlled via the user interface. The scene has a different representation of its objects in JS application (objects) and GLSL shader (structures). How to retrieve scene data from external file(s) to JS application and later pass to GLSL shader we will discuss in Section 6.2 since that is one of the most crucial parts in our system.

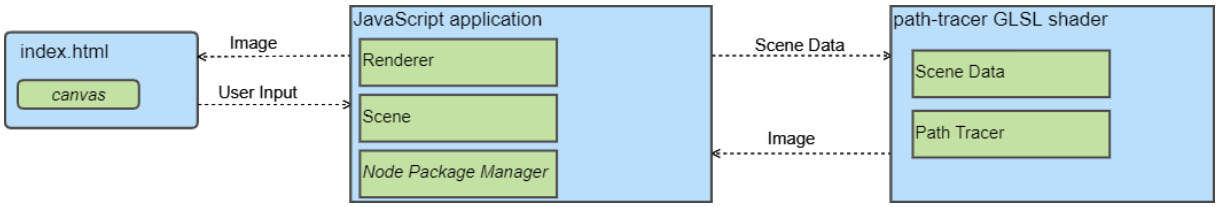

Figure 10: WebGL path tracer architecture overview.

The following sections will give a more certain overview of the JavaScript application and GLSL shader responsibilities.

6.1.1. JavaScript application side

The core entity of JS application is *Renderer*, which serves as the main API to control our web path tracer application; it is also responsible for drawing the final rendered image on *canvas*. The *Path Tracer* is an internal component of *Renderer* which is used to perform

actual image rendering and to dispatch the necessary GLSL shaders where the path-tracing algorithm is performed. In order to run the *path-tracer* shader, it is necessary to ensure that the scene data is loaded, preprocessed and set to GL textures; all these tasks are handled in the JS application side.

6.1.2. GLSL shader side

In order to render scene image, the *Renderer* component operates on two GLSL shaders: *renderer* and *path-tracer*, both of them contain vertex and fragment shader parts. Each shader is implemented in a separate file and is virtually merged into the JavaScript files via *webpack-glsl-loader* during the application building process. By using GLSL loader, we can organize our shader code to be scalable and reusable. We encapsulate some general shader code in separate files (e.g. all scene structures such as ray, triangle, material, light, bounding-box; data-fetch functions; utilities such as the random number generator). Currently, there is no state-of-the-art on how to organize GLSL shader code, so we propose the internal convention for GLSL shader files naming: vertex shaders have **.vertex.glsl* and **.fragment.glsl* file extensions accordingly; files containing shared GLSL code have **.glsl* extension.

The *renderer* shader is very simplistic; it has the only purpose - to take a texture as input and draw it on the screen (*canvas*). The *path-tracer* fragment shader is responsible for the pivotal algorithm. This shader performs path tracing and serves an input for the *renderer* shader. The path tracing algorithm implementation inside the shader contains the code ported from the LightHouse (Lambert BSDF, NEE, MIS) and newly created code (scene structures, data fetching, BVH traversal, intersections, texturing).

6.1.3. Scalability

In this project, we also research the implementation of complex graphics applications for the web in general. In order to contribute to this research, it was decided to design our application in the way to highly de-couple responsibilities between different entities. This makes our system scalable and source code reusable. An important observation is that research in other related areas can advance. Consequently, we could easily adapt our application by replacing out-dated functionality with new state-of-the-art without affecting other core parts. Several interesting design decisions presented bellow.

- We have implemented minimalistic WebGL framework which is responsible for tasks like data retrieving and passing to the shaders, shader uniforms setting, GL buffer drawing. This part of the system can be beneficial for other WebGL researchers since most graphics applications typically implement before-mentioned functionality.
- Architecture decisions such as implementing object factory design pattern for scene creation can be extended, e.g. to develop advanced scene designer tool. This is valid because *Scene* object is just an input for our *Renderer*.
- Different types of geometry (triangle, material, light, texture, BVH, bounding-box) implementation has highly de-coupled responsibilities from other entities. This approach makes them easily adaptable to the more advanced algorithms in the future

without affecting other parts of the system. This makes scene extension trivial since new objects (e.g. different types of primitives or lights) could be added without dealing with already implemented geometry.

6.2. Geometry Loading

In order to render a scene view from an arbitrary camera position, we need a representation of the scene. In this chapter, we will present the whole process of how scene data is retrieved into JavaScript application and then passed to the GLSL shader.

Firstly, the *Scene* serves as an input for the *Renderer* component. A *Scene Factory* is implemented in JS application, and it is responsible for composing all parts of the scene into one object. Secondly, we transform the scene object into arrays of vectors which are the input for *path-tracer* fragment shader. Data from JS application to GLSL shader is transferred via GL textures and uniforms. The number of available GL textures and uniforms can deviate on different hardware (numbers for reference can be 16 GL texture and 1024 uniform units available on GeForce GTX 1050). Inside the shader, scene data is fetched from GL textures upon request, only when needed to perform particular path tracing step (e.g. in order to perform ray-primitive intersection test, we fetch particular triangle from GL texture).

In our path tracer, scene geometry is received from external files. The essential components of the scene are 3D volume geometry and skydome. Other properties such as camera, light sources are created when composing scene inside the *Scene Factory*. In the current application state, *gltf* files are supported for volume geometry and its materials loading; *hdr* files are used to get skydome data. To retrieve text & binary data from external files to JS application we use external libraries (*gltf-loader-ts*, *parse-hdr*). For image color data loading, internal methods are implemented inside the *Geometry Loader*. Consequently, this loader is exploited by *Scene Factory*.

During the scene creation process when triangles are set, we need to build a BVH structure over the provided geometry. The acceleration structure building and traversal will be presented in Section 6.3.

6.2.1. Data Transferring

Once the scene object is compounded inside the JS application, we can start the data transfer process to the GLSL shader. Typically, the scene contains an excessive amount of data. When working with WebGL, we do not have a trivial way to pass the complex scene data to the GLSL shaders. As a solution for this project, we propose to pack the data into GL textures so that we can fetch it back inside the shader. Although, that arises the problem on how to pack that data into a limited amount of GL textures. It is important to point out that the size of GL texture is also limited by the hardware (3379 x 3379 x 4 is the maximum size of one GL texture provided by WebGL function call on GeForce GTX 1050). We also use uniforms to pass multiple parameters (such as the real size of each GL texture, camera position, resolution, other arbitrary settings) to the shaders.

The scene data which we pack into GL textures include triangles, BVH, materials (and their albedo textures), lights, skydome. In our system, we limit that we use a maximum of 16 GL textures (one is used for input already accumulated image). In order to pack scene

data, we dedicate each GL texture for storage of particular scene components. The purpose of each GL texture we use in our system is provided in Figure 11. The size of each GL texture is determined dynamically, depending on the hardware. Each GL texture is limited to be squared, and one dimension size must be the power of 2. For each type of data, there are estimated desired GL texture size, although if the desired size exceeds maximum available GL texture size on the device, we choose the maximum possible size which is still the power of 2.

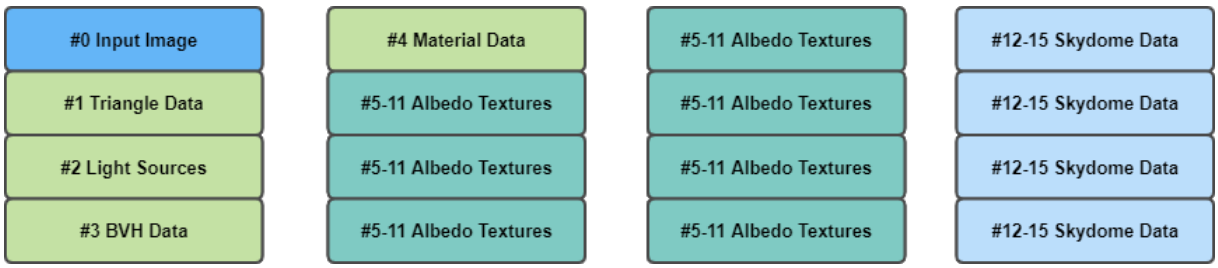

Figure 11: The purpose of each GL texture storage in WebGL path tracer.

We can see each GL texture as a one-dimensional array of vectors (*vec3* and *vec4* types are used for data transferring); even if we actually deal with two-dimensional textures with depth. Consequently, there is a need to flatten our scene object data into multiple arrays of such vectors.

Let us consider an example of how to pack triangle data into GL texture. One triangle contains three vertexes, three UV values, normal vector, triangle ID, material ID. We can pack this data into seven *vec3*. To store n triangles we need $7 \times n$ vectors. Visual representation of triangle data packed into an array of vectors which is stored inside GL texture is provided in Figure 12.

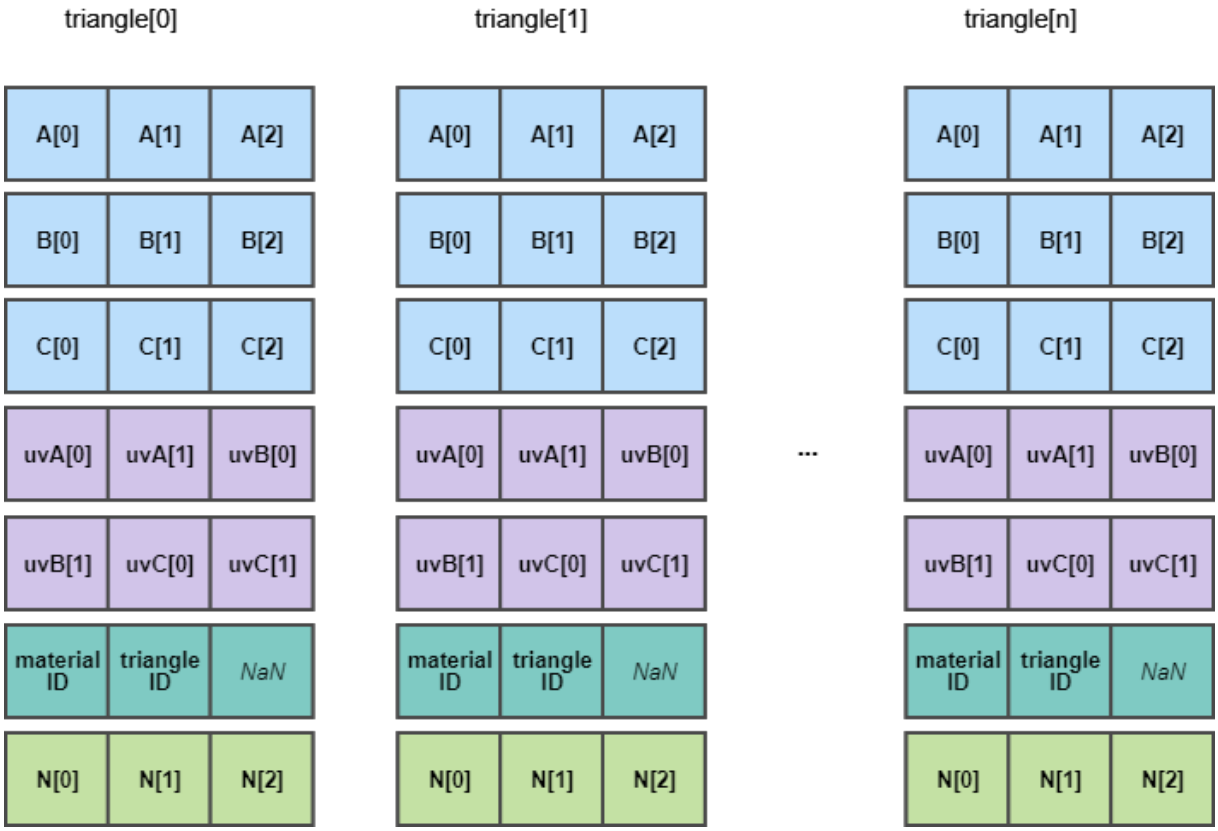

Figure 12: Triangle data packed into GL texture. *A*, *B*, *C* - vertexes; *uvA*, *uvB*, *uvC* - u, v values per vertex, *N* - normal.

Scene data such as light sources, BVH, materials are processed with the same logic - break each object into multiple vectors and put them into the array; store it inside GL texture. For before-mentioned scene components (including triangles) we use arrays of *vec3* since most of the data is naturally represented as vectors of 3 members (mostly vertexes).

Complex scenes can contain various materials, and each of them can have albedo texture (image depicting an object color). Whenever a number of albedo images exceed the maximum available amount of GL textures (7 units in this case), occurs a problem on how to fit that data. We propose to merge multiple albedo images and store multiples per one GL texture. Consequently, that arises another problem how to track where (in which GL texture and in which position) particular image is stored. In order to store multiple albedo images per GL texture, we introduce two variables per material for tracking:

- *Albedo Texture Offset* - indicates in which GL texture is particular albedo image stored.
- *Albedo Pixel Offset* - serves as a pointer for the first pixel of the albedo image inside the given GL texture.

On top of that we store *width* and *height* of each albedo image. This approach let us stitch multiple albedo images into one array of *vec4* and store inside the GL texture. Before-mentioned variables are used to fetch the albedo images inside the GLSL shader. It is also

important to mention that we do no split one albedo image per multiple GL textures, so in particular cases, empty memory gaps inside the GL textures can occur neither over-sized images cannot be stored.

The last component of the scene which we need to fit into GL textures is skydome. Complexity occurs when dealing with skydome data which exceeds the maximum size of one GL texture. In this case, we need to split complex skydome data per multiple GL textures (maximum four units available in the system for this purpose, all with the same size). In this case, it is simple to split data between multiple GL textures since skydome is just one massive data array. We need to store skydome *width* and *height* in order to fetch this data inside the GLSL shader (GL texture size is also known). The data split algorithm is straightforward - we divide skydome colors array into chunks which size equals the GL texture size; the last residual chunk is typically smaller.

6.2.2. Data Fetching

Once all the scene data is packed and stored inside the GL textures, the system can dispatch *path-tracer* fragment shader. Inside the shader, we need to fetch scene data from GL textures and use it for calculations. There is implemented generic GLSL function to retrieve vector data from GL texture (implementation provided in Listing 1).

```
1    vec4 getValueFromTexture(sampler2D texture, int index, int size, int
     sizePower) {
2        ivec2 uv = ivec2(
3            index & size - 1,
4            index >> sizePower
5        );
6
7      return texelFetch(texture, uv, 0).rgba;
8    }
```

Listing 1: Function to fetch a vector by index from the specified GL texture with an arbitrary size (which must be power 2).

It is important to note that each before-mentioned scene component has its structure representation inside the GLSL shader and a function to compound a particular structure by fetching data from GL texture. With the help of *getValueFromTexture(...)* function, we fetch all vectors which form the particular entity, and then we compound its structure (e.g. for the first triangle in the list we fetch first seven vectors, recall Figure 13). This method allows us to deal with complex scene data conveniently, even inside the shader. Code related with data fetching and scene component structures building is encapsulated in *src/shaders/common/data-fetch.glsl* file.

The only different components are albedo images and skydome. In these cases, there are no structures representing entities implemented. It is not necessary because we deal with only one pixel at the moment, and there is no need to fetch full data. We just simply need to calculate *pixel ID* and fetch it with a help of *getValueFromTexture(...)* function.

6.3. Acceleration Structure

In LightHouse CUDA path tracer, acceleration structure is handled via Optix API. Although, while developing with WebGL, we need to implement acceleration structure building and traversal algorithms from scratch. This particular task creates additional complexity for our project since this part of the application cannot be ported from LightHouse, nor any third party software cannot be applied. According to our literature research, we have chosen to implement a BVH acceleration structure for WebGL path tracer (Section 3.1).

It is worth noting that acceleration structure building and traversal algorithms can be adjusted according to the current state-of-the-art since this part of the WebGL path tracer is independent of WebGL functionality itself. The important aspect when upgrading the acceleration structure is the amount of storage needed to transfer it from JavaScript side to a GLSL shader.

6.3.1. Building

In our path tracer, BVH building is performed after scene geometry has been loaded. BVH building algorithm is based on a greedy algorithm with SAH [MB90]. Each BVH node has two children nodes unless it is a leaf. To determine the best split of primitives into the bounding boxes, we consider all three axes. On top of that, for each axis, we divided space into ten bins and split all primitives into the corresponding bins (considering the center position of each primitive). Finally, we evaluate each combination of bins by comparing the SAH values. The lowest SAH value for the combination indicates that this is the optimal split (the lower SAH, the bigger probability of ray hitting a primitive inside the bounding box).

Once the BVH building process has been completed, all bounding boxes are put into the list, starting with the root node. Consequently, we use the before-mentioned method described in the Data Transfer chapter to pack all bounding boxes into GL texture and transfer them to GLSL shader. The packed BVH data representation provided in Figure 13.

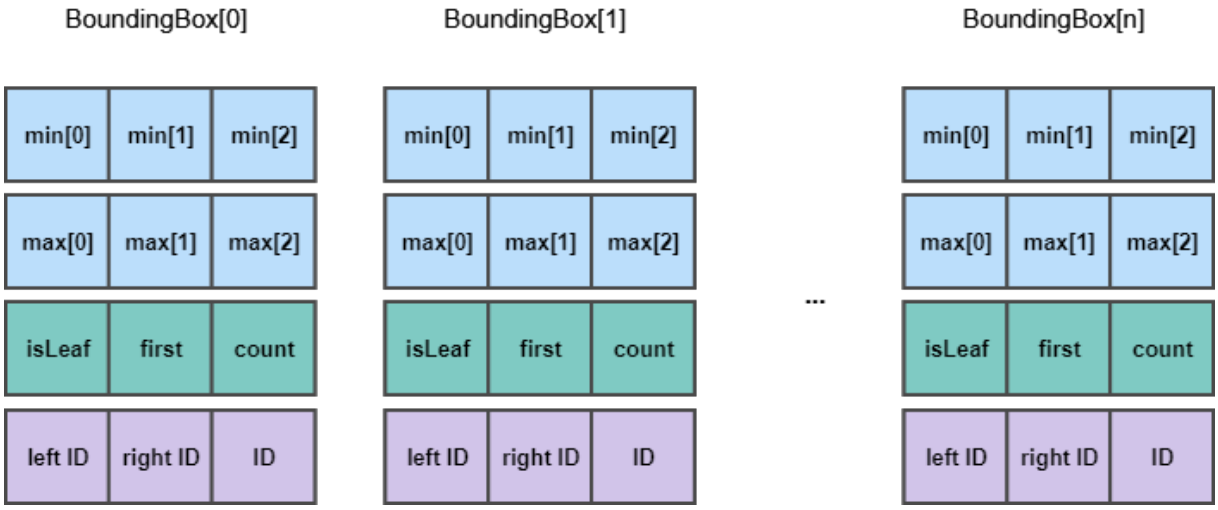

Figure 13: BVH data packed into GL texture (*first* - index of the first primitive inside the node; *count* - number of primitives; *left ID*, *right ID* - children node indices).

6.3.2. Traversal

BVH traversal is implemented inside the *path-tracer* fragment shader. Traversal process has to be performed to determine which primitive (and which point on it) is intersected by the ray. There are two types of traversal: for primary (shot from a camera) and for shadow (bounced from the intersection point) rays. Although, the only difference for shadow rays is that we can sometimes terminate the process earlier - as soon as an intersection was found.

Our BVH traversal implementation is based on ordered stack-traversal algorithm [ref]. The system performs BVH traversal by popping a node (bounding box) from a stack and determining if a ray hits it. In case a leaf node was hit - there is ray-primitive intersection performed. Otherwise, if a ray hits an internal node, we have to determine which children node to traverse next. This is solved by calculating the distance to both children and pushing the *nearest* node into the stack. Whenever we need to pop a node from the stack, the system fetches the particular bounding box from the GL texture and forms the entity (structure depicting bounding box inside the GLSL shader is provided in Listing 2). Once the stack becomes empty, the traversal process is terminated.

```
1    struct BoundingBox
2    {
3        vec3 min, max;
4
5        bool isLeaf;
6        int first, count;
7        int left, right, id;
8    };
```

Listing 2: BVH bounding box representation inside the GLSL shader.

6.4. Path Tracing

Path tracing algorithm explained in Section 2.1 is implemented inside the *path-tracer* vertex and fragment GLSL shaders. The vertex shader (*path-tracer.vertex.glsl*) is mainly responsible for the generation of the rays based on a pixel position. A pivotal part of the path tracing algorithm is implemented inside the fragment shader (*path-tracer.fragment.glsl*). Key parts of the implementation include:

- Data-fetching from GL textures (Section 6.2.2);
- Ray-primitive intersection by traversing BVH, including ray intersection with triangles and bounding-boxes (Section 6.3.2);
- Skydome sampling;
- Texture mapping and barycentric coordinates calculation;
- Pixel shading and occlusion detection, including ray-sphere intersection for light-sources (Section 6.4.1);
- Random number generator (Section 6.4.2).

6.4.1. Shading

Once the program provides ray-primitive intersection result (hit-point, texture, normal), we need to determine the final color of a pixel taking the light sources into account; this process is called **shading**. It is important to point out that this part of the algorithm was directly ported from LightHouse to the GLSL shader. In order to render photorealistic images and keep the result as similar as possible to the LightHouse, robust techniques such as Multiple Importance Sampling and Next Event Estimation were ported (explanation of some relevant techniques provided in Section 2.1.2). The porting process for this part of the application was trivial since the programming language differences between CUDA and GLSL were insignificant for this particular case. In order to successfully port the CUDA code, it was necessary to ensure that our GLSL shader provides the required input to implement Lambert BSDF.

6.4.2. Random Number Generator

One of the most important parts while implementing the path tracing algorithm is the generation of random numbers in the GLSL shader. They are used to generate a random direction for a bounced ray, also to determine the random light source. Random number generation needs to be unbiased. Otherwise, that creates non-realistic noise patterns on the rendered image. In this project, we try to produce visually similar output as LightHouse. Consequently, the random number generator has been ported from LightHouse to avoid different noise patterns on the rendered image in our application. The final implementation is presented in Listing 3 where `unifySeed(...)` serves as an utility function to prepare the `seed` which is later randomized by `randomFloat(...)` function. The primary `seed` comes from the JavaScript application side, where a new number is randomized for each frame.

```
uint wangHash(uint s) {
    s = (s ^ 61u) ^ (s >> 16u),
    s *= 9u,
    s = s ^ (s >> 4u),
    s *= 0x27d4eb2du,
    s = s ^ (s >> 15u);

    return s;
}

uint unifySeed(uint seed) {
    uint xpos = uint(gl_FragCoord.x * resolution[0]);
    uint ypos = uint(gl_FragCoord.y * resolution[1]);
    uint pixelIdx = xpos + ypos * uint(resolution[0]);

    return wangHash(pixelIdx + seed);
}

uint randomInt(inout uint s) {
    s ^= s << 13u,
    s ^= s >> 17u,
    s ^= s << 5;

    return s;
}

float randomFloat(inout uint seed) {
    return float( randomInt(seed) ) * 2.3283064365387e-10;
}
```

Listing 3: Random Number Generator ported from CUDA to GLSL.

7. Results

The main focus of this chapter is to provide results which will help to answer the research questions formulated in Section 1.1. The first research question is mainly answered throughout the case study by successfully porting LightHouse to WebGL. The second question requires a performance evaluation, which we provide in this chapter, before drawing final conclusions.

7.1. Test Data

For our experiments, we use several scenes. Details about each scene setup are provided in Table 2; the view of each scene from an arbitrary camera position is shown in Figure 14. To cover most cases, we use scenes of varying complexity. The Avocados scene is set to be simple - just a few primitives with no skydome loaded; it is used to compare performance amongst different mobile browsers and devices (taking into account low GPU memory limits on such hardware). Pica Room contains a larger number of primitives, although most of them do not have textures; consequently, it is ideal for testing ray-primitive intersection performance - a typical path tracing bottleneck. Sponza contains a large amount of textured primitives and has a complex sky; it will be used as a stress-test on best-performing browsers and devices.

Scene	**Primitive Count**	**Skydome**
Avocados	2×682	*none*
Pica Room	76274	*Triangular Sky*, 26.8 MB
Sponza	262267	*City Surroundings*, 36.5 MB

Table 2: Scenes for path tracers performance comparison.

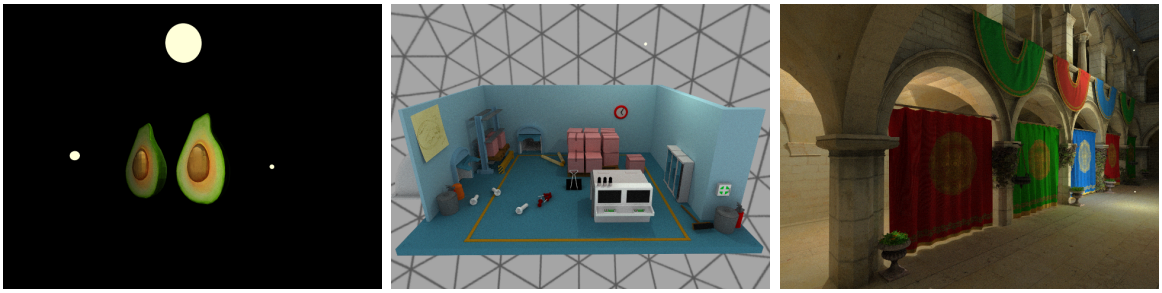

Figure 14: Example view of each scene: Avocados, Pica Room, Sponza.

7.2. Test Platforms

It is important to understand how viable WebGL is for implementing complex graphics applications. In order to gain insight, we perform experiments by running our path tracer on various devices and browsers. Our application performance mainly depends on scene complexity and device capability. We choose the combination of scenes and platforms which covers edge cases and also shows the medium-ground. For all experiments, we provide an

identical scene set-up (camera position, light sources, geometry, textures, skydome) in order to get objective results.

Platform	**Device**	**Web Browsers**
Windows PC	Intel i5-7300HQ CPU (2.5 GHz) CPU NVIDIA GeForce GTX 1050 GPU	Google Chrome Mozilla Firefox Microsoft Edge Beta
Windows PC	Intel i5-7300HQ CPU (2.5 GHz) CPU Intel(R) HD Graphics 630 GPU	Google Chrome Mozilla Firefox Microsoft Edge Beta
Windows PC	AMD Ryzen 5 3600 CPU AMD Radeon RX580 GPU	Google Chrome Mozilla Firefox Microsoft Edge Beta
Linux PC	i7-7700HQ (2.80 GHz) CPU GeForce GTX 1050 Ti GPU	Google Chrome Mozilla Firefox
Android Mobile	Samsung Galaxy A3 (2017) ARM Mali-T830 GPU	Google Chrome Mozilla Firefox Opera Samsung Internet
Android Mobile	Huawei Honor 7 ARM Mali-T830 GPU	Google Chrome
Android Mobile	Samsung Galaxy S8 ARM Mali-G71 GPU	Google Chrome
Android Mobile	Nokia 6.1 Plus Adreno 509 GPU	Google Chrome
Android Mobile	Xiaomi Pocophone F1 Adreno 630 GPU	Google Chrome

Table 3: Platforms for WebGL Path Tracer performance evaluation. Apple Safari browser was excluded from our experiments because it currently does not support WebGL 2.0.

It is worth to mention that we disable *vsync* for browsers on the PC platform while rendering the Avocados scene. This is necessary because application performance typically exceeds the 60 FPS cap set on various web browsers: Google Chrome, Mozilla Firefox, Microsoft Edge Beta.

7.3. Test Scenario

The performance of WebGL Path Tracer is measured by rendering each scene from Table 2 on each platform from Table 3. Rendering performance is measured in frames per second (**FPS**) and frame-rate (**ms/frame**). We measure the performance of **500** subsequent frames, skipping the first **20** frames to allow the caches to warm up. For each test, we determine the standard deviation to depict rendering smoothness; besides, the average FPS

is calculated. Path tracer performance strongly depends on the resolution of the image being rendered. In order to keep experiments consistent, the resolution is set to **640** $\times$ **480** regardless of the testing platform. On top of that, BVH building time for each scene is recorded; it depicts waiting time before the scene can be rendered.

The second research question requires comparison of WebGL path tracer performance with the native CUDA path tracer. For this, we measure the performance of both applications on the same platform. We choose a consumer-grade PC from Table 3 (Windows PC with Intel i5-7300HQ CPU (2.5 GHz) CPU and NVIDIA GeForce GTX 1050 GPU). In order to measure LightHouse performance, we use an identical set-up for each scene from Table 2. The scene is rendered on the *Optix Prime* core by evaluating 100 subsequent frames after cache warm-up. Finally, we will calculate average refresh-rate in FPS.

7.4. Performance Evaluation Analysis

As established in Section 7.3, we have investigated how the performance of our WebGL path tracer compares on different platforms. In Appendix 9 we provide experimental results (Table 4 - Table 12) which present WebGL path tracer application performance measurements on a particular device and a browser. Note that for Windows PC with GeForce GTX 1050 (Table 4) BVH building times were averaged (10 test cases for each scene). Furthermore, we have calculated standard deviation for BVH construction on this platform.

In the following subsections, we analyse performance measurement data. The performance of the WebGL path tracer will be examined from a few different aspects: general refresh-rate overview, BVH building speed, performance on PC and mobile platforms, comparison versus native CUDA path tracer.

7.4.1. WebGL Path Tracer Performance Overview

In this subsection, we provide a general overview of WebGL path tracer rendering speed. Figure 15 represents refresh-rate per scene on all platforms (PC and mobile combined). From these charts, we can observe explicit dependency between GPU hardware capability and WebGL path tracer's rendering performance. Simple scenes are rendered at high refresh-rates (200 FPS can be reached on Avocados scene), however, complex scenes like Sponza only achieve 30 FPS on a Windows PC with AMD Radeon RX580 GPU. All mobile devices yielded lower rendering performance results - up to 10 FPS for Avocados scene, and only a few phones were able to render Pica Room and Sponza scenes on marginally low refresh-rate.

Figure 15: WebGL Path Tracer refresh rate on Google Chrome for various platforms.

7.4.2. BVH Building Performance

In Figure 16, we present BVH building times of WebGL path tracer. This measurement depicts our application performance of its JavaScript side. In this chart, we observe similar trends as rendering performance in Figure 15, although BVH building time gap between PC and mobile platform is not that big as for rendering speed. The time before the scene can be displayed depends on its complexity. Consequently, acceleration structure for Avocados and Pica Room room is computed much faster than for Sponza scene. BVH building time varies from 30 ms to 13 s on the PC platform, and from 50 ms to 36 s on Mobile devices depending on scene complexity and device computational power.

In order to assess the consistency of acceleration structure construction, we recorded average BVH building time on Windows PC with GeForce GTX 1050 on multiple browsers for each scene (Figure 17). Any severe fluctuations were not noticed. However, building times on Mozilla Firefox are not that persistent as for Google Chrome and Microsoft Edge Beta. An interesting observation is that the standard deviation for Sponza is considerably lower even though it contains roughly three times more primitives than Pica Room scene.

Figure 16: WebGL Path Tracer BVH building times on Google Chrome for various platforms.

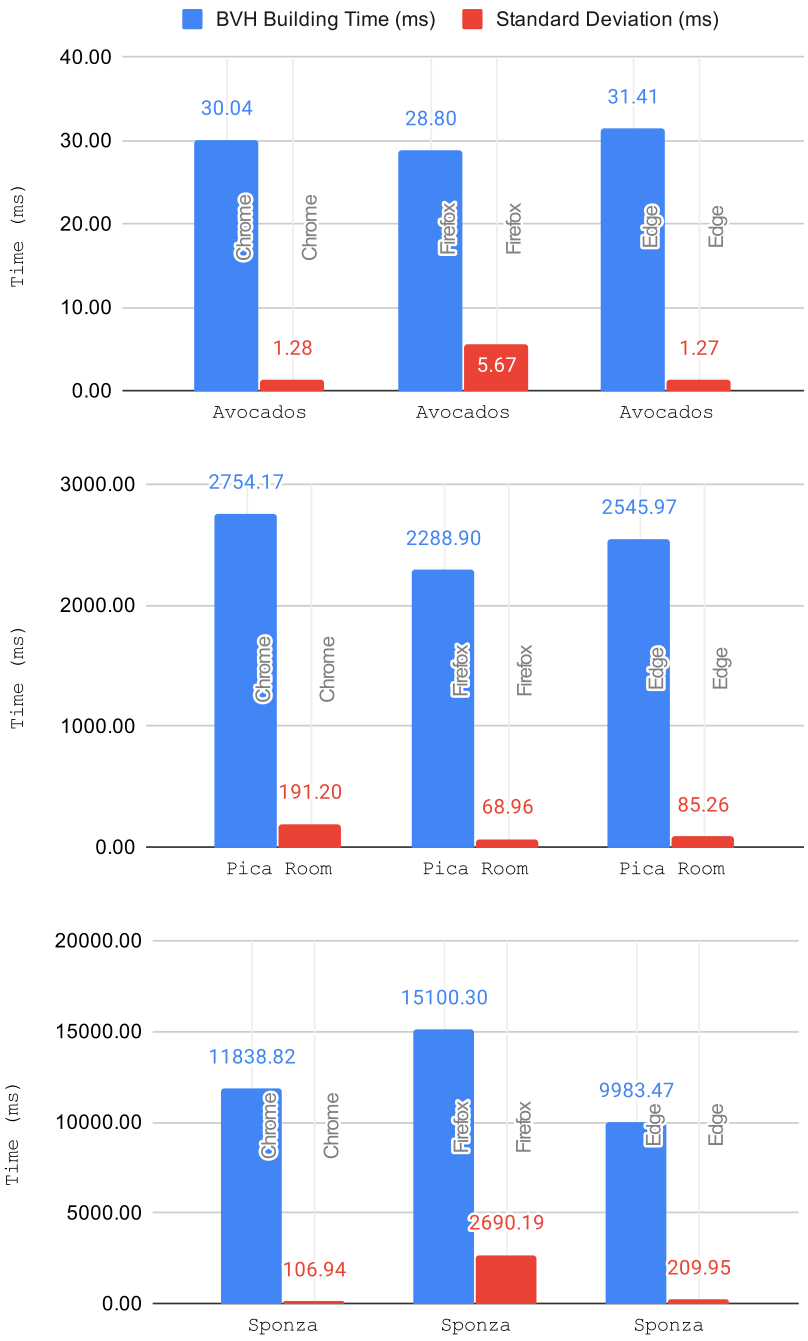

Figure 17: WebGL Path Tracer BVH building times with standard deviation on Windows PC with GeForce GTX 1050.

7.4.3. PC Platform

In this subsection, we represent WebGL Path Tracer performance experiment results on several consumer-grade computers. Figure 18 shows rendering time per frame with standard deviation recorded on a few different browsers.

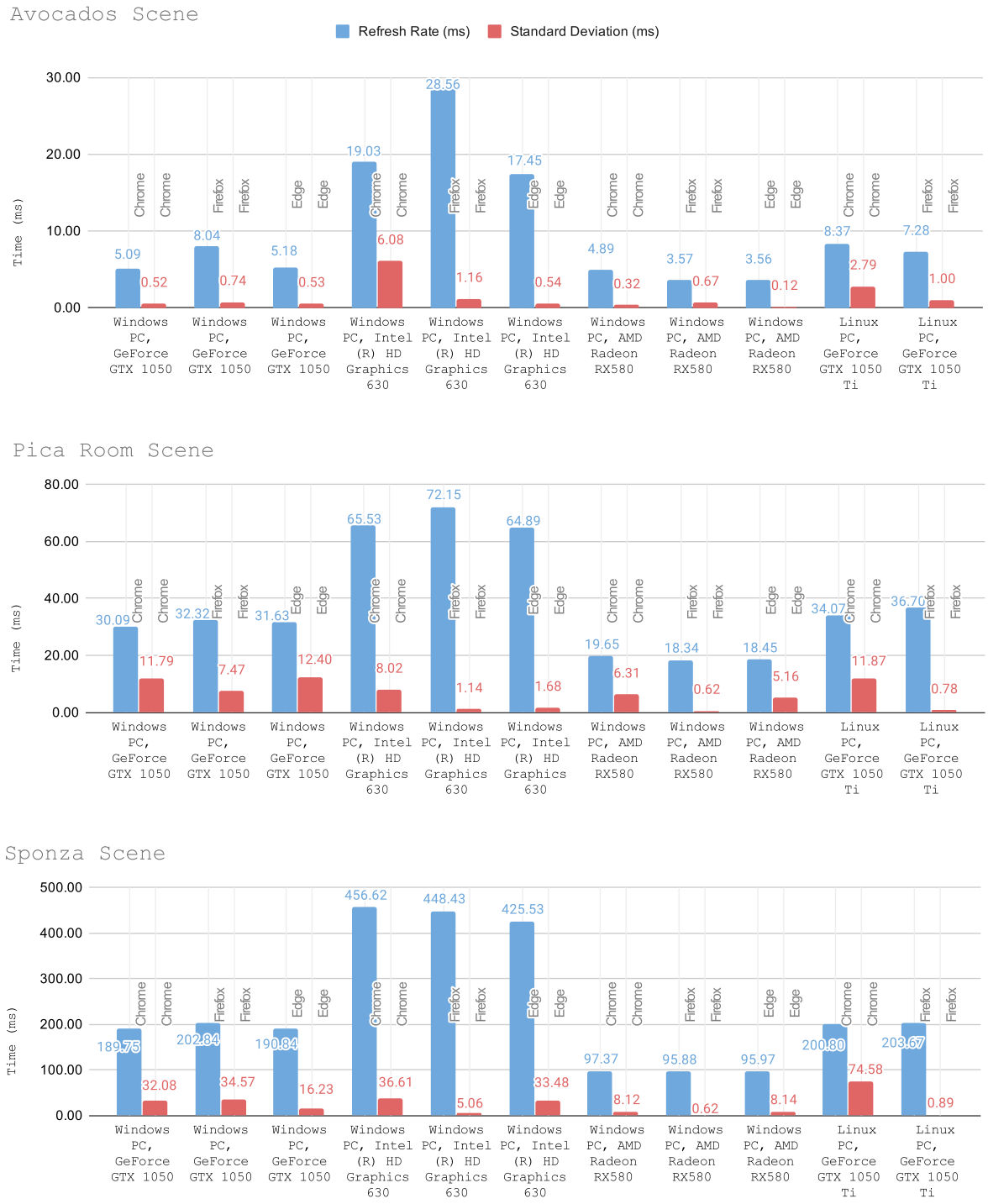

Figure 18: WebGL Path Tracer refresh rate with standard deviation on PC platform

One notable aspect is that rendering time on Google Chrome tends to have more fluctuation than on other browsers. Generally, we observe that rendering performance is loosely de-

pendent on the browser (variety decreases when scene complexity increases). As already observed in subsection 7.4.1, current data also confirms that rendering performance is strongly affected by scene complexity and hardware capability regardless of the browser.

7.4.4. Mobile Platform

Similarly, as in the previous subsection, we represent the application's performance results on a mobile platform (Figure 19). For Samsung Galaxy A3 Android mobile phone we have performed performance tests on multiple web browsers (including Google Chrome, Mozilla Firefox, Opera, Samsung Internet). This experiment shows that WebGL path tracer can run on a variety of mobile web browsers, respectively refresh-rate remains consistent amongst them. Another interesting point is that rendering time tends to deviate significantly less on phones with Adreno 509/630 GPU's while on other mobile devices we notice enhanced refresh-rate fluctuations. Throughout experiments, the only device (Samsung Galaxy S8 with ARM Mali-G71 GPU) was able to load and render Sponza scene (2702.70 ms per frame with 1391.21 standard deviation).

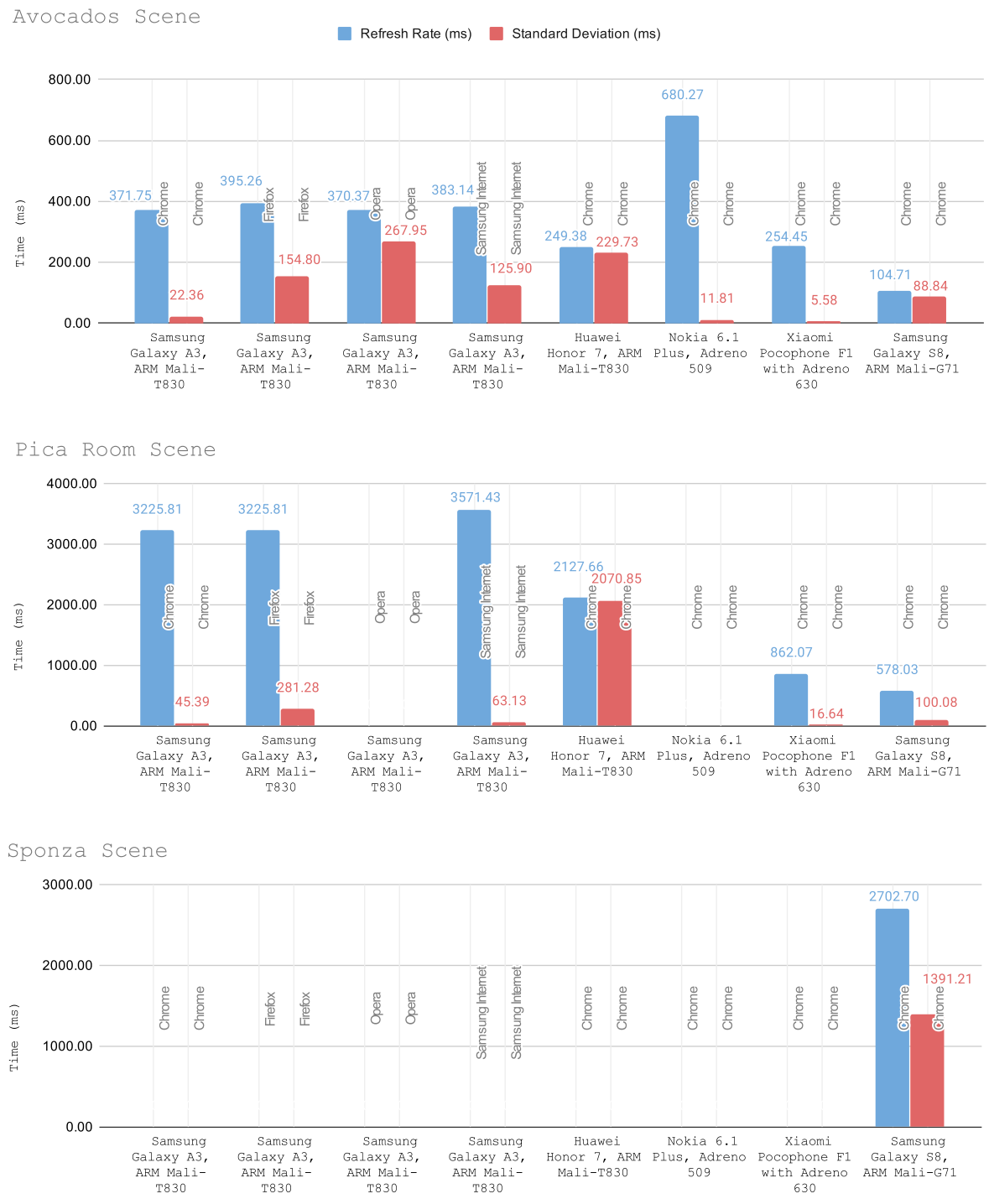

Figure 19: WebGL Path Tracer refresh rate with standard deviation on Mobile platform

7.4.5. LightHouse

As established in Section 7.3, we have evaluated LightHouse performance on Windows PC with GeForce GTX 1050. In order to answer the second research question, we provide a performance comparison between WebGL and CUDA path tracers (Figure 20). One of the most meaningful trends in this chart is the consistency of LightHouse refresh-rate regardless of scene complexity while WebGL path tracer performance significantly drops when scene complexity grows. We can observe that WebGL path tracer on Avocados scene can maintain high refresh-rates and even surpass CUDA path tracer. However, rendering Pica Room scene on LightHouse is roughly two times faster than on WebGL path tracer; for Sponza scene this gap increases even more drastically - LightHouse outperforms up to 15 times.

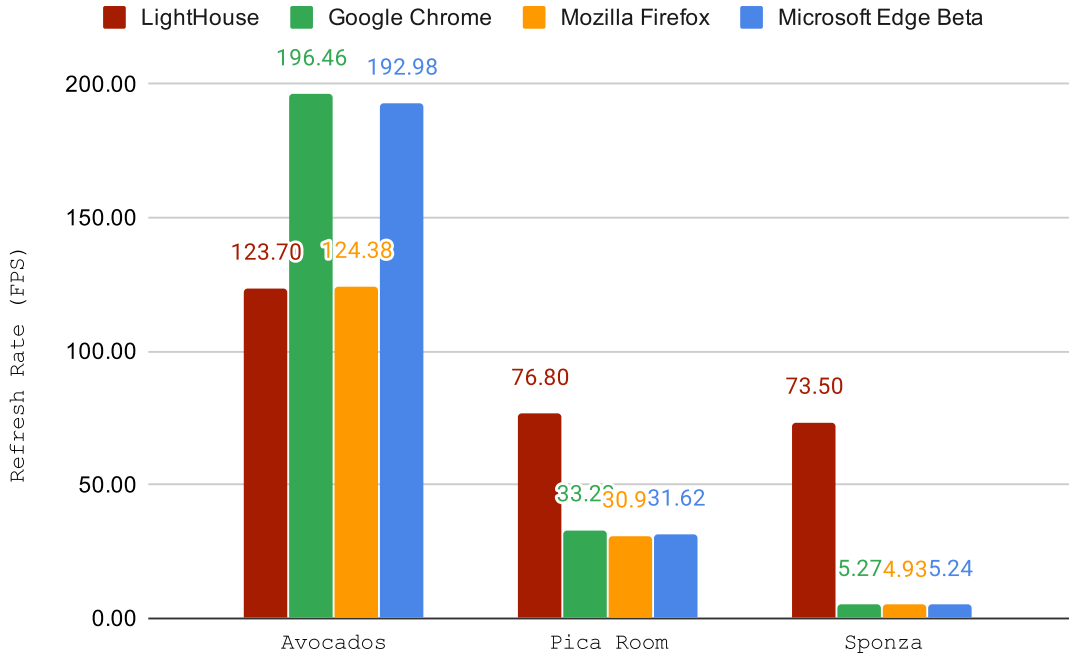

Figure 20: WebGL Path Tracer versus LightHouse on Windows PC with GeForce GTX 1050.

8. Conclusions

In this master thesis, we addressed the problem that path-tracing software is typically implemented as a native application, which limits the accessibility of this type of programs. Two research questions (Section 1.1) were formulated to investigate if a web-based implementation is feasible. We conducted literature research in Section 3 to establish state-of-the-art in path-tracing and web technologies for GPU software development. To answer the research questions, we performed a case study by porting LightHouse to WebGL.

To prove that an industry-strength path tracer can run on a website, we successfully implemented a WebGL path tracer, which is functionally comparable to LightHouse. Our application is capable of rendering complex scenes in a real-time. This answers the first research question: a path tracer can indeed be implemented as a web application. However, WebGL suitability for such a task can be questionable. In Section 7.4, we investigated how the WebGL path tracer performs on multiple platforms. From this analysis, we observe that the WebGL path tracer performs worse on complex scenes and low-end devices. Besides, one of the most popular browsers (Apple Safari) does not fully support WebGL 2.0. This creates an obstacle that our application cannot be accessed for a significant number of users.

In order to answer how the performance of a real-world medium-scale rendering application implemented in WebGL compares to native performance, we examined LightHouse versus the developed web path tracer in Subsection 7.4.5. The path tracer implemented with WebGL can run on the web with a sufficient frame-rate on scenes with low to medium complexity (such as Avocados and Pica Room). Even though rendering complex scenes like Sponza on WebGL is feasible, the web path tracer cannot compete with a native CUDA path tracer when considering more complicated textured scenes.

Concluding, we can state that WebGL is a viable option for a web-based path tracer implementation when rendering low-to-medium complexity scenes. However, it remains a challenging task to achieve decent performance on complex scenes.

8.1. Discussion

One aspect of WebGL that could be considered a flaw when implementing complex graphics applications is data transfer from JavaScript to the GLSL shader. Currently, there is a restriction on how many GL textures and uniforms are available. This limits the amount of data which can be accessed inside the shader. Our approach is to dedicate particular GL textures for a particular type of data. This results in memory gaps in the available GL textures. This happens because different scenes contain varying amounts of components (e.g. one scene contains multiple non-textured primitives while other contains few heavily-textured primitives). To overcome this issue, we either need more GL textures available, or the ability to read data inside the GLSL shader from a buffer (similar to CUDA). Both solutions require WebGL to evolve and cannot be solved by the end-programmer with the current WebGL 2.0 version.

The wavefront algorithm is typically desired for GPU path tracing. However, when working with WebGL, the wavefront approach is hardly adaptable. In order to implement it, we need to be able to output an arbitrary amount of data from the GLSL shader (e.g.

we generate n shadow rays which have to be tested against scene geometry in the next step, where n is estimated at a runtime). This task is not trivial because we cannot define the output texture size at the GLSL shader's runtime. Although, we can define an arbitrary size for output texture before running shader, this is not sufficient to implement the algorithm in a wavefront manner because the output data size is typically unknown before execution.

Overall, the implementation of the WebGL path tracer was successful, even considering the limitations. Our application is able to load scene geometry, pass it to the GLSL shader, perform path tracing and draw the final rendered image on the *canvas* which converges over time. Finally, considering the current state-of-the-art in path tracing, we see a wavefront approach as the ultimate goal for our study case.

8.2. Future Work

We have shown that WebGL is an appropriate technology to implement complex graphics applications for the web to a certain extent. However, some limitations still exist in the WebGL API, as well as in our implementation which affect the flexibility and performance of the application.

One of the most significant drawbacks of WebGL in our case is the inability to transfer a large amount of data to the GLSL shader conveniently. That is a topic for future work and the issue may require advancing WebGL API by enabling new ways of transferring data to the shader. Another option could be to improve the current application by implementing dynamic decision making on 'how many' and 'what size' GL textures to use for particular scene components, nevertheless, this approach would only be an optimisation and not an actual solution.

With the current implementation, before the scene can be rendered, we need to download its data from a remote server. In addition, we need to build a BVH over the loaded geometry. This results in a significant waiting time before rendering. As a solution to this problem, we could implement a pre-built BVH structure for each scene to avoid BVH building times. This would require the implementation of a convenient data structure to store a BVH. On the other hand, if we consider a dynamic (animated) scene, this approach becomes obsolete, and the top-level BVH should be the desired solution.

9. References

[AWY00] Charu Aggarwal, Joel Wolf, and Philip Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 11, 08 2000.

[BU19] Denis Bogolepov and Danila Ulyanov. LightTracer.org Path Tracer. `https://lighttracer.org/`, 2019. Accessed: 2019-10-21.

[DHK08] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, pages 1225–1233, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[DJAI15] Jeff Gilbert (Mozilla Corp.) Dean Jackson (Apple Inc.). Editor's Draft Fri Oct 26 21:28:35 2018 -0400, 2015.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[Kaj86] James T. Kajiya. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[LKA13] Samuli Laine, Tero Karras, and Timo Aila. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143, New York, NY, USA, 2013. ACM.

[Lof] Erich Loftis. WebGL Path Tracer implemented on THREE.js framework. `https://github.com/erichlof/THREE.js-PathTracing-Renderer`. Accessed: 2019-02-17.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.

[MPH16] W. Jakob M. Pharr and G. Humphreys. *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

[Net] Mozilla Developer Network. Browser storage limits and eviction criteria. `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria`. Accessed: 2019-03-27.

[Nik] Simon Niklaus. Webassembly Raytracer. `https://sniklaus.com/blog/raytracer`. Accessed: 2019-02-17.

[Par12] Tony Parisi. *WebGL: Up and Running: Building 3D Graphics for the Web.* Safari Books Online. O'Reilly Media, 2012.

[PB03] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[PBD+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional Representation for Fast Rendering of Complex Scenes. *SIGGRAPH Comput. Graph.*, 14(3):110–116, July 1980.

[Ter] Rye Terrell. Caffeine path tracing demo. `https://wwwtyro.net/2018/02/25/caffeine.html`. Accessed: 2019-02-17.

[Wal] Evan Wallace. WebGL Path Tracing. `http://madebyevan.com/webgl-path-tracing`. Accessed: 2019-02-17.

[Wan99] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999.

[Wat18] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of*

the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, pages 53–65, New York, NY, USA, 2018. ACM.

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. Getting Rid of Packets âĂŞ Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *IEEE Symposium on Interactive Ray Tracing*, 2008.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.*, 26(1), January 2007.

[YKL17] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics*, HPG '17, pages 4:1–4:13, New York, NY, USA, 2017. ACM.

[Zak11] Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM.

Appendices

A. WebGL Path Tracer Performance Measurement Tables

A.1. PC Platform

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Average Time*	**Standard Deviation** *of BVH building time*
Google Chrome				
Avocados	196.46	0.52 ms	30.04 ms	1.28 ms
Pica Room	33.23	11.79 ms	2754.17 ms	191.2 ms
Sponza	5.27	32.08 ms	11838.82 ms	106.94 ms
Mozilla Firefox				
Avocados	124.38	0.74 ms	28.8 ms	5.67 ms
Pica Room	30.94	7.47 ms	2288.9 ms	68.96 ms
Sponza	4.93	34.57 ms	15100.3 ms	2690.19 ms
Microsoft Edge Beta				
Avocados	192.98	0.53 ms	31.41 ms	1.27 ms
Pica Room	31.62	12.4 ms	2545.97 ms	85.26 ms
Sponza	5.24	16.23 ms	9983.47 ms	209.95 ms

Table 4: Performance measurement table. Windows PC with GeForce GTX 1050.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	52.56	6.08 ms	28.36 ms
Pica Room	15.26	8.02 ms	2456.2 ms
Sponza	2.19	36.61 ms	10396.02 ms
Mozilla Firefox			
Avocados	35.02	1.16 ms	34.0 ms
Pica Room	13.86	1.14 ms	2846.0 ms
Sponza	2.23	5.06 ms	12979 ms
Microsoft Edge Beta			
Avocados	57.3	0.54 ms	30.01 ms
Pica Room	15.41	1.68 ms	3990.61 ms
Sponza	2.35	33.48 ms	16436.34 ms

Table 5: Performance measurement table. Windows PC with Intel(R) HD Graphics 630.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	204.53	0.32 ms	32.0 ms
Pica Room	50.88	6.31 ms	1681.27 ms
Sponza	10.27	8.12 ms	7117.3 ms
Mozilla Firefox			
Avocados	280.11	0.67 ms	26.0 ms
Pica Room	54.52	0.62 ms	1378.0 ms
Sponza	10.43	0.62 ms	5994.0 ms
Microsoft Edge Beta			
Avocados	280.8	0.12 ms	35.41 ms
Pica Room	54.2	5.16 ms	1709.6 ms
Sponza	10.42	8.14 ms	7129.21 ms

Table 6: Performance measurement table. Windows PC with AMD Radeon RX580.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	119.5	2.79 ms	37.06 ms
Pica Room	29.35	11.87 ms	3732.29 ms
Sponza	4.98	74.58 ms	13115.5 ms
Mozilla Firefox			
Avocados	137.4	1.0 ms	46.0 ms
Pica Room	27.25	0.78 ms	3407.0 ms
Sponza	4.91	0.89 ms	13927.0 ms

Table 7: Performance measurement table. Linux PC with GeForce GTX 1050 Ti.

A.2. Mobile Platform

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	2.69	22.36 ms	213.1 ms
Pica Room	0.31	45.39 ms	22540.7 ms
Sponza	*N/A*		
Mozilla Firefox			
Avocados	2.53	154.8 ms	245.0 ms
Pica Room	0.31	281.28 ms	20802.0 ms
Sponza	*N/A*		
Opera			
Avocados	2.7	267.95 ms	198.7 ms
Pica Room	*N/A*		
Sponza	*N/A*		
Samsung Internet			
Avocados	2.61	125.9 ms	213.7 ms
Pica Room	0.28	63.13 ms	22973.6 ms
Sponza	*N/A*		

Table 8: Performance measurement table. Samsung Galaxy A3 with ARM Mali-T830.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	4.01	229.73 ms	254.0 ms
Pica Room	0.47	2070.85 ms	24525.0 ms
Sponza	*N/A*		

Table 9: Performance measurement table. Huawei Honor 7 with ARM Mali-T830.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	1.47	11.81 ms	147.0 ms
Pica Room	*N/A*		
Sponza	*N/A*		

Table 10: Performance measurement table. Nokia 6.1 Plus with Adreno 509.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	3.93	5.58 ms	55.4 ms
Pica Room	1.16	16.64 ms	6308.4 ms
Sponza	*N/A*		

Table 11: Performance measurement table. Xiaomi Pocophone F1 with Adreno 630.

Scene	**Refresh Rate** *in FPS*	**Standard Deviation** *of Refresh Rate*	**BVH Building** *Time*
Google Chrome			
Avocados	9.55	88.84 ms	96.7 ms
Pica Room	1.73	100.08 ms	9558.6 ms
Sponza	0.37	1391.21 ms	35889.2 ms

Table 12: Performance measurement table. Samsung Galaxy S8 Plus with ARM Mali-G71.