Real-time path tracing of small scenes using WebGL

Master's thesis in Computer Science – algorithms, languages and logic

Martin Nilsson
Alma Ottedag

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Real-time path tracing of small scenes using WebGL

Martin Nilsson, Alma Ottedag

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Real-time path tracing of small scenes using WebGL
Martin Nilsson, Alma Ottedag

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisor: Magnus Pettersson, RapidImages AB
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image rendered by the path tracer developed in this report.

Typeset in LaTeX
Gothenburg, Sweden 2018

Real-time path tracing of small scenes using WebGL
Martin Nilsson, Alma Ottedag
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Monte Carlo path tracing is becoming increasingly viable as a method for rendering global illumination in real-time. We explored the potential of using path-tracing and WebGL to rendering real-time 3D graphics in a web browser. The project focused on rendering small scenes where objects are dynamically translated, rotated, and scaled. We examined the performance of various acceleration data structures (ADS) including 3D grids, irregular grids, and bounding volume hierarchies. To reduce the noise inherent in path-traced images, we separated the lighting into several lighting terms and applied an À-Trous wavelet filter on each term. We explored both the results of splitting the direct and indirect lighting terms and splitting the glossy and diffuse terms. We also applied the surface albedo in a post-processing step to better retain texture details.

On small scenes, we were able to trace 720x540 pixel images at interactive framerates, i.e. above 10hz, at one sample per pixel with a maximum path depth of five. Using per-object bounding volume hierarchies, we can render dynamically changing scenes, e.g. moving objects, at interactive framerates. The noise reduction filter executes in less than 10 milliseconds and is successful at removing noise but over-blurs some image details and introduces some artefacts. We conclude that while real-time path tracing is possible WebGL, there are several caveats of the current version of the WebGL library that makes some state-of-the-art optimisation techniques impractical. For future work, we suggest several approaches for improving the path tracer. For instance, extending the noise reduction filter with temporal accumulation and anti-aliasing, and optimising the encoding of triangles and ADS nodes.

Keywords: Computer Science, Graphics, Ray Tracing, Path Tracing, WebGL

Acknowledgements

We would like to extend thanks to our supervisor Erik Sintorn for the great feedback and guidance. We are also very grateful for all the feedback and support from everyone at RapidImages AB, especially our company advisor Magnus Pettersson.

Martin Nilsson & Alma Ottedag, Gothenburg, June 2018

Contents

List of Figures

List of Tables

1 Introduction

Real-time ray tracing has been a milestone in the computer graphics field for a long time. However, ray tracing has struggled with reaching real-time applications due to its computational cost. With the rapid increase in computing capacity of Graphics Processing Units (GPU), real-time ray tracing has become increasingly viable. Additionally, the introduction of WebGL 2 (Web Graphics Library 2)[1] in 2017 made it possible to run more general-purpose GPU programs on the web. Our thesis explores the use of real-time ray tracing for rendering 3D graphics using WebGL.

The following chapter introduces the topic of the thesis as well as the research question. Firstly, it gives a background to the project and the research field. Secondly, it explains the research question followed by an account of the project limitations.

1.1 Background

Consider an interior-design company desiring collections of hundreds or even thousands of images of furniture on their website. They want each image to show the furniture in a representative environment and from a selection of angles. Using photography for this task could take months to do. Also, it would be challenging to ensure coherence of lighting and angles between different photo shoots. Instead, the company wants to generate these pictures through computer rendering 3D models of their designs.

RapidImages AB attempts to fulfil the need for large amounts of computer-rendered product images. Therefore, the company has created a development tool in a web environment for the design of product items in 3D scenes. The purpose of the tool is for designers to be able to fine-tune scenes, adjust materials and other properties of the product items. The scenes can then be processed into industrial-quality images to be used on, for example, an interior-design company's website.

The development tool uses WebGL to visualise these small scenes in real time. WebGL is commonly used to render graphics on the web. WebGL is also the only tool which can access the Graphics Processing Unit (GPU) from within a web browser.

The issue with the current development tool is that the rendering is not realistic. Presently, the renderer does not adequately visualise several materials and effects. Especially problematic are global illumination effects such as reflections, refraction, colour bleeding, and soft shadows.

[1]https://www.khronos.org/registry/webgl/specs/latest/2.0/

The physically incorrect image output of the current renderer is problematic during the design process. For example, a designer aims to find the perfect material for every part of a chosen product. They both design and apply materials for the product. However, if the lighting in the rasterised image is misleading the designer may end up using trial and error to achieve their desired result.

The current solution to help the designer is the functionality to request a realistic image preview from the development tool. The tool sends the request to a **render farm**: a high-performance system whose purpose is to generate these images. The render farm produces images using an offline renderer, such as V-Ray[2]. However, generating these higher quality previews usually takes several minutes; a *long* delay in feedback for the designer. Therein lies the problem to solve in this thesis.

The company and its customers desire a solution that reduces the visual gap between the workspace for the designer and the requested more realistic previews. More specifically, they wish to find a way to *achieve a more physically accurate visualisation for the designer workspace in real time.* The suggested approach is to replace the current renderer with a real-time path tracing solution, also built in a WebGL-context. Ideally, the renderer removes the need for previews. If not, the renderer could be used to provide faster previews, rendered locally on the client side.

1.2 Problem Statement

The problem considered in this report is how to utilise path tracing in an interactive WebGL renderer. For this report, we define real time as frame rates above 10hz, i.e. a per-frame execution time of less than 100 milliseconds. This problem contains several subproblems. Firstly, what acceleration data structures are best suited for small scenes? Ray tracing using the data structures shall be fast enough to allow for real-time rendering. The construction of the data structures shall also be fast enough for updating the scenes in real time. Secondly, how can we mitigate the noise inherent in path-traced images? The de-noising algorithm must also be fast enough for real-time use. Additionally, the de-noising shall introduce as few artefacts to the image as possible.

1.3 Limitations

The renderer is not intended for high-end image productions, but rather to provide real-time previews of production-quality images. Since the primary use case of the renderer is to visualise small sets of products we only consider small scenes. For example, a few pieces of furniture on a floor. Furthermore, we do not consider any form of animated geometry. However, we do consider the functionality to move and rotate geometry, e.g. rearranging the furniture on the floor.

Additionally, we do not consider non-ray-tracing algorithms for rendering global illumination. For example, techniques used in modern game engines such as light probes, light maps, and light propagation volumes.

[2]https://www.chaosgroup.com/

2
Previous Work

In this chapter, we introduce previous work in the field of ray tracing. We give a background to data structures and algorithms used to speed up the performance and improve the visual quality of ray tracing programs. First, we give an account of the origins of ray tracing and path tracing. We then explain various data structures used to accelerate ray tracing, namely acceleration data structures. Finally, we introduce previous work on algorithms used to remove noise from path-traced images.

2.1 Ray tracing

The first example of *ray tracing* in computer graphics came from Whitted & Turner [1] in 1979 and can be described as an attempt to create physically accurate images. They traced rays originating from a virtual camera onto a scene, and from it to light sources, to calculate the lighting. Whitted's model for ray tracing was, however, limited in several aspects. It produced perfect sharpness in both *shadows*, *reflected* and *refracted light*. In reality, the blurriness of reflections, refraction and shadows are determined by a number of parameters such as light source shape and the material properties of the scene geometry. In 1984, Cook et al. [2] achieved realistic reflections, refraction and shadows by spreading the direction of the traced rays according to suitable distributions instead of shooting them in the same direction. For example, *shadow rays* would be cast across the surface of light sources to achieve soft shadows with penumbra (half-shade).

An extended version of ray tracing called *path tracing* was introduced in 1986 by James Kajiya as a method of solving the simultaneously introduced *rendering equation* [3]. A path is a series of connected rays. Thus, a path is traced by tracing a ray from the camera and letting it reflect off every surface it hits. The rendering equation models how light hitting a surface is scattered. Path tracing solves the rendering equation, which rarely has an analytic solution, using Monte Carlo integration. Monte Carlo integration is a method for solving complicated integrals by stochastic sampling.

2.1.1 GPU Ray Tracing

Though computationally costly, path tracing is a very parallelisable task which makes it well suited for the GPU (Graphics Processing Unit). GPU path tracing has, therefore, been a widely researched topic since the advent of programmable GPU pipelines around the turn of the millennium. Purcell et al. [4] conducted an early study of how path tracing could be implemented on a programmable GPU. Also, Aila and Laine studied the efficiency of some GPU acceleration structure traversal- and ray-triangle intersection methods [5, 6]. Laine et al. [7] later expanded upon their work and proposed a path tracer structure and scheduling approach suitable for modern GPUs.

Because path tracing is generally too slow for real-time applications, it has mostly been used for offline rendering. However, with the increasing computation capabilities of GPUs, real-time path tracing has become increasingly viable. One notable result is the Brigade engine [8]. A more recent advancement is the introduction of real-time ray tracing techniques in game engines[1].

2.2 Acceleration Data Structures

Reaching interactive frame rates requires tracing a large number of rays against the scene geometry every frame. A typical scene can contain many thousands of triangles. It is infeasible to test each ray for intersection against all triangles in the scene exhaustively. Instead, *Acceleration Data Structures* (ADS) are used to reduce the number of ray-triangle intersection tests needed. The acceleration data structures generally work by spatially sorting the triangles in some way, allowing the tracing algorithm to discard triangles outside the vicinity of a ray. Acceleration structures should allow for fast traversal and discard as many triangles as possible. In ray tracing, common acceleration structures include bounding volume hierarchies, space-partitioning trees, and grids.

One commonly used acceleration structure is the *kd-tree* [9]. It was long considered the most efficient ADS for fast ray tracing [10, 11]. However, more recently the popularity has shifted in favour of Bounding Volume Hierarchies (BVH) [12, 13]. Some of the benefits of using a BVH compared to a kd-tree is that the BVH generally has a lower memory footprint and is simpler to construct [12, 13]. Vinkler et al. [14] showed that BVHs consistently outperform kd-trees when tracing simple to moderately complex scenes on modern GPUs.

Wald et al. [11] provide a BVH construction algorithm that uses the Surface Area Heuristic (SAH) introduced by Macdonald and Booth [15]. However, Karras et al. introduced an alternative construction algorithm more suitable for GPUs [13]. Additionally, Stich et al. [12] introduced the Split Bounding Volume Hierarchy (SBVH), which improves the tracing performance by reducing the spatial overlap between sibling nodes in the hierarchy. For large scenes containing many objects, it is common to use two-level BVHs, where a large number of BVHs, usually one per object, are sorted into a top-level BVH. This approach is used in the ray-tracing frameworks Embree [16] and Optix [17]. An algorithm for optimising two-level BVHs was introduced by Benthin et al. [18].

A conventional and efficient approach for traversing a BVH is to use a stack, as when doing a depth-first traversal of a tree. However, this approach introduces

[1]Unreal Engine: `https://www.unrealengine.com/en-US/blog/epic-games-demonstrates-real-time-ray-tracing-in-unreal-engine-4-with-ilmxlab-and-nvidia`
EA Seed: `https://www.ea.com/seed/news/seed-project-picapica`

an undesirable memory footprint. Several methods for stackless BVH traversal were developed to remove the memory footprint of the stack. One such approach is to use skip pointers as described by Smits [19]. Torres et al. developed a similar method for GPU ray tracing [20]. Binder and Keller [21] introduced a traversal method using a hash table to backtrack during hierarchy traversal efficiently.

Wide Bounding Volume hierarchies is a type of BVH in which the branching factor is increased to provide better traversal performance on SIMD architectures [22, 10]. A recent extension to this idea is the Compressed Wide BVH introduced by Ylitie et al. [23]. They improved both the traversal performance and memory footprint by compressing both the BVH and the traversal stack.

Another commonly used acceleration structure is the three-dimensional grid. It was introduced by Fujimoto et al.[24] in 1986 by another name: Spatially Enumerated Auxiliary Data Structure (SEADS). A common drawback of grids is that they partition the scene into uniform cells regardless of how the geometry is distributed. This issue is called the *teapot in a stadium* problem. Consider a scene containing a stadium with a teapot placed in the middle of the field. If the scene is sorted into a grid, the teapot is likely contained within a single grid cell, while the surrounding cells are empty. The uneven distribution of geometry makes the cell containing the teapot expensive to traverse since a ray would have to be exhaustively tested against all triangles in the teapot.

Jevans and Wyvill [25] introduced an adaptive grid which performed better on scenes with uneven geometry distributions. It works by recursively subdividing the scene into a grid such that each cell can contain another grid. A similar approach was used by Kalojanov et al. [26] who introduced a two-level grid optimised for modern GPUs. The two-level grid was further extended by Pérard-Gayot et al [27] into what they call an *irregular grid.* The irregular grid is constructed from a two-level grid. From the two-level grid, adjacent cells are merged using the surface area heuristic. The merging process results in a grid where the cells are irregular in size, i.e. smaller in high-density regions and vice versa.

2.3 Noise reduction

Path tracing is most widely used for offline applications. Therefore, most noise reduction techniques are designed specifically for offline use. Offline de-noising usually uses more than one sample per pixel with execution times well above the latency limits of interactive graphics. Zwicker et al. [28] divide de-noising techniques into two categories based on the method by which they adapt the sampling rates and reconstruction filters. Firstly, techniques which analyse the light transport equations and, secondly, techniques using statistics based on local image samples. An example from the first category is Durand et al.[29]. They locally approximate 2D image bandwidths to guide a cross-bilateral filter. The second category includes, for example, image-space filtering such as in Rouselle et al.'s paper [30]. There is also a real-time approach using a multi-scale edge-avoiding wavelet filter by Dammertz et al. which can reconstruct a 720p image in 10ms. [31].

Other ways to reduce noise involve machine learning. Chaitanya et al. [32] used a recurrent auto-encoder to filter path-traced images with one sample per pixel (spp).

They achieve a reconstructed 720p image in 55ms. Moreover, Bako et al. [33] trained convolutional neural networks to filter the noise in Pixar's animated movies. Another machine learning technique, by Dahm and Keller [34], uses reinforcement learning to learn in what direction samples are more likely to have higher contributions. As the reinforcement-learned model improves, it is more likely to choose high-importance samples resulting in faster convergence.

The past year, two papers have arisen that focus on improving real-time path tracing; *Spatiotemporal variance-guided filtering* by Schied et al.[35] and *An efficient denoising algorithm for global illumination* by Mara et al. [36]. Both methods can produce smooth results from input images with just one sample per pixel. Schied et al. produce stable 1080p images in 10ms, while Mara et al. achieve similar results in 10ms with 720p images. As a base, Schied et al. use Dammertz et al.'s [31] À-Trous wavelet filter, while Mara et al. [36] use a cross-bilateral filter by Eisemann and Durand [37]. Schied et al. and Mara et al. achieve a higher similarity to the ground truth with their filters by splitting the lighting components when filtering. For example, filtering a reflection as harshly as a diffuse light easily over-blurs the reflection. Mara et al. filter the diffuse and glossy light separately to achieve better reflections. Schied et al. separate the indirect and direct light and filter direct light less harshly, preserving sharp shadows. Schied et al. and Mara et al. also use similar techniques to stabilise the image during motion. Both re-use previous frames that are re-projected to the current frame using movement vectors. Apart from stabilising the image, re-using previous frames yields a couple of extra samples per pixel as well as a potential boost in performance. Additionally, both algorithms use temporal filtering based on the temporal anti-aliasing by Karis et al. [38].

3
Theory

This chapter delves into the details required to understand this report. First, we explain the rendering equation and how path tracing is used to solve it. Next, we introduce the acceleration data structures and noise reduction algorithms used in this project. Finally, we explain the architecture of a modern GPU and the programming considerations it induces.

3.1 Path Tracing

Path tracing, introduced by Kajiya [3], is a rendering method where ray tracing is used to solve the rendering equation. The rendering equation, shown in equation 3.1 and visualised in Figure 3.1, is derived from that light interacting with a surface is energy conserving. More specifically, it states that the radiance, i.e. light intensity, from a point p on a surface towards a direction w_o, $L_o(p, w_o)$, is equal to the emitted radiance $L_e(p, w_o)$ plus the reflected radiance. The reflected radiance is an integral over the hemisphere above the point p, where the integrand, $f(p, w_o, w_i)L_i(p, w_i)(w_i \cdot n)$, consists of three terms. The first term, $f(p, w_o, w_i)$, is the Bidirectional Reflectance Distribution Function (BRDF), which describes what proportion of the irradiance coming from w_i is reflected towards w_o. The second term, $L_i(p, w_i)$, is the irradiance from the direction w_i. The final term is the clamped cosine term $(w_i \cdot n)$, which weighs the reflected radiance by the cosine of the angle between the surface normal and the incoming light direction.

$$L_o(p, w_o) = L_e(p, w_o) + \int_\Omega f(p, w_o, w_i)L_i(p, w_i)(w_i \cdot n)dw_i \tag{3.1}$$

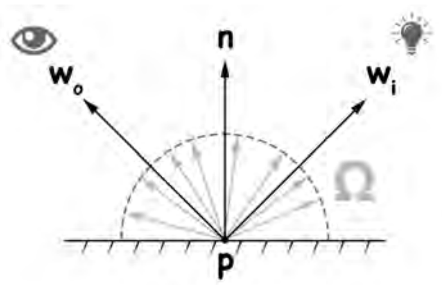


Figure 3.1: Figure showing the terms in the rendering equation. The light reflected toward w_o at a point p is found by integration over all directions w_i in the hemisphere Ω.

An image can be rendered by solving the rendering equation for all visible points in a scene, i.e. finding the radiance of all visible points towards the camera. However, the integral in the rendering equation rarely has a closed form. Therefore, the integral cannot be solved analytically except in some trivial cases. What Kayiya proposed was to solve the integral by Monte Carlo integration:

$$L_o(p, w_o) \approx L_e(p, w_o) + \frac{1}{N}\sum_{i=1}^{N} \frac{f(p, w_o, w_i)L_i(p, w_i)(w_i \cdot n)}{p(w_i)} \tag{3.2}$$

where $p(w_i)$ is the probability density function (PDF), i.e. the probability density for sampling the direction w_i. To sample the incoming light $L_i(p, w_i)$, a ray with origin p and direction w_i is traced against the scene. If the ray hits a surface, the rendering equation is sampled at that position to estimate the amount of light reflected towards the first position. Path tracing is thus a recursive process in the sense that the reflected radiance at a point depends on the irradiance at the point. Which in turn is dependent on the reflected radiance at other surfaces in the scene.

In practice, the colour of each pixel is integrated by tracing *paths* towards the scene. A path is sampled by tracing a ray from a virtual camera against the scene, letting it reflect (or refract) off every surface it hits, and sampling the rendering equation once at every surface interaction. This process is visualised in Figure 3.2. A path is terminated when it hits a light source or does not hit any geometry. However, the likelihood of a ray hitting a light source can be quite small. For example, consider a light bulb inside a room; The light bulb only covers a small fraction of the space. Luckily, the rendering equation can be split into two separate integrals: one for direct lighting and one for indirect lighting. Direct lighting at a point is the light coming directly from a light source, while indirect lighting is the light reflected from other surfaces. The direct lighting at a point is also integrated using Monte Carlo integration. Sampling the direct lighting at a surface involves testing whether the light emitted from a light source reaches the surface. It is done by tracing a *shadow ray* from the surface toward a randomly sampled point on the surface of the light source. If the shadow ray is unobstructed, the light contributes to the direct lighting at the surface. However, if the shadow ray intersects any geometry between the surface and the light, the surface lies in shadow. The contribution of each sample is weighted by the area of the projection of the light source on the hemisphere above the surface, i.e. how large the light source appears from the surface's point of view. Weighing the samples this way ensures that larger and closer light sources yield correspondingly higher contributions to the direct lighting.

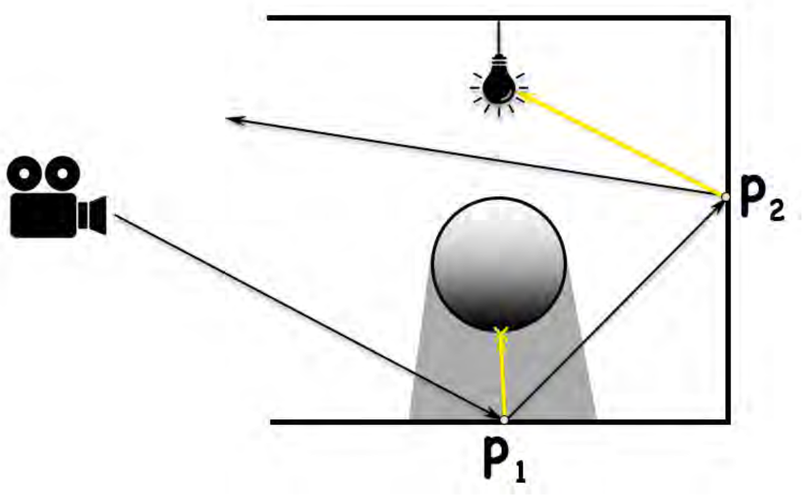


Figure 3.2: 2D visualisation of how a path is traced against a scene. The path starts at the camera and is reflected at points p_1 and p_2. At each surface interaction, a shadow ray is traced towards the light source. The first point lies in shadow, since its shadow ray intersects the sphere before reaching the light source.

Only terminating paths when they hit a light source or leave the scene is quite impractical since the paths can be unbounded in length. One solution is to define an upper bound for the path length. When a path reaches the upper bound, it is terminated, discarding the contributions of subsequent bounces. However, defining such a bound makes the result biased, i.e. darker than the correct result. The bias can be removed by terminating the path using Russian Roulette [39, p. 788]. Russian roulette works by randomly terminating the path with a probability depending on the contribution of the path, such that long paths with low contribution are less likely to be continued. This method avoids introducing bias when terminating a path by increasing the contribution of unterminated paths by a factor of the inverse termination probability.

Note that Russian Roulette will not completely remove the issue of unbounded paths, since the termination is random, though it mitigates the issue. Also worth noting is that the contribution of samples along a path decreases with the length of the path, due to the conservation of energy. The bias introduced by terminating at a fixed depth thus decreases as the depth increases. With a large enough maximum depth, the bias is imperceptible. In many cases, five to ten bounces of light are quite enough.

3.2 Acceleration Data Structures

As mentioned in Chapter 2, acceleration data structures (ADS) are used to speed up ray tracing applications by reducing the number of ray-primitive intersection tests. This section covers the ADS variants explored in this report, namely bounding volume hierarchies and grids.

3.2.1 Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVH) are, as explained in Section 2.2, one of the most widely used and efficient acceleration structures in ray tracing applications. As described by Rubin and Whitted [40], a BVH subdivides a scene into a hierarchy of *bounding volumes* such that each node encloses its descendants. The type of volume used is arbitrary, though in practice it is common to use simple geometric shapes that are fast to test for intersection. For example, axis-aligned or oriented bounding boxes (AABB, OBB), i.e. rectangular blocks, are commonly used. A visual explanation of how a BVH can subdivide a scene is found in Figure 3.3.

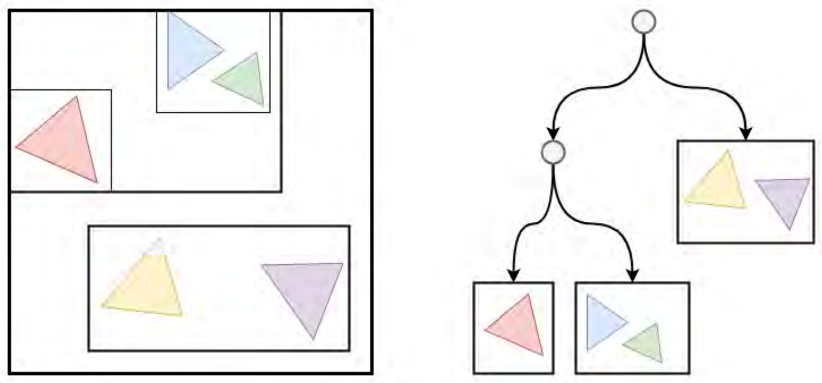


Figure 3.3: 2D example of a BVH. One the left is a visualisation of how a scene can be subdivided into a hierarchy of axis-aligned bounding boxes. The tree structure of the same hierarchy is visualised on the right.

Constructing an optimal BVH is thought to be an NP-hard problem [41]. It is therefore common to use greedy algorithms with heuristics to find *good enough* BVHs. One such algorithm was introduced by Wald et al. [11], where a BVH of AABBs is greedily constructed from the top down using the *Surface Area Heuristic* (SAH). The SAH, introduced by Macdonald and Booth [15], models the cost of traversing a BVH node. The SAH, shown in Equation 3.3, defines the traversal cost of a node split into two subsets A and B as the cost of traversing the parent a node plus cost of traversing A and B. The cost of traversing a subset X is defined as the probability of hitting the subset $P(X)$ times the number of primitives in the subset $|X|$, times the cost of the intersection test with a primitive C_i.

$$C(A,B) = C_T + P(A)|A|C_i + P(B)|B|C_i \tag{3.3}$$

The probabilities of hitting the two subsets, shown in Equation 3.4, are found based on the *surface area* of the two subsets, i.e. the summed area of the triangles in the set.

$$P(A) = \frac{Area(A)}{Area(A \bigcup B)} \qquad P(B) = \frac{Area(B)}{Area(A \bigcup B)} \tag{3.4}$$

Wald's construction algorithm [11] works by splitting the scene into two subsets using the SAH, and then recursively subdividing the children. A node is split by partitioning the primitives in the node such that the traversal cost according to the SAH is minimised. The optimal partition (according to the SAH) is found by evaluating different partitions. However, it is quite unnecessary to test all possible partitions, partly because it is costly, and partly because it is beneficial that the two subsets overlap as little as possible. Instead, the possible partitions are found by sorting the triangles along each major axis (i.e. x, y, z). For each such axis, the triangles are sorted based on their position along the axis. A set of candidate partitions are then generated by splitting the triangles at each triangle along the axis. That is, for each triangle, the triangle set is split into two sets; one containing all triangles before the current triangle and one containing the remaining triangles. The partition yielding the lowest cost is then used to split the node. The recursive process stops when it is not beneficial to split a node, which generally occurs when the number of triangles in the node is small.

3.2.2 Bounding Volume Hierarchy Traversal

Testing if a ray intersects the geometry in a BVH is done by traversing the BVH. The goal of the traversal is to visit the leaf nodes intersected by the ray as cheaply as possible. A common approach is to do a depth-first traversal using a stack. Traversing a BVH using a stack starts at the root node and works as follows: a node is visited if the ray intersects the bounds of the node. Otherwise, the node and its descendants are skipped. Visiting an internal node and a leaf node is different. If the node is internal, the traversal continues to one of the children while the index of the other child is pushed on the stack. If the node is a leaf, the ray is tested against all the primitives, i.e. triangles, within the leaf. After traversing a leaf, the traversal continues to the node on the top of the stack. The traversal ends whenever the stack is empty.

3.2.3 Grids

A grid is constructed by creating a uniformly sized 3D grid spanning the entire scene and inserting each primitive into the grid cells it overlaps. To know where any cell starts a resolution needs to be defined for the grid. The resolution decides how large each grid cell is. With small grid cells, more cells are discarded when tracing, but the amount of duplicated triangles in every cell increases. That is, a ray may intersect a triangle in one cell while the point of intersection on it is in a cell farther away. This exemplifies how intersection tests are wasted which can quickly lead to a bottleneck for high grid resolutions. If the resolution is too small, the outcome is akin to using no acceleration structure at all. Namely, a large number of primitives are inside just one cell and are traced exhaustively. A common heuristic for deciding the grid resolution was introduced by Cleary et al.[42] in 1983:

$$N_i = d_i \sqrt[3]{\frac{\lambda N_i}{V}} \tag{3.5}$$

The resolution N from Equation 3.5 is then calculated for every dimension i. d_i is the grid bounds for dimension i, and V is the volume of the scene bounds. λ is a user-adjusted parameter whose optimum can vary scene by scene. However, it is usually pre-calculated for a satisfactory output and allowed to remain constant.

3.2.4 Grid traversal

When it comes to traversing a regular three-dimensional grid, a conventional method is the 3D-Digital Differential Analyser (3D-DDA) algorithm proposed by Fujimoto et al. [24]. The basic idea rests on that there is a constant distance between cell walls for every dimension. So, by knowing which ray dimension's direction is closest to the next cell intersection, the next cell along the ray is easily found. Thus every cell along the ray visited, regardless of whether the cells contain geometry or not. However, visiting an empty cell is wasteful since there are no possible intersections within the cell.

Early exiting of the grid is a feature which allows for more efficient grid traversal. It means that the ray does not necessarily traverse *all* grid cells in the ray direction. Instead, if the ray's *point of intersection* lies within the bounds of the current cell, the traversal can be terminated.

Another approach for reducing the number of traversal steps is to skip empty cells [43]. By storing the distance to the closest non-empty cell inside the empty ones the ray can traverse empty regions in fewer steps. Thus, when it finds an empty cell, it instead traverses the distance stored in the cell along the ray. For example, if the distance from the current cell to the closest surface is five grid cells, the ray can skip the closest four cells.

3.2.5 Irregular Grids

A significant limitation of grids is, as mentioned in Section 3.2.3, that the acceleration structure does not adapt to the geometry distribution of the scene. One way to alleviate this problem is to use an adaptive grid, which adapts to the geometry distribution by sorting the contents of each grid cell into its own grid [25]. A recent extension to adaptive grids is the *Irregular Grid* proposed by Pérard-Gayot et al. [27]. The main idea behind the irregular grid is that the grid can adapt better to the scene geometry if irregularly sized grid cells are allowed. It works by merging the cells of an adaptive grid where suitable, such that there are no redundant grid cells in low-density regions of the scene. A visualisation of how the irregular grid differs from a uniform and two-level grid is shown in Figure 3.4.

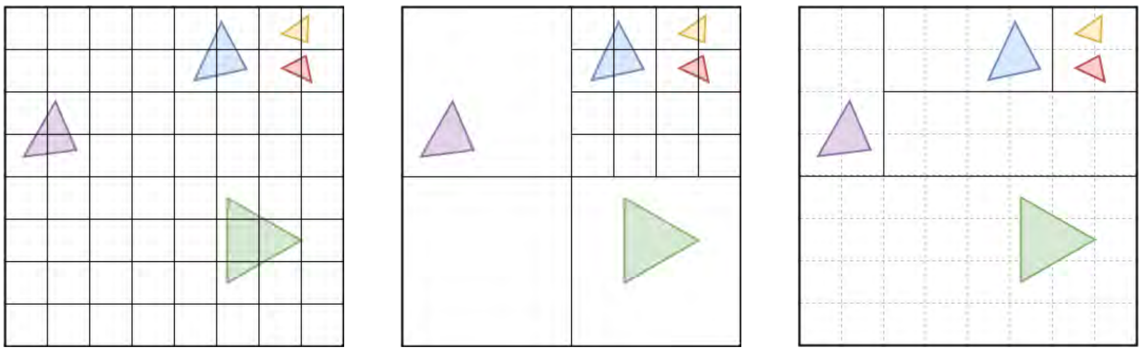

(a) Uniform Grid (b) Two-level Grid (c) Irregular Grid

Figure 3.4: 2D visualisations showing the difference between a uniform, two-level, and irregular grid. Note how the cells in the irregular grid align to an underlying uniform grid, shown as dashed lines.

In a uniform grid, the cell at a certain position can be found by calculating the cell index from the position. It works by dividing the position relative to the grid origin by the cell size and *flooring* the result to the closest integer. In an irregular grid, finding a cell at a position is not as straightforward due to the irregularity of the cells. However, although the cells are irregular, their dimensions are always multiples of the smallest cells in the grid they were merged from. Therefore, all irregular cell boundaries lie on boundaries of an underlying *virtual* grid (see Figure 3.4c). By creating a uniform grid with the resolution of the underlying virtual grid which stores the index of the overlapping irregular cell, the irregular cells can be indexed using a position.

In addition to merging the cells, one of the key features of the irregular grid is that the cells can be expanded such that they encompass neighbouring cells, allowing the neighbours to be skipped during traversal. The cell expansion builds on the observation that traversing a cell is redundant if all triangles within the cell already have been tested. For example, when traversing two neighbouring cells containing the same set of triangles, only one cell has to be visited. Visiting both cells would result in testing the ray against each triangle twice. Therefore, the bounds of a cell can be expanded to encompass neighbouring cells which only contain a subset of the cell triangles.

The construction of an irregular grid can be summarised as the following three-step process:

Irregular Grid construction overview:
1. Construct a two-level grid.
2. Merge cells according to the surface area heuristic.
3. When possible, expand cell bounds such that neighbours with a subset of the cell triangles are enclosed.

In the construction step, a top-level grid is constructed using the grid construction algorithm described in Section 3.2.3. After constructing the top level, each cell is itself sorted into an acceleration structure. Pérard-Gayot et al. [27] uses an *octree* instead of a grid for the second level. The octree enforces the resolution to be equal in all directions and a power of two. However, the same result can be achieved with a grid as long as the resolution is restricted to be a power of two. The resolutions of the two levels are determined using the grid heuristic, tuned by two parameters λ_1 and λ_2 [27].

In the merge step, the cells are merged along the Cartesian axes in a round robin fashion. That is, for each pass, the cells are first merged along the x-axis, then the y-axis, and finally the z-axis. When merging along each axis, all cells are checked to see if it is possible and beneficial to merge with any of the neighbours along the axis. A merge is possible when the two cells form a box, i.e. the cell extents along the two other axes are equal. Whether or not a merge is beneficial is determined using the Surface Area Heuristic, described in Section 3.2.1. Cells that should be merged form merge chains, i.e. lists of consecutive cells that should be merged. The cells in each chain are then pairwise merged until only one cell remain, or no merges are beneficial. Merging a chain of length n may therefore require $log_2(n)$ iterations. When merging two cells a and b, the bounds of a are expanded to encompass the bounds of b, and the triangles in b are added to a. When the merging is complete, b is removed from the grid and replaced with a.

The final construction step is the cell expansion. As in the merge step, the cells are visited in multiple passes. In each pass, the cells are expanded along each axis, i.e. first all cells are expanded along the x-axis, then the y-axis, and finally the z-axis. A cell can be expanded in a direction along an axis if all neighbours in that direction contain a subset of the triangles in the cell. If that is the case, the cell is expanded as much as possible in that direction. For example, if there are multiple neighbours in a direction, the cell can only be expanded to the closest *far* side of those neighbours. The expansion process may continue until no cell expansions are possible. However, Pérard-Gayot et al. [27] found that there was little to be gained by running more than three expansion passes. That is, most of the performance gains come from cells expanded in the first few passes.

The irregular grid is not traversed with the standard 3D-DDA algorithm due to the irregularity of the grid cells. Instead, the length of each step along the ray is determined by the extents of the grid cell. The traversal thus works as follows:

Irregular Grid Traversal:

1. Start by setting the current ray position to the first point within the grid along the ray. If no such point exists, exit.
2. Find the current cell by looking in the virtual grid using the current position on the ray. If the current position is outside the grid, exit the traversal.
3. Trace the ray against the primitives in the cell. If an intersection is found within the cell bounds, return the closest such intersection.
4. Update the current position to be the point where the ray exits the current cell. Then go to step 2.

3.3 Noise reduction

Solving the rendering equation using path tracing does eventually result in the image converging to the expected value of the equation. However, convergence often requires several thousand paths per pixel. It can take several minutes if not hours of rendering time depending on the efficiency of the path tracer, the hardware it runs on, and the complexity of the scene. Because of this, path-traced images inherently suffer from pixel variance, i.e. noise, at lower sample counts. Consider sampling the radiance from two adjacent points on a surface with one randomly scattered path each. It is likely that the radiance at the two points is different since the two paths are likely to scatter in different directions. However, if many paths per surface point are sampled the means of those samples will be more alike, resulting in less noise.

The image noise is especially troublesome in real-time applications due to latency limits. For example, suppose that we are targeting a frame rate of 25 Hz and can trace 100 million rays per second (MRay/s). This amount of rays per second is roughly the order of magnitude which can be expected on a modern GPU [27, 6]. The performance constraints yield 4 million rays per frame. We are thus limited to four rays per frame and pixel, barely enough to sample a single path when rendering a one-megapixel image.

The noise can be reduced by increasing the number of samples per pixel (spp). However, assuming that the frame rate is fixed the ray tracing performance would have to increase by several orders of magnitude to render noise-free images. Another approach is to apply noise-reduction filters to the image. Filtering the image is more time-efficient than tracing additional paths, but introduces bias and artefacts to the image. However, noise-reduction techniques are required both to reach interactive frame rates and to produce noise-free images.

3.3.1 Real-time De-noising

Real-time noise filtering refers to approaches that optimise the filtering speed over quality to reach interactive frame rates. Currently, that means the sample budget is only one sample per pixel. However, Mara et al. [36] and Schied et al. [35] have made advances to bring the quality of path tracing to real-time by combining $1spp$ filtering methods, ie. methods that filter a 1spp path-traced image, with some fine-tuning techniques. In particular, Schied et al. uses Dammertz et al.'s [31] À-Trous edge-stopping wavelet filter. A wavelet is a scaling function on some signal, here on the pixel colours in a noisy image, which attenuates frequencies outside its range [44, p.102]. Dammertz's filter kernel is based on a $B3$ spline interpolation as such:

$$\begin{pmatrix} \frac{1}{256} & \frac{1}{64} & \frac{3}{128} & \frac{1}{64} & \frac{1}{256} \\ \frac{1}{64} & \frac{1}{16} & \frac{3}{32} & \frac{1}{16} & \frac{1}{64} \\ \frac{3}{128} & \frac{3}{32} & \frac{9}{64} & \frac{3}{32} & \frac{3}{128} \\ \frac{1}{64} & \frac{1}{16} & \frac{3}{32} & \frac{1}{16} & \frac{1}{64} \\ \frac{1}{256} & \frac{1}{64} & \frac{3}{128} & \frac{1}{64} & \frac{1}{256} \end{pmatrix}$$

Figure 3.5: A $5x5$ $B3$ spline kernel showing the contributions around a central point (with contribution $\frac{9}{64}$). It is the $2D$ equivalent of this $1D$ kernel: $(\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16})$

Dammertz et al. combine discrete convolutions of wavelet transforms of the image, meaning versions of the image filtered using increasing *bandwidths*. Á-Trous refers to the filter containing 'holes', meaning that for increasing filter sizes it continues to filter the same amount of pixels. In practice it is a way get around having to use a large amount of sample points to increase the filter size. Instead, the filter size grows in each convolution with the same amount of sample points. For example, a $3x3$ pixel kernel, i.e. a filter that operates on 9 pixels, would first filter out a $3x3$ pixel area. However, as the bandwidth grows in each wavelet convolution, for example into a $5x5$ pixel area, only 9 of those pixels would contribute to the filter. See an example of this in Figure 3.6.

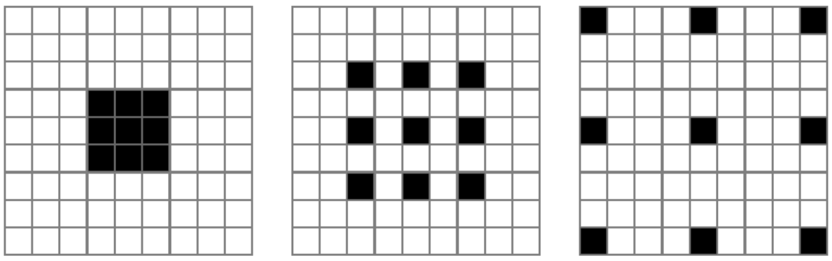


Figure 3.6: À-Trous filtering: increasing the pixel sampling distance between filter iterations. The three images show a $9x9$ pixel grid. The black squares indicate filter samples. In the leftmost image, the sampling distance is one, yielding a $3x3$ pixel sampling block. In the middle, the sampling distance is two, and in the rightmost it is four.

Edge-stopping refers to the exclusion of colour samples that deviate too far from the middle pixel value in a sampling block. In practice, one lowers the weight, i.e. the contribution, of such samples. To achieve edge detection, Dammertz uses a so-called geometry buffer, i.e. a buffer filled with information about the scene geometry. For each pixel, it stores geometry positions, normals, and albedo. The albedo is the colour information of the scene geometry visible from a pixel. For example, Dammertz et al. find edges using the world-space position of sampled pixels [31]. World-space refers to the relative position in the scene rather than, for example, the position on the screen. A pixel sample in the filter has a small contribution if its geometric position is far away from the central pixel. Similarly, it contributes little if its albedo is not the same as, or its normal direction is not similar to, the central pixel. Diminishing the contribution, or weight, of a sample, works as edge-detection meaning it helps retaining image details.

Finally, it is possible to filter light components with different harshness by splitting the light components and filtering them separately. Mara et al. and Schied et al. both retain certain image details by splitting their lighting components. For example, filtering glossy reflections less harshly than diffuse reflections results in better retention of glossy reflections, which otherwise might be over-blurred.

3.4 GPU Architecture & Programming

Modern GPUs are single instruction multiple threads (SIMT) machines [7]. SIMT means that a group of threads, often called a warp or a wavefront, is executed in parallel on a GPU core using a single instruction stream. Each core on a GPU is thus a single instruction multiple data (SIMD) processor. What this means in practice is that a single GPU core has a large set of arithmetic logic units (ALU) that execute the same instructions on different pieces of data.

The SIMT architecture is very suitable for highly parallel code but is worse at handling control flow divergence. When the threads in a warp execute different branches, for example in an *if-then-else* statement, the different branches cannot be executed in parallel since the threads would need to execute different instructions. This problem is solved by masking out threads that should not execute the current branch. For example, if the threads within a warp take two different branches, then one of the branches is executed first with some threads masked out. When the first branch is finished, the second branch is executed with the other threads masked out. Control flow divergence is therefore detrimental to performance since it decreases the utilisation of the GPU [7, 5].

Another important factor for GPU performance is that the memory is *high-throughput high-latency* [7]. This trade-off between throughput and latency results in threads being stalled for relatively long periods of time when fetching memory. To hide this latency, modern GPUs process multiple warps concurrently. While one warp is stalled waiting for memory, another warp can be executed. The limiting factor for this dynamic scheduling is the memory and register usage. When the state of each warp increases in size, e.g. the number of used registers, fewer warps can fit in memory which effectively limits the number of concurrently active warps [7].

The GPU architecture imposes some performance considerations specific to the context of ray tracing. One crucial factor is the coherence of the rays within a warp. If the rays are incoherent, i.e. have significant variations in origin and direction, they will have different traversal patterns over the accelerations structures. Different traversal patterns could lead to problems when, for example, some rays finish the traversal early, i.e. using early exits, or when two rays within a warp visit ADS leaves with a different number of primitives. Incoherent rays also lead to incoherent memory accesses, for example when fetching ADS nodes or material properties.

Another issue occurs when the paths traced by a warp of threads terminate at different depths, which is detrimental to the performance since the threads that traced the terminated paths will be idle until all the paths within the warp are terminated.

There are multiple approaches for adapting ray tracing applications to fit the GPU architecture better. Aila and Laine [5] reduce the issue of varying execution times between threads by using persistent threads. It works by only spawning as many threads as the hardware can run concurrently, and scheduling the work distribution manually. The scheduling works by letting each thread fetch jobs from a global work pool, using atomic operations to avoid race conditions. Another approach is *wavefront path tracing*, introduced by Laine et al.[7]. The idea is to keep a large pool of paths ($\approx 2^{20}$) in the device memory of the GPU. All paths are then traced one step at a time. The path tracer first traces the first ray of *all* paths, then evaluate the ray scattering and illumination at the intersection point for *all* paths. All paths are thus kept in sync, rather than tracing the paths to completion in batches. Structuring the program this way allows the path tracer to be divided into a small set of specialised programs, i.e. one program for the path tracing logic, one program for tracing rays, and one program for evaluating materials. This division ensures that all concurrently running threads will run the same code and have somewhat more coherent memory accesses. For example, the program responsible for tracing rays will only access the ADS and triangle data, while the program evaluating materials will only access material data. Furthermore, the smaller size of the specialised programs is helpful since GPUs generally have smaller instruction caches than CPUs [7]. Lastly, tracing the pool of paths one step at a time allows the system to regenerate terminated paths to keep the pool fully occupied. That is, when a significant number of paths are terminated, they are replaced with new paths.

3.4.1 GPU Programming in WebGL

WebGL is quite limited compared to other GPU programming APIs such as CUDA. WebGL imposes stricter constraints both on how a program can be structured as well as how the input and output data may be structured and encoded. This section explains the most important of these constraints.

A WebGL shader program consists of two programmable stages; The vertex shader and the fragment shader. The shader program runs when some geometry, for example, a triangle mesh, is drawn. The vertex shader transforms the triangle vertices to their screen positions. After the geometry is transformed, it is rasterised. Rasterisation is the process of turning an input shape, such as a triangle, into the set of pixels covered by the shape, called fragments. Each fragment is processed by the fragment shader, which calculates the colour of the fragment. All of the fragments are then merged into the output image of the program.

The shaders are quite limited in terms of reading and writing data. The vertex shader can process the input geometry, the fragment shader can output pixel colours, and both stages can read data from uniform variables and textures. Each shader thread may read and write data to memory local to that thread, but it is not possible to pass data between threads or retain data across multiple executions of the same shader. The output of a shader may be written to a texture to use the result in another shader. However, a shader may not read and write to the same texture.

Uniform variables are generic pieces of data, for example floating point numbers, vectors, and matrices. Each shader program has a limited number of uniform variables. This limit varies but is at least 1024.[1] Since the number of uniform variables is limited, the variables are often used for smaller sets of data, such as material and lighting parameters.

Textures are images in one, two, or three dimensions. In addition to its data, a texture has a resolution and a format. The resolution defines the number of pixels and how they are arranged, while the format describes how data is stored in each pixel. The format specifies both the number of components per pixel and how each component is stored. A monochrome texture only needs one component per pixel, while a colour texture usually has three (red, green, and blue). The data is stored as integer or floating point numbers of various sizes. For example, the format *RGB32F* means three 32-bit floating point numbers per pixel, while *R8I* means a single signed 8-bit integer per pixel.

A shader program can read data from a texture in two different ways. The first way of reading from a texture is to sample the texture using normalised texture coordinate, i.e. a position in the range $[0, 1]$. For example, the coordinate $(0, 0)$ refers to the lower left corner of a two-dimensional image, while $(1, 1)$ refers to the upper right corner. Sampling a texture in this way applies filtering, i.e. interpolation between neighbouring pixels, according to the configuration of the texture. Possible configurations include for example not filtering at all, or linearly interpolating between pixels. The second way of reading from a texture is to fetch a single pixel using its index, i.e. as one would read a value from an array in many programming languages. Fetching a single pixel does not apply any filtering.

[1]https://www.khronos.org/opengl/wiki/Uniform_(GLSL)

4
Method

We implemented an interactive path tracing system which runs in a web browser. The path tracer was implemented using JavaScript and WebGL. For the path tracer, we implemented several acceleration structures. Additionally, we implemented multiple variants of a denoising algorithm. This chapter covers the implementation of the system, the acceleration structures, and finally the denoising algorithm.

4.1 System Overview

The path tracing system is a JavaScript application that utilises WebGL to run rendering tasks on the GPU. The process of rendering an image works as follows. Initially, the application loads a scene by fetching all the required assets, i.e. 3D-models and textures. When all the scene data is loaded, the acceleration structure is constructed. When the ADS is constructed, the shader program containing the rendering logic is compiled. The reason the shader is compiled last is that we can provide data as compile-time constants, i.e. using the `#define` macro. When everything is set up, the path tracer can start to render images. The process of rendering one frame (iteration) can be summarised as such:

Rendering process overview:

1. Handle user input.
2. Sample one path per pixel using the path tracer.
3. If denoising is enabled, apply the denoising filter on the rendered image.
4. Run a post-processing shader on the result and display it on the screen.

An image is rendered by sampling one path per pixel in the image. The pixel samples are accumulated over time to increase the image quality. The system is interactive in the sense that the user can move the camera in real time. Doing so resets the accumulation. If denoising is enabled, the result of the path tracer is filtered by the denoising system. The denoising filter uses a geometry buffer (G-Buffer) containing the position, normal vector, and albedo of the surfaces in view. Finally, post-processing effects are applied on the image, before displaying it on the screen. The process of rendering one frame is visualised in Figure 4.1.

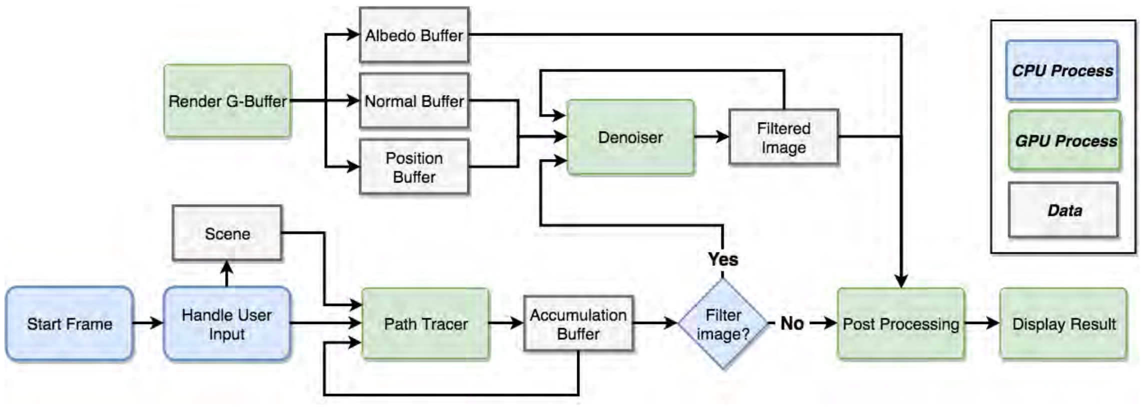


Figure 4.1: Overview of the rendering process. CPU processes are shown in blue, GPU processes in green, and data in gray.

We do not schedule the path tracing manually, for example by using persistent threads or wavefront path tracing as described in Section 3.4. The reason for this is the constraints imposed by WebGL. Shader programs in WebGL do not have write access to any global memory, removing the possibility for a global pool of tasks or paths. Each thread can read or write to its local memory, but that memory is not persistent across multiple executions of the program. The only possibilities for keeping state between two executions of a fragment shader is to read from and write to textures. It could be possible to implement some wavefront path tracing by keeping the path state in a texture. This approach would, however, introduce significant overhead. The overhead comes partly from restarting the shader and partly from reading and writing the path state. Laine et al. [7] kept 212 bytes of state per path. In WebGL, a single pixel can at most contain 16 bytes of data, and a fragment shader can write to at most one pixel per render target. Therefore, this approach would require writing to at least 14 separate render targets.

4.2 Path Tracer Implementation

To sample one path per pixel, we want to spawn one GPU thread per pixel and have each thread sample a path for its pixel. We achieve this by drawing a triangle covering the entire screen, which spawns a fragment (pixel) shader thread per pixel on the screen. The process of tracing a single path is summarised below:

Path tracing process summary (in the GPU shader program):

1. Generate a primary ray originating at the camera position. The direction is determined based on the camera orientation, focal length, and the position of the pixel. The camera is stored as a structure of uniforms.
2. Trace the ray against the scene using the acceleration structure, as described in Section 4.3. If the ray does not hit the scene, sample the scene's environment (i.e. background). Our environment is either a flat colour or a spherical environment map.
3. If the ray hit some scene geometry, trace a shadow ray originating at the current point of intersection toward a random light source. We only use spherical

light sources, defined by a position, radius, light colour, and intensity. A position on the light is sampled by sampling a random point on a disc, which is positioned at the light source and oriented towards the intersection point. If the shadow ray hits something before it reaches the sampled position, the intersected geometry counts as *in shadow*. In that case, the sampled light source does not illuminate the intersection point.

4. Generate a new ray by importance sampling the BRDF at the intersection point. Scale the contributions of subsequent samples along the path by the BRDF and PDF. If the max depth is reached, or Russian Roulette terminates the path, the process stops. Otherwise, go to step 2.

Note that it could quickly become costly to trace a shadow ray toward every light source as the number of lights increase. Instead, we randomly sample one of the light sources, and weight the contribution by the inverse probability that the light is sampled. Sampling the lights this way increases the colour variance and decreases the ray coherence when there is more than one light source. However, the increased colour variance can be counteracted to some extent by the denoising filter described in Section 4.4.

4.2.1 Materials & Shading

If the path tracer is to produce realistic visualisations, it must be able to simulate different types of materials. We chose to implement a simple physically based material model inspired by the model used at Disney [45]. *Physically based* refers to the materials following the laws of physics, such as conservation of energy, and mapping the material parameters to physical properties of the material instead of using artistic parameters mapping to the appearance of the material. The parameters used in our model are specular weight, roughness, base colour, metallic ratio, and the index of refraction (IOR). Diffuse reflections are calculated using the Burley diffuse BRDF [45]. For specular reflections, we use the Trowbridge-Reitz (GGX) BRDF [46].

The materials are provided to the shader as uniform variables. However, many of the material properties use a texture map instead of a constant value. For example, when the colour or roughness varies across the material surface. A scene with a large number of materials may, therefore, need a large number of textures. Abundant use of textures could have caused problems because, in WebGL, rendering is generally limited to 32 concurrently bound textures. We avoid this issue by baking all texture data into a single texture array allowing us to access all maps through a single texture unit. We store the layer index and texture-coordinate extents of each map in its material. Doing so allows us to find any map within the texture array. To access all colours and texture values the same way, we store constant values as single pixels in one layer of the texture array.

4.3 Ray Tracing & Acceleration Structures

Multiple acceleration structures were implemented to find the best suited for our use case. We began by creating a regular axis-aligned bounding volume hierarchy. Additionally, we implemented two other variants of a BVH. Firstly, a BVH with skip pointers, which removes the need for a stack during traversal. Secondly, an ADS where a stackless BVH is created for each object in the scene. Two grid-based acceleration structures were also implemented. The first is a uniform one-level grid. The second is an irregular grid as described by Pérard-Gayot et al. [27]. All ADS variants we implemented were constructed on the CPU (i.e. in JavaScript). The following sections explain how rays are traced using acceleration structures in our path tracer. Firstly, the structure of the tracing procedures is explained, followed by overviews of how the evaluated acceleration structures were implemented.

4.3.1 Ray Tracing Interface

The path tracer traces rays for two separate purposes. The first is to find the first surface intersection along a ray, should there be one, which is useful when tracing a path. The second is to test whether a ray intersects *any* geometry within a provided distance, which is useful to test whether a surface lies in shadow or not. These operations, let us call them *trace* and *trace shadow*, are implemented differently based on the underlying ADS. To simplify the evaluation of many different acceleration structures, each ADS is implemented to follow a common interface. Therefore, each ADS implements the same two procedures, `trace` and `trace_shadow`.

Following the same interface allows us to switch ADS at runtime. The way this works is that the source code for the path tracing shader contains all acceleration structure implementations, but only the selected implementation is compiled, i.e. by using preprocessor macros for conditional compilation. The acceleration structure can thus be replaced by recompiling the shader and uploading the new ADS data to the GPU.

All acceleration structures run intersection tests between a ray and a primitive, i.e. triangle, in their tracing procedures. That is, given a ray and a triangle index, does the ray intersect the triangle, and if so at what distance along the ray. We use the Möller-Trumbore ray-triangle intersection test [47] for these intersection tests. The triangle data is made available to the shader through a large texture, containing the triangle vertices, normal vectors, texture coordinates, and the material id.

4.3.2 Bounding Volume Hierarchy

We implemented a BVH as described in Section 3.2.1. The BVH is constructed using Wald's algorithm [11] and traversed using a stack. The only difference is that when building the BVH, we only evaluate split candidates along the *longest* axis. Our construction algorithm can thus be summarised as follows:

BVH construction summary:

1. Find AABB of triangles.

2. Sort triangles by their position along the longest axis of the AABB.
3. Evaluate the cost of splitting at each triangle index using the SAH.
4. If a split is beneficial, split at the index where the SAH is minimised and create the two child nodes. Otherwise, terminate the process.
5. Recursively continue by splitting the two child nodes.

The stack used to traverse the BVH is implemented as a constant size buffer of integer values together with an index keeping track of the top of the stack. Pushing a value on the stack works by writing the value to the buffer at the current index, and then increasing the index by one. Retrieving a value works by decreasing the index by one, and then returning the value at the new index.

We encode the BVH data to a single texture. Each node is encoded to three vectors. The first vector contains a flag stating whether or not the node is a leaf and two indices, i.e. the index of the left and right child or the range of primitives. The remaining two vectors are the minimum and maximum points of the bounding box. These three vectors are encoded as three pixels and can be accessed from the shader using the index of the node.

4.3.3 Stackless Bounding Volume Hierarchy

One potential improvement to the BVH is to remove the need for the stack when traversing the tree, reducing the memory footprint of the traversal procedure. To do this, we extend the BVH with a skip pointer tree, as described by Brian Smith [19].

To construct our stackless BVH, we first construct a regular BVH as described in the previous section. We then add a skip pointer to each node in the BVH. The skip pointers are constructed in a single depth-first traversal of the BVH. Starting at the root, we set the skip pointer to *null*. Then for each node, the skip pointers of its children are set the following way: The skip pointer of the left child is set to point at the right child, while the right child gets the same skip pointer as the parent. The result is that each skip pointer points at the node which succeeds the sub-tree of the current node in a depth-first traversal order, i.e. the node at which we want to continue if we skip all descendants of the current node. An example of a skip pointer tree and its traversal is shown in Figure 4.2.

The stackless BVH is encoded similarly to our normal BVH. The only difference is how the indices are stored. In the stackless BVH, the right child pointer is not used in internal nodes, and we can replace the right child pointer with the skip pointer. For leaf nodes, we need to use both the left and right index to define the range of primitives. Therefore, we combine the skip pointer with the flag stating whether a node is internal or a leaf. When a node is internal, the value is -1, and when it is a leaf, the value is the skip pointer. It works because the skip pointer is always greater or equal to zero.

The traversal of the skip pointer tree works as follows; When a node is visited, we test whether or not to continue down the tree by checking if the ray intersects the bounds of the node. If there is an intersection, continue to the left child of the node, otherwise, follow the skip pointer. If the node is a leaf, we test the ray against the primitives in the leaf and then follow the skip pointer. The traversal ends when we reach an invalid (null) pointer.

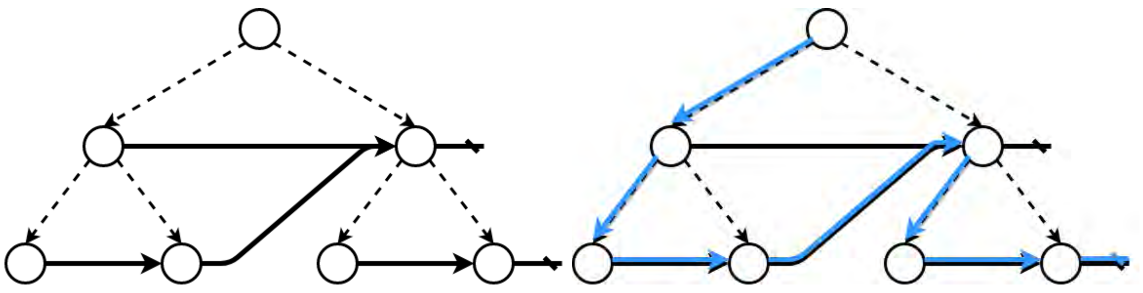


Figure 4.2: Example of our version of the Skip Pointer Tree with child pointers shown as dashed arrows and skip pointers shown as bold arrows. A stackless depth-first traversal of the tree is shown in blue.

4.3.4 Per-Object Bounding Volume Hierarchy

The third and final variant of BVH we implemented is a *per-object* BVH. The core idea is that instead of having a single BVH containing all the geometry in the scene, we have a set of BVHs; One for each object in the scene. It is, therefore, a type of two-level BVH. Two-level BVHs can simplify the construction of scenes containing many objects, and are used in several high-end ray-tracing frameworks such as Embree [16] and Optix [17]. However, we do not sort the objects into a top-level hierarchy. Instead, we store the per-object BVHs in an array, which is sufficient when the number of objects is small. Each BVH is built in object space, i.e. a coordinate system relative to the model rather than to the scene.

Using a two-level BVH may introduce significant overhead during traversal [18], though the majority of this overhead occurs when the object bounds overlap. However, there are also several benefits to having separate acceleration structures. Firstly, when an object is modified, added, or deleted, we only need to update the BVH for the affected object. Secondly, the BVHs can be pre-computed and loaded together with the objects. Furthermore, building the BVHs in object space brings two significant benefits. Firstly, it allows us to apply *affine* transformations, i.e. translation, rotation, and scaling, to the model without updating the BVH. Since we do not handle animated objects, all transformations in our system are affine. Secondly, it allows for *instancing*, which is a method for reducing memory usage when rendering several instances of the same object. It works by storing a separate transformation for each instance, but share the geometry and ADS between the instances. We do not, however, utilise instancing in our system.

Building the per-object BVH is quite straight-forward since we can reuse the construction procedure from the stackless BVH. Thus, for each object, we construct a stackless BVH. The nodes of these BVHs are then packed into a single texture, exactly as for the other BVH variants. However, we also upload an array of structures containing additional data for each object. Each structure contains the index of the object's root node, its bounding box, and its inverse transformation matrix.

Traversing the per-object BVH works by traversing each BVH individually. Since the hierarchies are in object space, we must transform the ray to the object space of the mesh. This transformation is done by multiplying the ray origin and direction with the inverse transformation matrix of the object. When the ray has been transformed to object space, the traversal works precisely as for the stackless BVH.

4.3.5 Uniform 3D Grid

The grid we implemented is constructed as described in Section 3.2.3. The resolution of the grid is calculated using Cleary's heuristic with $\lambda = 3$. To encode the grid, we flatten the triangle lists in all the cells to a single array. Then for each cell, the minimum and maximum index to the triangle array are stored. We then encode these indices in a three-dimensional texture with the same resolution as the grid. Metadata such as the resolution and bounds of the grid is uploaded as uniform variables or as pre-processor definitions to the shader.

The grid is traversed using the 3D-DDA algorithm described in Section 3.2.4, using early exits. We do not, however, make use of any space skipping, such as distance fields. The traversal process is summarised below:

Grid traversal summary:

1. If the ray intersects the grid, find the first grid cell along the ray.
2. Test the ray for intersection with all the triangles in the current cell.
3. Early exit if the closest intersection point along the ray lies within the bounds of the current cell. Otherwise, move to the next cell along the ray and go to step 2.
4. Terminate when all cells along the ray have been visited.

4.3.6 Irregular Grids

We also implemented an irregular grid, as described in Section 3.2.5. There are two significant differences between our implementation and the implementation described by Pérard-Gayot et al. [27]. Firstly, we construct the grid sequentially on the CPU, instead of in parallel on the GPU. Secondly, we use grids for the second level of the initial two-level ADS instead of octrees. However, there is no conceptual difference since our grids are restricted to the same resolution as the octrees, i.e. the resulting subdivision of space is identical.

The irregular grid is encoded as a single three-dimensional texture, with the same resolution as the virtual grid. Each pixel in the texture encodes the contents of the irregular grid cell covering the corresponding virtual cell. Since each cell may cover many virtual cells, there is some redundancy in the texture. That is, all pixels in the texture containing the same cell contains the same data. This redundancy can be removed by adding a level of indirection, i.e. keeping the mapping from the virtual cell and the cell contents in two separate textures. However, that would increase the number of texture fetches from one to two per cell when tracing.

Each cell is encoded to a four component vector of 32-bit integers, i.e. a structure of 16 bytes, containing both the minimum and maximum triangle index as well as the bounds of the cell. The first two integers contain the minimum and maximum triangle index, while the cell bounds are packed into the remaining two integers. The cell bounds can be packed as integers since they align to the virtual grid. Therefore, we only store the minimum and maximum virtual grid index and reconstruct the bounds in the shader by multiplying the indices by the virtual cell size. The six indices, i.e. min and max for x,y, and z, are restricted to ten bits each, allowing us to pack them into two 32-bit integers. Limiting the indices to ten bits restricts the virtual grid size to 1024 in all directions. However, this is not a problem since such large grids are well above our memory limits. For example, even if each cell only contained one 32-bit integer, a 1024^3 grid would require four gigabytes of memory which is above the memory limit for many graphics cards.

4.4 Noise reduction

We chose to base our de-noising work on the methodology of Dammertz et al. [31], Mara et al. [36] and Schied et al. [35], because they are designed for interactive path tracing. We implement Dammertz edge-avoiding À-Trous wavelet transform as described in Section 3.3.1. We use a geometry buffer containing the surface normals as well as surface positions to guide the filter. The geometry buffer also contains the surface albedo (colour or texture) which is separated from the image before it is filtered to avoid blurring it. An example of how the geometry buffer might look is shown in Figure 4.3. Thus, we defer the albedo from the path tracer in the first bounce and apply it after performing the filtering, similar to Mara et al., Schied et al. and Bako et al. [36, 35, 33]. The albedo is applied to the filtered image in a post-processing step.

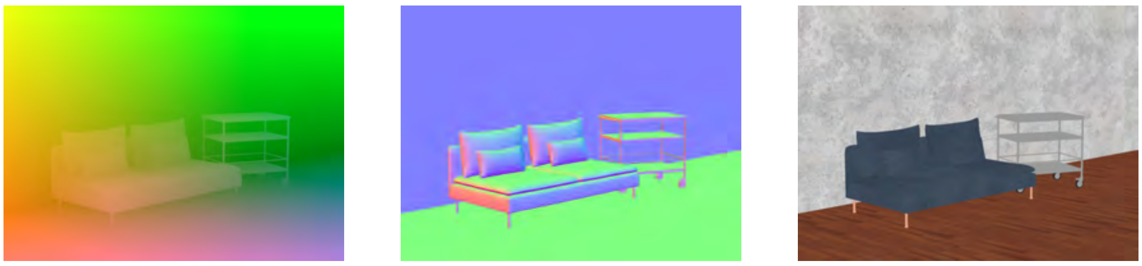


Figure 4.3: Geometry buffer contents. To the left: surface positions, middle: surface normals, and to the right: the surface albedo.

We also filter some lighting terms separately to retain some aspects of the image better. Firstly, the direct and indirect lighting are separately filtered as done by Schied et al. [35] to reduce over-blurring of sharp shadows. The lighting is disjointed in the path tracer and stored in separate textures. The direct lighting is filtered less aggressively with a lower filter size. We use three iterations of Dammertz's filter on the direct lighting, and five on the indirect. However, reflections are not retained using Schied et al.'s separation method. Therefore, we also filter two other lighting terms separately, similarly to how Mara et al. did. We filter the diffuse and specular reflections at the first surface interaction independently. The filtering system does not use both strategies simultaneously. Instead, we use one strategy at a time and can switch between them using pre-processor macros similarly to how we switch between acceleration structures. A summary of the filtering and deferred shading we use is described below.

Summary of the noise reduction process

1. Before path tracing, calculate the surface albedo, surface normals and world-space surface positions for the scene and store them in a geometry buffer.
2. In the path tracer, separate the lighting into two textures. Either separate the diffuse and specular terms or the direct and indirect terms.
3. After the path tracing is finished, send the two textures as well as the geometry buffer as input to the filtering kernel. The filter parameters are fine-tuned for each scene.
4. Perform five iterations of the filter on the diffuse or indirect terms. On the glossy or direct terms perform three iterations.
5. Send the filtered images as well as the albedo map of the geometry buffer as input to a post-processor shader. Add the two lighting textures together and multiply them with the albedo to achieve the final image.

5

Results

To evaluate the renderer, we measured the performance of the acceleration structures and the noise reduction filters. The results for the acceleration structures are covered in Section 5.1. The noise-reduction results are presented in Section 5.2. All metrics were collected on a desktop computer running Ubuntu 17.10, with an *Intel i7-4930K* CPU and an *Nvidia GeForce GTX Titan* GPU.

5.1 Acceleration Structures

The primary goal for the renderer is to maintain interactive frame rates, i.e. above $10hz$. It is thus necessary that the path tracer is fast enough to sample every pixel of an image within an interactive time span, i.e. within 100 milliseconds. We valuate the ray tracing performance by measuring the time required to sample one path per pixel along with the number of rays traced per second. The execution time of the path tracer is measured using a GPU timer included in WebGL. The ray tracing rate, i.e. number of rays per second, is measured by counting the average number of rays traced per frame, and dividing it by the execution time of the path tracer. The execution times are averages of 25 runs and the number of rays traced are averages of ten independent frames.

We evaluate the acceleration structures on a small set of scenes of varying complexity. Testing the acceleration structures on several scenarios serves two purposes. Firstly, it lets us observe the performance as the scene complexity increases. Secondly, it ensures that the renderer is not optimised for a single scenario. The scenes we use are shown in Table 5.1.

Scene	Triangle Count	View
Cornell Box	34	
Living Room	10514	
Record Player	79667	
Crytek Sponza	279162	

Table 5.1: Table listing the scenes used to evaluate the renderer. The images are rendered using our path tracer.

5.1.1 Ray Tracing Performance

Table 5.2 lists the rendering time and the measured number of rays per second for the different scenes and acceleration structures. The time listed is the average number of milliseconds it takes to sample one path per pixel for a 720 by 540 pixel image. We measure the number of rays per second for two different scenarios. In the first scenario, the image is traced as usual with a maximum path length of 5 and one light sample per scattering event. In the second scenario, only primary rays are traced, resulting in higher ray coherence.

For all scenes except Crytek Sponza, the path tracer reaches interactive framerates. On the smallest measured scene, the Cornell Box, it executes in 10ms while on the largest scene, the Crytek Sponza, the fastest execution time is 161 ms, resulting in a frame rate of at most five to six frames per second. Overall, the best performance is achieved with the Irregular grid, though all BVH variants outperform the grids on the Cornell Box. Furthermore, the performance difference between the Irregular Grid and the BVHs increase with the size of the scene. The ray tracing rate decreases as the number of triangles increases, spanning from 288MRay/s to 24MRay/s from the Cornell Box to the Crytek Sponza. There is also a gap between the ray tracing rate for standard paths and primary rays. Tracing only primary rays increases the ray tracing rate by around 200% to 300% for most scenes. The only exception is the BVHs on the Sponza, where the increase is around 30% to 40%.

Scene	ADS	Time (ms)	MRay/s	Primary MRay/s
Cornell Box	BVH	12.00	259	529
	Stackless BVH	10.96	283	637
	Per-object BVH	**10.80**	**288**	**845**
	Uniform Grid	16.58	183	806
	Irregular Grid	15.20	204	679
Living Room	BVH	42.30	42	144
	Stackless BVH	36.60	49	168
	Per-object BVH	42.01	42	168
	Uniform Grid	70.75	25	111
	Irregular Grid	**34.95**	**51**	**236**
Record Player	BVH	33.09	15	62
	Stackless BVH	26.53	19	69
	Per-object BVH	29.66	17	63
	Uniform Grid	53.76	9	24
	Irregular Grid	**14.26**	**36**	**132**
Crytek Sponza	BVH	700	5.4	7.6
	Stackless BVH	599	6.1	8.7
	Per-object BVH	603	6.4	8.2
	Uniform Grid	258	14.6	82
	Irregular Grid	**161**	**24.1**	**91**

Table 5.2: Table listing the performance metrics of the acceleration structures on the test scenes. The metrics shown are, firstly, the running time in milliseconds for sampling one path per pixel. Secondly, the measured ray tracing performance in terms of million rays per second. Finally, the ray tracing performance when only tracing primary rays. The best results for each scene are marked in bold. Note the discrepancy in ray tracing performance between tracing normal (scattered) paths, versus only tracing primary rays.

5.1.2 Ray Coherence & GPU Utilisation

We measure the relation between ray coherence and performance by comparing the tracing performance of the renderer to an altered version where the rays are forced to be mostly coherent. In the unaltered version of the path tracer, the pseudo-random number generators are seeded by the pixel position and the iteration number. Thus, the behaviour of each path is practically independent. In the altered version, the random generators are seeded only by the iteration number, which causes the paths to be dependent on each other. The effect of this change on the behaviour of the paths is shown in Figure 5.1. In the altered version, rays hitting similar surfaces, i.e. the same material and orientation, are reflected in the same direction, resulting in highly coherent rays.

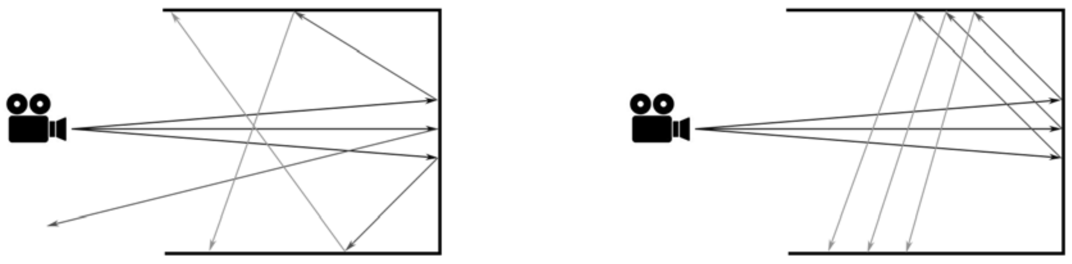


Figure 5.1: Figure showing the difference in ray coherence between seeding the pseudo-random generator with iteration and pixel position (left) versus seeding with only the iteration number (right).

The difference in performance between the two ways to seed the random generator is listed in Table 5.3. The table lists the tracing rate with and without forced coherence for a stackless BVH. Forcing the coherence increases the rate of traced rays for all scenes but not as much for the record player. The increase on the record player is smaller because the average path length is low. Due to the shape of the scene, many of the paths miss the scene entirely, or exit the scene after a low number of reflections. The ray coherence is higher when the average path length is low since the paths diverge when they are reflected.

Scene	MRay/s	Coherent MRay/s	Increase
Cornell Box	274	452	65 %
Living Room	47	122	160 %
Record Player	17	18	6 %
Crytek Sponza	7	12	71 %

Table 5.3: Table listing the ray tracing performance of a stackless BVH when tracing as usual compared to the tracing performance when ray coherence is forced.

5.1.3 ADS Construction

The tracing performance is not the only important aspect of the acceleration structures. The build time is also relevant because it defines the delay between loading or modifying a scene and starting the rendering process. We measure the average build times of the acceleration structures on the test scenes. The results are listed in Table 5.4. Keep in mind that for the per-object BVH, the listed results are not applicable when the scene is modified by transforming objects. Thus, when using the per-object BVH, we can move, rotate, and scale objects dynamically.

The table shows that the fastest build time varies between 1ms on the Cornell Box (34 triangles) and 19 seconds on Crytek Sponza (279162 triangles). Overall, only the Cornell Box is constructed fast enough for real-time ADS reconstruction. The Living Room (10514 triangles) and Record player (79667 triangles), which are representative of our problem domain, both yield build times well above what is plausible for interactive rates. The fastest build time for the living room results in at most five rebuilds per second, while the record player takes several seconds to construct.

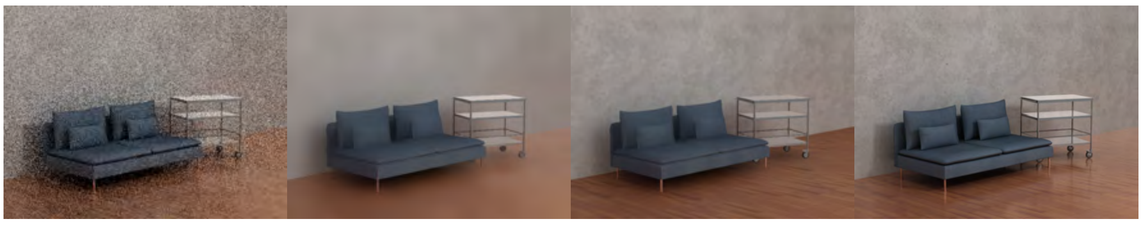
Scene	Cornell Box	Living Room	Record Player	Crytek Sponza
Triangle Count	34	10514	79667	279162
BVH	0.98 ms	241 ms	**2838 ms**	28171 ms
Stackless BVH	**0.86 ms**	303 ms	3302 ms	30779 ms
Per Object BVH	2.82 ms	396 ms	4454 ms	27945 ms
Uniform Grid	4.38 ms	**225 ms**	5771 ms	**19004 ms**
Irregular Grid	2.93 ms	1096 ms	11161 ms	54383 ms

Table 5.4: Lists the mean ADS build time in milliseconds for the different scenes. The fastest build times for each scene are shown in bold. Each listed time is the average of 25 runs. Note that the variance is high for the Cornell Box and the Crytek Sponza due to the very short and very long build times. Also, keep in mind that the acceleration structures are built in JavaScript without any parallelism.

5.2 Noise Reduction

We evaluate the two key properties of our noise reduction: the visual quality, and the execution time. The execution time is relevant since the filter must run at interactive frame rates. We measure the execution time using the GPU timer, and take the average of 100 measurements. To quantitatively measure the quality of a filtered image, we use the structural similarity index (SSIM) [48]. SSIM measures the similarity between two images, where a value of 1 means that the images are identical, while a value of 0 implies that they are entirely dissimilar. To evaluate the quality of two images, we compare both to an image we consider the *ground truth*, i.e. an image we consider as the correct result. The ground truth images are rendered by letting the path tracer converge with filtering disabled, stopping at 2048 samples per pixel. The filters are thus evaluated by applying the filters on a one sample per pixel input image and comparing the results to the ground truth.

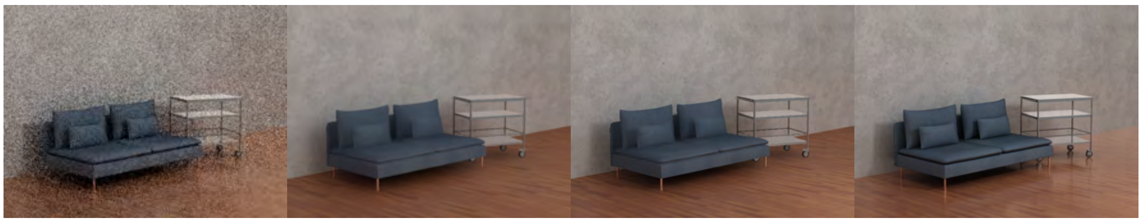
Figure 5.2 shows the effect of separating the albedo from the filter input. In both cases, the image is filtered with 5 iterations of the wavelet filter. The total execution time of the filter is 5.7 milliseconds. The result of not separating the albedo is shown in Figure 5.2b. It is clear that the texture detail, such as on the floor and wall, is over-blurred. Filtering with albedo separation is shown in Figure 5.2c, where the texture detail is better retained.



(a) SSIM = 0.31 (b) SSIM = 0.87 (c) SSIM = 0.93 (d) SSIM = 1.00

Figure 5.2: Shows the effects of demodulating the albedo and adding it during the post-processing. The 1 spp filter input is shown in **(a)**. The result of filtering without separating the albedo is shown in **(b)**, while the result with the albedo split lies in **(c)**. The ground truth is shown in **(d)**.

The effect of filtering the direct and indirect illumination separately, as by Schied et al. [35], is shown in Figure 5.3. Filtering without any separation is shown in Figure 5.3b. The result of separating the illumination is shown in Figure 5.3c. The direct and indirect illumination is filtered with three and five iterations of the wavelet filter respectively. The average total execution time is 8.3 milliseconds. Splitting the illumination results in better preservation of sharp shadows, which can be observed on the wall behind the sofa and behind the pillows in the sofa.



(a) SSIM = 0.31 (b) SSIM = 0.93 (c) SSIM = 0.93 (d) SSIM = 1.00

Figure 5.3: Shows the impact of filtering the direct and indirect illumination separately. **(a)** shows an unfiltered image at 1spp. **(b)** shows a filtered image with deferred albedo. **(c)** shows the combination of deferring the albedo and splitting the indirect and direct lighting. **(d)** shows the ground truth.

The differences when separately filtering direct and indirect illumination versus separately filtering glossy and diffuse reflections can be observed in a scene with a highly reflective surface, such as in Figure 5.4. Filtering the direct and indirect illumination separately (Figure 5.4b) results in an over-blurred reflection on the sphere. The reflection is better preserved when glossy and diffuse reflections are filtered separately (Figure 5.4c), as done by Mara et al. [36] and Bako et al. [33]. The glossy and diffuse reflections are filtered with two and five iterations of the wavelet filter respectively, which takes 8.3 milliseconds to execute.

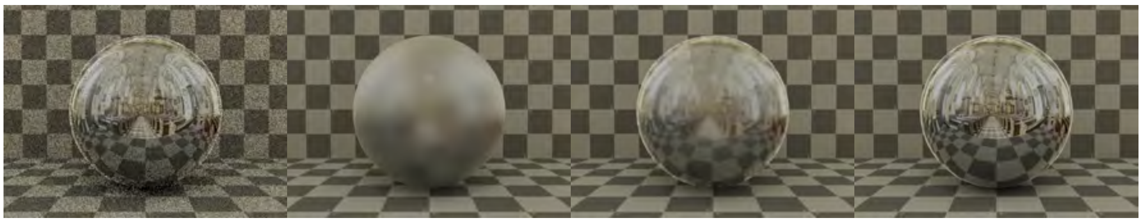

(a) SSIM = 0.41 (b) SSIM = 0.84 (c) SSIM = 0.89 (d) SSIM = 1.0

Figure 5.4: Reflections: comparison between separating the filtering of the glossy/diffuse light and separating the filtering of the direct/indirect lighting. **(a)** shows an unfiltered 1spp image. **(b)** shows a 1spp image filtered separately for indirect and direct lighting. **(c)** shows glossy and diffuse filter separation also for a 1spp image. **(d)** is the ground truth.

In the living room scene, the direct/indirect and glossy/diffuse splitting results in the same SSIM value. However, there are still some notable differences. For example, the sharp shadow of the sofa is clearer in 5.5a than in 5.5b. Similarly, the glossy floor reflections are somewhat distinguishable in Figure 5.5b and completely blurred out in Figure 5.5a. Both preserves most parts of the ground truth in Figure 5.5c but neither are able to both retain glossy reflections and hard shadows.

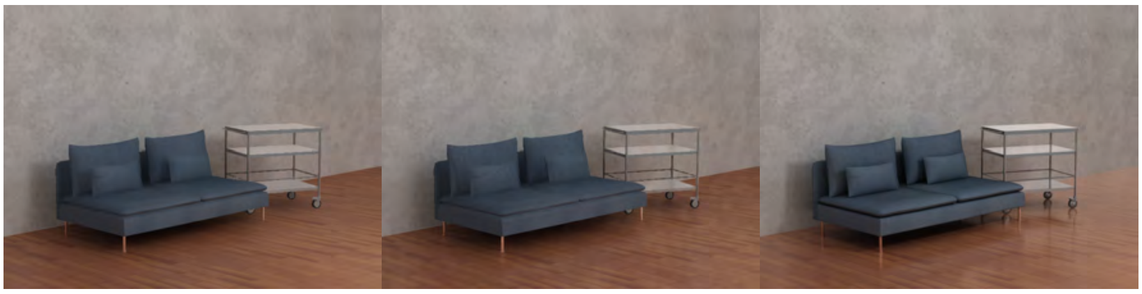

(a) SSIM = 0.93 (b) SSIM = 0.93 (c) SSIM = 1.00

Figure 5.5: Shows the difference between splitting the illumination into indirect/direct versus glossy/diffuse when filtering. **(a)** shows the result of filtering the direct/indirect illumination, while **(b)** shows the glossy/diffuse filtering. **(c)** shows the ground truth.

6
Discussion

This chapter discusses the implications of the results presented in Chapter 5. We first discuss the performance of the path tracer and the acceleration structures, followed by a discussion on the noise filtering. Finally, we touch upon potential ethical dilemmas related to this project.

6.1 Ray Tracing & Acceleration Structures

The performance metrics listed in Table 5.2 show that we can render images at interactive frame rates, albeit low-resolution images and relatively small scenes. However, our ray tracing performance is somewhat low when compared to the literature. For example, our implementation of the irregular grid achieves a rate of 91 million primary rays per second on the Crytek Sponza, while the original authors report a rate of 653 million primary rays, or 274 million randomly oriented rays, per second for the same scene [27]. However, they use a somewhat more powerful GPU (Geforce Titan X (Maxwell)) and somewhat different construction parameters when evaluating the performance. Nevertheless, there is a significant gap in performance between our implementation and the state-of-the-art. Since the gap is apparent for primary rays, it is not caused by incoherent rays and low GPU utilisation. Therefore, the gap is likely caused by either us having a less efficient ray tracing kernel, or less efficient memory accesses. When tracing primary rays, our tracing kernel is quite minimal and uses the same ray-triangle intersection test as Pérard-Gayot et al. [27]. It is therefore unlikely that optimising the tracing kernel and intersection test would increase the performance by a significant factor. Therefore, it seems likely that we can make the most substantial performance gains by optimising the encoding and fetching of the triangle data. For instance by sharing vertices between triangles, e.g. using triangle strips, or by optimising how the triangles are laid out in memory, i.e. increasing the cache locality.

Regarding acceleration structures, the irregular grid outperforms the three BVH variants while the uniform grid lags behind. The BVHs also seem to perform proportionally worse compared to the irregular grid as the number of triangles increases. It is especially evident on the Crytek Sponza, where the BVHs only reach a ray tracing rate of around 25% of the irregular grid and around 40% of the uniform grid. However, though our irregular grid outperforms our BVHs, one should not conclude that the irregular grids outperform BVHs in general. For instance, Ylitie et al. report that their compressed wide BVH outperforms the irregular grid when tracing incoherent rays [23].

There are several possible explanations for why the BVHs might scale worse than the grids. One is that the Crytek Sponza is quite unevenly tesselated, causing a large amount of nodes to overlap for the BVHs. The overlapping can be reduced by using the Split BVH introduced by Stich et al. [12]. Another potential explanation is that the BVHs are limited by memory accesses on the Sponza. The BVH nodes are larger in terms of memory than the grid nodes since the node bounds are stored as floating point vectors. That is, in the uniform grid the bounds are implicit, and for the irregular grid they can be packed more efficiently since they align to the virtual grid. The size of the BVH nodes can be reduced by discretising the BVH bounds, i.e. align them to some grid spanning the scene, and packing them as integers, similarly to how the irregular grid is encoded.

6.1.1 Ray Coherence & GPU Utilisation

The results in Table 5.2 and Table 5.3 shows, as expected, that increasing ray coherence leads to a significant increase in ray tracing performance. The magnitude of the increase raises the question of how the path tracer can be structured to improve the GPU utilisation. Approaches such as wavefront path tracing and persistent threads are impractical due to the limitations of the WebGL API. However, it might be possible to use parts of those approaches to improve the utilisation.

We can *technically* use the seeding change described in Section 5.1.2 to speed up the path tracer. As long as we choose the random generator and per iteration seeds such that the random number sequences are independent across iterations, the path tracer converges to the correct result. However, forcing coherence that way is undesirable for real-time purposes because the colour variance manifests itself in a way that looks worse than the usual grainy noise. When forcing ray coherence, all rays originating at the same triangle are parallel. Therefore, in each iteration, the indirect lighting on each triangle is an orthographic projection of the surrounding scene onto the triangle from the sampled direction. At low sample counts, the resulting indirect illumination contains sharp and artificial looking patterns. However, it might be possible to hide these patterns and keep the performance gains. By changing the seeding such that all threads within a warp share a seed, we get one path behaviour for each block of pixels rendered by the same warp. The patterns would thus no longer be the same across each surface, but different patterns would appear in each block of pixels. We can avoid these blocky patterns by letting each warp render a sparse pattern of pixels instead of a compact block of pixels. We could achieve this by using some mapping or pairing function. For example, if we pair up every pixel with another pixel in a nearby block, and let each fragment shader trace a path corresponding to the pixel position of its pair, then each warp would trace a sparse pattern of pixels. The mapping can be reversed in the post-processing such that each pixel appears in the correct place. If the mapping function is chosen such that neighbouring pixels are traced by different warps, the patterns caused by sharing seeds between threads would be harder to notice.

Also, because the ray coherence decreases as the average path length increases, the shape of a scene can impact the ray coherence, since the average path length can depend on the scene. For example, in an open scene, such as a planar surface with some objects on top, many rays are reflected out of the scene and terminate. Conversely, for an enclosed scene, e.g. like the interior of a room, the average path length is likely higher since fewer paths are scattered out of the scene. Do keep in mind that terminated paths are only beneficial if all paths within a warp are terminated. However, using a scene where the average path length is short can improve the tracing performance. One example of this is that the irregular grid achieves similar framerates on the Cornell box and the record player, despite the Cornell box having 34 triangles compared to the 79667 of the record player.

6.1.2 ADS Construction

The build times reported in Section 5.4 are not fast enough to rebuild the scene at interactive framerates. The only exception is the build times for the Cornell box, where building an ADS is trivial due to the low triangle count. Dynamically changing a scene would, therefore, introduce long delays between changing an object and displaying the result. The ability to rebuild the ADS once per frame is required for the user to be able to dynamically rearrange the scene, i.e. moving, rotating or scaling objects.

We are only able to achieve dynamic updates using the per-object BVH, where we do not need to reconstruct the ADS when objects are transformed. Furthermore, the per-object BVH performs at par with the stackless BVH across all scenes. However, we have not evaluated the per-object BVH on a scene with a large number of objects, where the overhead would be more significant. On the other hand, our per-object can be improved to better handle a large number of objects by replacing the array of objects with an acceleration structure, e.g. a BVH.

Our build times are several orders of magnitude slower than the build times reported in the literature [27, 13, 26]. The primary reason for this discrepancy is that we do not utilise the GPU, or any parallelism for that matter, when constructing our acceleration structures. Utilising the GPU for ADS construction is complicated by the limitations of WebGL, i.e. the lack of shared memory and random access writes. It might be less of an issue in the future, should compute shaders be introduced to WebGL.

6.2 Noise Filtering

All filtering versions of the path tracer execute in real time, at an average execution time below 10 ms. Separating the albedo term of the first surface interaction improves the retention of texture details, as can be seen in Figure 5.2. However, when compared to Mara et al. [36] and Schied et al. [35] the filter lacks in one area in particular. Namely, their filter steps use temporal filtering and temporal anti-aliasing. Our lack of temporal accumulation when moving the camera causes some flickering artefacts. Also, our lack of anti-aliasing results in highly aliased edges when filtering is enabled. The addition of temporal filtering would result in a lower variance on the filter input, i.e. by re-projecting previous the samples of previous frames; we would have more than one sample per pixel. The lower variance could potentially reduce the problem with the blotchy artefacts, as seen in Figure 5.5b and the reflection in Figure 5.4c. The temporal accumulation would, of course, come with a computational cost, although, the decrease in variance could potentially decrease the number of filter iterations required for a noise-free result.

Splitting the light that goes into the filter did not produce all-round image-detail improvements. Despite retaining sharp shadows, separately filtering the direct and indirect lighting did not preserve sharp reflections. For example, note the lack of reflection in Figure 5.5a and 5.4b compared to the ground truth. It left a major gap in quality for scenes with any reflective materials. Furthermore, for scenes where no sharp shadows are present, i.e. when the light sources are large, splitting the illumination makes less of a difference. Also, it is not possible to preserve gloss by filtering the direct/indirect lighting less harshly with this setup. It yielded artefacts in the form of blotchy areas. Note that the gloss-retaining filter separation does not remove *all* shadows, but over-blurs them. The result is that all shadows are *soft*, regardless of the shape of the light source. Conversely, while the shadow-preserving filter does retain shadows well, it does not retain *any* sharp reflections. The glossy filtering also caused blotchy artefacts in reflections. Using more than two filter iterations often smudged any sharp reflections, but filtering less harshly produced artefacts. It is possible that introducing more samples per pixel through temporal filtering would mitigate the blotch artefacts. Reducing the variance by temporal accumulation could also allow for less harsh filtering, which in turn would reduce the over-blurring.

On another note, there is a discrepancy in filter quality *between scenes*. Because we used an edge-stopping function based on the position buffer, the filter is affected by the scale of the scene geometry. Namely, the relative distance to nearby geometry varies depending on the scale of the scene. Therefore, the parameter controlling that edge-stopping required fine-tuning for each scene. The fine-tuning could be automated, either by determining the parameters based on the dimensions of the scene or calculating the parameters from an estimated variance of the position buffer.

6.3 Ethical considerations

The project itself does not have many ethical considerations when used in the context of its purpose. However, while the purpose of the project is to visualise products such as furniture, it does not hinder the user from using other 3D models. The path tracer can thus be used to portray virtually anything. Thereby comes some potential ethical implications. It could, for example, be used to render visually disturbing imagery or propaganda. Although, the same is true for any 3D-rendering software. There is also a wider dilemma in computer graphics regarding how photorealistic computer graphics, especially in virtual or augmented reality, affects the human psyche. Though this dilemma is somewhat out of scope for this project since we do not focus on creating an immersive experience.

The expansion of our and similar research strives toward a future with real-time path tracing. However, as with any research on the topic, we cannot control how such path tracing is used in its entirety. That responsibility lies with those who use it.

The program itself used in its real context, visualising products in a web browser, does not have any negative impact on the user. If anything, rendering the visualisations locally in a web browser comes with several benefits. One such benefit is that the user gets more immediate feedback from the system, reducing time wasted waiting on images from the render farm. Another benefit is that the system has the potential to reduce or even remove the need for the render farm and the back-end systems that support it, reducing the economic and perhaps also environmental cost of the system. Lastly, depending on how the technology is distributed, it may have a lower barrier of entry than the alternatives used today. In conclusion, the project, when used as intended, does not implicate any tangible ethical issues.

7
Conclusions

We can conclude that our path tracer can render small scenes in real time while also filtering away much of the inherent noise. We can also let the renderer accumulate samples, resulting in locally rendered high-quality images of the scene. We evaluated the path tracer using a resolution of 720×540 pixels. It is possible to render higher resolution images, though the rendering time increases more or less linearly with the number of pixels. The path tracer can, therefore, be used for both real-time visualisations and high-quality previews in RapidImages' development tool. However, the development tool can render more complex scenes than, for example, the living room with around 10000 triangles or the record player of roughly 80000 triangles. It may therefore only be possible to use the path tracer on scenes where the number of objects and triangles is relatively small.

The construction of the acceleration structures in JavaScript is not fast enough for rebuilding the scene at an interactive framerate. For instance, the fastest construction time of the living room scene would allow at most four rebuilds per second. By using a per-object BVH, we can avoid rebuilding the acceleration structures when applying affine transformations on objects. We can, therefore, render dynamic, e.g. moving, objects in real time. The per-object BVH also allows us to pre-compute the acceleration structures for each object, reducing the initial loading time of the application.

Furthermore, the noise reduction filter mitigates the path tracer noise. However, it is not able to preserve both sharp reflections and sharp shadows, and it introduces blotch artefacts and aliasing. With regard to future work, both issues can be reduced by introducing temporal filtering and temporal anti-aliasing. The next section expands upon the most promising future work regarding our project.

7.1 Future Work

We found several interesting areas in which our work can be extended and improved. These areas include extending our noise filters to retain image details better, improving the acceleration structures, and increasing the GPU utilisation.

Regarding the noise reduction, the obvious next step is to extend the filter with temporal accumulation and anti-aliasing. We can also address the issue of not retaining both shadows and glossy reflections by splitting the illumination even further. For instance, by filtering the direct lighting, glossy reflections, and diffuse reflections separately, we could retain both shadows and reflections at the price of a slower filter. It might, however, be unnecessary if temporal accumulation is added.

There are many ways in which the acceleration structures can be improved. One could, for example, reduce the number of texture fetches during ray traversal by optimising the encoding of the triangles and acceleration structures. For instance, the triangles within each ADS node could be encoded as a triangle strip, i.e. sharing vertex data between triangles that share an edge. Also, discretising the BVH node bounds would allow each node to packed in a single four-component vector, which enables retrieving a node in a single texture fetch. Improving the BVH encoding can be combined with implementing a more recent BVH variant, such as the compressed wide BVH by Ylitie et al. [23].

Building acceleration structures in JavaScript and without parallelism leads to very long construction times, especially compared to construction algorithms utilising the GPU. As mentioned in Section 6.1.2, the restrictions of WebGL complicates the utilisation of the GPU for ADS construction. However, it would nevertheless be interesting to explore how the construction can be sped up using WebGL.

Using per-object acceleration structures to avoid ADS constructions shows much potential. The next steps for improving our per-object approach would be to sort the objects into a top-level acceleration structure, as in Embree [16] and Optix [17]. For small sets of objects, the top-level ADS would be small and fast to reconstruct and would, therefore, still allow for dynamic updates of the scene. However, building acceleration structures for each object can induce a traversal overhead compared to a single ADS containing all geometry. That is, the per-object acceleration structures are locally optimised rather than globally. This overhead can potentially be reduced by the partial re-braiding algorithm described by Benthin et al. [18].

Finally, there are a few possibilities for improving the GPU utilisation. One such possibility is to introduce some of the concepts from wavefront path tracing. For example, decomposing the path tracer into smaller specialised programs. However, this approach assumes that data can be effectively stored and retrieved across executions of the smaller programs. In WebGL, the only way of keeping a state across several executions of a shader is to read and write to a set of textures. It is therefore uncertain whether the decomposition of the path tracer would be beneficial. Another potential approach is the seeding change discussed in Section 6.1.1, where forcing coherence within a warp and letting it render a sparse pattern of pixels may simultaneously improve the GPU utilisation and avoid sharp patterns.

Bibliography

[1] T. Whitted, "An improved illumination model for shaded display," in *ACM SIGGRAPH Computer Graphics*, vol. 13, p. 14, ACM, 1979.

[2] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *ACM SIGGRAPH Computer Graphics*, vol. 18, pp. 137–145, ACM, 1984.

[3] J. T. Kajiya, "The rendering equation," in *ACM Siggraph Computer Graphics*, vol. 20, pp. 143–150, ACM, 1986.

[4] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 21, pp. 703–712, ACM, 2002.

[5] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the conference on high performance graphics 2009*, pp. 145–149, ACM, 2009.

[6] T. Aila, S. Laine, and T. Karras, "Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum," *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

[7] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: wavefront path tracing on gpus," in *Proceedings of the 5th High-Performance Graphics Conference*, pp. 137–143, ACM, 2013.

[8] J. Bikker and J. van Schijndel, "The brigade renderer: A path tracer for real-time games," *International Journal of Computer Games Technology*, vol. 2013, 2013.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[10] H. Dammertz, J. Hanika, and A. Keller, "Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays," in *Computer Graphics Forum*, vol. 27, pp. 1225–1233, Wiley Online Library, 2008.

[11] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, p. 6, 2007.

[12] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*, pp. 7–13, ACM, 2009.

[13] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*, pp. 89–99, ACM, 2013.

[14] M. Vinkler, V. Havran, and J. Bittner, "Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing," in *Computer Graphics Forum*, vol. 35, pp. 68–79, Wiley Online Library, 2016.

[15] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 3, pp. 153–166, 1990.

[16] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: a kernel framework for efficient cpu ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 143, 2014.

[17] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, *et al.*, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 66, 2010.

[18] C. Benthin, S. Woop, I. Wald, and A. T. Áfra, "Improved two-level bvhs using partial re-braiding," in *Proceedings of High Performance Graphics*, p. 7, ACM, 2017.

[19] B. Smits, "Efficiency issues for ray tracing," *Journal of Graphics Tools*, vol. 3, no. 2, pp. 1–14, 1998.

[20] R. Torres, P. J. Martín, and A. Gavilanes, "Ray casting using a roped bvh with cuda," in *Proceedings of the 25th Spring Conference on Computer Graphics*, pp. 95–102, ACM, 2009.

[21] N. Binder and A. Keller, "Efficient stackless hierarchy traversal on gpus with backtracking in constant time," in *Proceedings of High Performance Graphics*, pp. 41–50, Eurographics Association, 2016.

[22] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs," in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pp. 49–57, IEEE, 2008.

[23] H. Ylitie, T. Karras, and S. Laine, "Efficient incoherent ray traversal on gpus through compressed wide bvhs," in *Proceedings of High Performance Graphics*, p. 4, ACM, 2017.

[24] A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated ray-tracing system," *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16–26, 1986.

[25] D. Jevans and B. Wyvill, "Adaptive voxel subdivision for ray tracing," 1988.

[26] J. Kalojanov, M. Billeter, and P. Slusallek, "Two-level grids for ray tracing on gpus," in *Computer Graphics Forum*, vol. 30, pp. 307–314, Wiley Online Library, 2011.

[27] A. Pérard-Gayot, J. Kalojanov, and P. Slusallek, "Gpu ray tracing using irregular grids," in *Computer Graphics Forum*, vol. 36, pp. 477–486, Wiley Online Library, 2017.

[28] M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S.-E. Yoon, "Recent advances in adaptive sampling and reconstruction for monte carlo rendering," in *Computer Graphics Forum*, vol. 34, pp. 667–681, Wiley Online Library, 2015.

[29] F. Durand, N. Holzschuch, C. Soler, E. Chan, and F. X. Sillion, "A frequency analysis of light transport," in *ACM Transactions on Graphics (TOG)*, vol. 24, pp. 1115–1126, ACM, 2005.

[30] F. Rousselle, C. Knaus, and M. Zwicker, “Adaptive sampling and reconstruction using greedy error minimization,” in *ACM Transactions on Graphics (TOG)*, vol. 30, p. 159, ACM, 2011.

[31] H. Dammertz, D. Sewtz, J. Hanika, and H. Lensch, “Edge-avoiding à-trous wavelet transform for fast global illumination filtering,” in *Proceedings of the Conference on High Performance Graphics*, pp. 67–75, Eurographics Association, 2010.

[32] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, “Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder,” *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 98, 2017.

[33] S. Bako, T. Vogels, B. Mcwilliams, M. Meyer, J. NováK, A. Harvill, P. Sen, T. Derose, and F. Rousselle, “Kernel-predicting convolutional networks for denoising monte carlo renderings,” *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 97, 2017.

[34] K. Dahm and A. Keller, “Learning light transport the reinforced way,” *arXiv preprint arXiv:1701.07403*, 2017.

[35] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi, “Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination,” in *Proceedings of High Performance Graphics*, HPG ’17, (New York, NY, USA), pp. 2:1–2:12, ACM, 2017.

[36] M. Mara, M. McGuire, B. Bitterli, and W. Jarosz, “An efficient denoising algorithm for global illumination,” *Proceedings of High Performace Graphics*, vol. 6, 2017.

[37] E. Eisemann and F. Durand, “Flash photography enhancement via intrinsic relighting,” in *ACM transactions on graphics (TOG)*, vol. 23, pp. 673–678, ACM, 2004.

[38] B. Karis, “High-quality temporal supersampling,” *Advances in Real-Time Rendering in Games, SIGGRAPH Courses*, vol. 1, 2014.

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

[40] S. M. Rubin and T. Whitted, “A 3-dimensional representation for fast rendering of complex scenes,” in *ACM SIGGRAPH Computer Graphics*, vol. 14, pp. 110–116, ACM, 1980.

[41] T. Aila, T. Karras, and S. Laine, “On quality metrics of bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, pp. 101–107, ACM, 2013.

[42] J. G. Cleary, B. M. Wyvill, G. Birtwistle, and R. Vatti, “Design and analysis of a parallel ray tracing computer,” in *Graphics Interface*, vol. 83, pp. 33–38, 1983.

[43] A. Es and V. İşler, “Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor,” *Journal of Parallel and Distributed Computing*, vol. 67, no. 11, pp. 1201–1217, 2007.

[44] S. Mallat, *A wavelet tour of signal processing: the sparse way.* Academic press, 2008.

[45] B. Burley and W. D. A. Studios, “Physically-based shading at disney,” in *ACM SIGGRAPH*, vol. 2012, pp. 1–7, 2012.

[46] T. Trowbridge and K. P. Reitz, “Average irregularity representation of a rough surface for ray reflection,” *JOSA*, vol. 65, no. 5, pp. 531–536, 1975.

[47] T. Möller and B. Trumbore, “Fast, minimum storage ray/triangle intersection,” in *ACM SIGGRAPH 2005 Courses*, p. 7, ACM, 2005.

[48] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, “Image quality assessment: from error visibility to structural similarity,” *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.