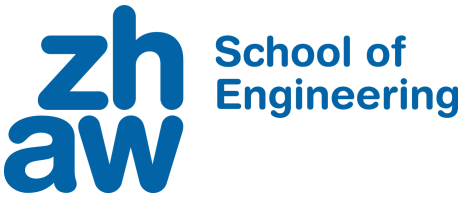


Physically-based Path Tracer using WebGPU and OpenPBR

by
Simon Stucki

supervised by
Dr. Philipp Ackermann

A thesis presented for the degree of
Master of Science in Engineering

InIT Institute of Computer Science
ZHAW Zurich University of Applied Sciences

August 2024

Abstract

This work presents a web-based, open-source path tracer for rendering physically-based 3D scenes using WebGPU and the OpenPBR surface shading model. While rasterization has been the dominant real-time rendering technique on the web since WebGL's introduction in 2011, it struggles with global illumination. This necessitates more complex techniques, often relying on pregenerated artifacts to attain the desired level of visual fidelity. Path tracing inherently addresses these limitations but at the cost of increased rendering time. Our work focuses on industrial applications where highly customizable products are common and real-time performance is not critical. We leverage WebGPU to implement path tracing on the web, integrating the OpenPBR standard for physically-based material representation. The result is a near real-time path tracer capable of rendering high-fidelity 3D scenes directly in web browsers, eliminating the need for pregenerated assets. Our implementation demonstrates the potential of WebGPU for advanced rendering techniques and opens new possibilities for web-based 3D visualization in industrial applications.

Preface

This report aims to provide comprehensive documentation of the master thesis while remaining accessible to a broad audience. The style is designed to allow readers from various backgrounds to understand the content. To achieve this, the report introduces basic concepts, offers references for further reading, and encourages more profound exploration of the subject matter. The report is fully self-contained, requiring no prior knowledge in computer graphics, with the goal to inspire readers to delve into the fascinating world of this field.
Parts of this report have been published in a short paper for WEB3D '24: The 29th International ACM Conference on 3D Web Technology [1]. This report extends the short paper by providing additional details and extended analysis.
I want to thank my supervisor Philipp Ackermann, for his guidance and support throughout the project. I also want to thank my family, friends, and co-workers for their support, time and encouragement.

Contents

1 Introduction

This section provides an overview of concrete applications and high-level procedures of 3D renderers. Furthermore, it introduces prior work conducted in this field, highlighting the novel aspects of this work. The goal is to show the potential and applicability of the developed solution. Details on the concepts and technologies involved will be introduced in chapter 2 but may already be referenced briefly during the introduction.

1.1 Use Cases

The web is a versatile platform that can be used for various applications. Most consumer devices have access to a web browser, which makes it an ideal platform for reaching a broad audience. Compared to native applications, web applications have the advantage of being platform-independent and do not require installation. This facilitates the distribution of applications and reduces the barrier of entry for users.
E-commerce is a key use case for product renderings. Commonly, these applications rely on taking pictures of the product. Like traditional physical catalogues, this approach struggles with highly configurable products due to the amount of images required to cover all possible configurations. Computer graphics addresses this challenge through product configurators for virtual assembly and visualization. They alleviate the need for physical processes, such as photography, and are scalable to a large number of configurations. This can be implemented by creating 3D models of the components and assembling them in a virtual environment.
As the number of components grows and the product evolves, the marketing models used for end-user visualization need to be coordinated with product changes. To circumvent this issue, leveraging existing production *CAD* models, prevalent in mechanical engineering and product design, offers a significant advantage. These models contain geometric and material information, which eliminates the need for redundant 3D models for marketing purposes. Geometric information defines the shape of the product, while material information refers to the surface description. One real-world example of a company with production *CAD* data is EAO. They manufacture highly customizable industrial push-buttons and operator panels. Due to the nature of the product, the number of possible assemblies grows exponentially with the number of component types.
To illustrate the use case in a simplified and universal form, consider a product assembled of n component types. Each component type (i) has o_i different options. This gives the total number of possible configurations (t) as shown in Equation 1.1.

$$t = \prod_{i=1}^{n} o_i \tag{1.1}$$

For a product consisting of n component types where each component type has the same amount of options (o), the equation can be simplified to $t = o^n$. This shows the exponential growth of possible assemblies.
One example at EAO is a specific product family within the so-called series 45. It has 14 component types (n), and each component type has approximately 10 different options (o). This results in 10^{14} possible configurations (t) based on 140 different components. In addition, EAO has many more product series and product families with varying numbers of components and options. This constitutes the primary use case considered for this thesis. The presented work uses EAO production models to demonstrate the feasibility and challenges of real-time rendering of highly configurable products. However, similar use cases can be found in other industries.
A web-based configurator using computer graphics for virtual assembly is well-suited to address such use cases where exponential growth in possible configurations is present.

1.2 CAD Data Preprocessing

To leverage production *CAD* models, the models need to be preprocessed before they can be used for 3D rendering. The steps of such a pipeline, as used by EAO to show one possible variant, are visualized in Figure 1.1.

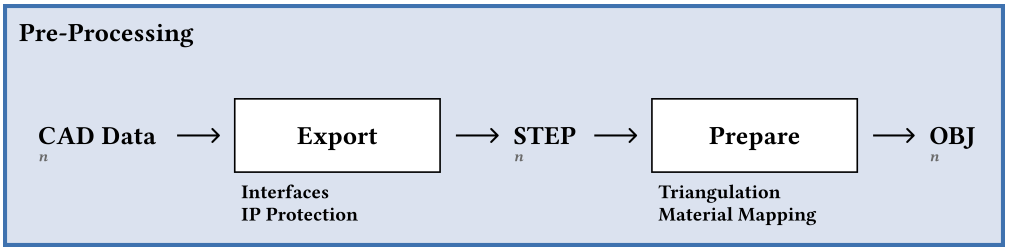


Figure 1.1: A two-step preprocessing stage is employed for offline as well as real-time rendering pipelines.

STEP[1] files are a common format for exchanging *CAD* data and can be used for this purpose. However, as will be discussed in section 2.5, other formats are more suitable for rendering and address the inherent limitations of *STEP* files. An important consideration is intellectual property rights when using *CAD* models from production processes. The models may contain proprietary information which should not be disclosed to the end user. As described by Stjepandić et al. [2], steps to circumvent this issue may include data filtering. Filtering can be implemented by removing meta information such as constraints or by removing occluded parts of the model, limiting the model to the hull of the assembly. As a positive side-effect, this potentially reduces the complexity of the model, which can be beneficial for rendering performance.

[1] *Standard for the Exchange of Product model data*, a standard formalized in ISO 10303 for product manufacturing information.

Preprocessing includes triangulation of the meshes, which is a common requirement for rendering engines. Frequently, the triangulated meshes are fine-grained and consist of a large number of triangles. This can lead to performance issues when rendering the scene. One way to handle this is to simplify the meshes by decimating triangles. Procedures for this purpose are well-established and include algorithms for generating level of detail (*LOD*) artifacts [3].
While numerous *CAD* formats include material information, they are often unsuitable for rendering because they do not describe optical properties. To address this, a material mapping can be defined, translating *CAD* materials to a suitable representation for the rendering pipeline. The concrete formats used for the preprocessing step are dependent on the *CAD* software. However, conversion tools exist for most common *CAD* formats and the general approach remains similar. After the preparation process, a rendering architecture paradigm can be employed to enable users to assemble the product virtually. In addition, a rule set for the assembly must be defined. This rule set provides data on what component types can be added to the assembly and where they can be attached. This interface definition can be provided using meta information files. Alternatively, it can also be represented geometrically within the model by using identifiable shapes or reference nodes.

1.3 Rendering Architecture Paradigms

In order to render the assemblies based on the preprocessed information, three main paradigms can be employed for web-based applications: offline rendering, client-side real-time rendering, and remote real-time rendering. These paradigms differ significantly and the technology stack varies between them. Therefore, choosing a suitable paradigm is pivotal for the implementation of the application.

Offline Rendering

Offline rendering requires pre-rendering all product configurations. This is theoretically possible for a finite number of configurations, but computationally expensive.
An offline rendering pipeline generates static images of the assembly, which are then displayed in the browser. This means that all possible assemblies need to be rendered and stored upfront. As the number of component types increases, the number of possible combinations grows exponentially, which can lead to large amounts of storage and processing power being required, as shown in Figure 1.2. Interactivity is limited by requiring all desired viewing angles to be defined upfront. 360° viewing experience using a single degree of freedom can be provided by rendering images in regular intervals. 100 images would be needed to get an image every 3.6°. These images can then be loaded in the browser as the user drags. However, if more degrees of freedom, different lighting situations, and zoom levels need to be supported, a similar growth in rendered images can be observed as for the assemblies.

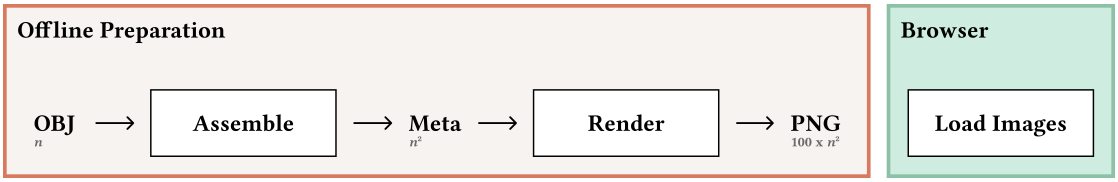


Figure 1.2: In an offline rendering setup, the number of images grows exponentially. The number of images increases further when offering 360° viewing.

In order to address latency, pre-loading images can be used. This may entail loading additional images in the background to enable a smooth user experience. However, this leads to increased bandwidth usage.

Client-Side Real-time Rendering

An alternative to offline rendering is real-time rendering. For real-time rendering, client-side rendering is a frequently used option. These approaches render the assemblies only as requested by the end user. This is the main benefit of using a real-time rendering pipeline, as illustrated in Figure 1.3. The rendering is done in the browser, limiting the server to serve the role of a file host and provide the geometry and material information in a 3D exchange format such as *glTF*[2]. This approach is more flexible and can be used for a large number of configurations. The amount of data to be stored and processed offline grows linearly with the number of components, independent of the number of possible configurations.

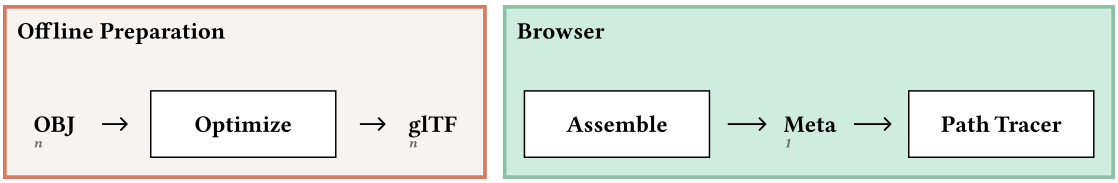


Figure 1.3: A real-time rendering pipeline solely relies on having an adequate model for each component of the assembly. It does not require pre-rendering every possible configuration.

This approach limits the technology that can be used for rendering. The rendering is done in the browser, which means that it is dependent on the browser engine as well as the specific device hardware. It restricts the scene's complexity to the device's hardware constraints.

[2] *Graphics Library Transmission Format, common 3D exchange format optimized for transmission and real-time rendering.*

Remote Real-time Rendering

For real-time rendering, an alternative paradigm is remote rendering [4], which employs a server to render the scene and stream the visualization to the browser. The main drawbacks of this approach are network latency, reliance on network stability, and operational cost for the server infrastructure, which frequently requires dedicated *GPUs*[3] for rendering.

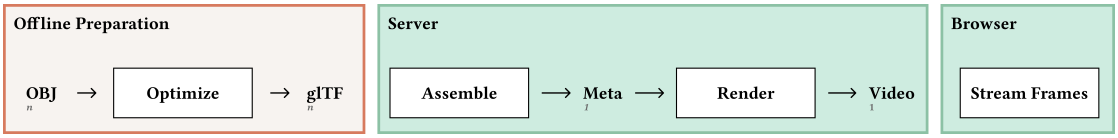


Figure 1.4: A remote rendering pipeline relies on a server to consistently stream the data in real-time to the client.

To fulfill thresholds frequently used for real-time rendering, the network latency should be below 20 ms. Recent studies have shown that this threshold is still a challenge, even when considering the fastest networks available at a given location. Many locations do not have access to any cloud data centers with a latency below 20 ms. [5]

Comparison

A set of criteria is defined to evaluate the different paradigms. These criteria are used to show the advantages and disadvantages of each paradigm. The importance of each criterion is dependent on the specific use case and can be a primary determinant for the choice of paradigm.

Preprocessing Cost

Measure how much computation effort is required to prepare the data for rendering. Generally, lower is better. Offline rendering needs to pre-render all assemblies, essentially doing exhaustive rendering. Client-side rendering and remote rendering only need basic preprocessing.

Hosting Cost

The infrastructure cost involved in hosting the application during operations. Generally, lower is better. Offline rendering does not require application servers and mainly relies on storage for the generated images. Similarly, client-side rendering needs storage for the models. These two paradigms may employ *CDNs*[4]. Remote rendering requires a rendering server. For scenarios where load is unpredictable, additional techniques such as auto-scaling and load balancing may be required. This can be costly and may induce additional complexity. In addition, bandwidth costs need to be considered.

[3] *Graphics Processing Unit*, specialized processor for parallel computation
[4] *Content Delivery Network*, distributed group of servers for caching content near end users.

Storage

The amount of data storage required for the application. Generally, lower is better. Offline rendering requires storage for all rendered images, possibly in multiple resolutions, to support different types of devices. Client-side rendering only requires storage for the models, as does remote rendering.

Main Performance Dependency

The main bottleneck in terms of rendering performance for the end user. Offline rendering sends static images, a process that is highly optimized by browser engines and image compression algorithms. Rendering such images is faster than real-time rendering, which is heavily dependent on the device hardware. Remote rendering is dependent on the network connection.

Interactivity

The level of interactivity the user can expect. Generally, higher gives more flexibility. Offline rendering means all possible views need to be pre-rendered. Extending views requires additional pre-rendering, which is optimally done using a delta change detection mechanism to prevent re-rendering unchanged views. Client-side rendering and remote rendering enable the user to interact in real-time without constraints. They also enable easier experimentation with different view configurations and open the possibility of using content provided by the end user in the rendering process. Such content could be lighting condition, background, or additional objects.

Network Dependency

The reliance on a stable network connection. Generally, lower is better. Offline rendering requires the loading of images, which is highly optimized in terms of data transmission and rendering performance. Client-side rendering requires an initial connection to download the models but is independent afterward. Remote rendering consistently requires a stable connection to stream the rendered images.

Maximum Scene Complexity Dependency

This determines the main bottleneck in terms of the complexity of the scene that can be rendered. Generally, using a server gives more flexibility as the provider has complete control over the hardware capabilities. Offline rendering can handle complex scenes given dedicated hardware. Client-side rendering is dependent on the device; the weakest supported device defines the possible scene complexity. Nowadays, rendering models with millions of triangles is possible on mid-range devices. Remote rendering can handle complex scenes, but is limited by the hardware constraints of real-time rendering, which does not cover huge scenes.

Change Management Complexity

This determines the ability to update the application based on new data. Generally, lower is better. Whenever a component changes, the application needs to be updated. Offline rendering necessitates re-rendering all images that contain the changed component. This can be a time-consuming process. Client-side rendering and remote rendering only require updating the model data and possibly meta information. This requires minimal changes in the application.

Assessment

Each of the paradigms has its own advantages. Table 1.1 highlights the strengths and weaknesses of each approach and provides a comparison of the criteria.

Rendering Paradigm	**Offline**	**Real-time**	
		Client-Side	**Remote**
Preprocessing Cost	High	Low	Low
Hosting Cost	Low	Low	High
Storage	High	Low	Low
Main Performance Dependency	network	device	network
Interactivity	Low	High	High
Network Dependency	Low	Low	High
Max Complexity Dependency	server	device	server
Change Management Complexity	High	Low	Low

Table 1.1: High-level comparison between different rendering architecture paradigms.

Offline rendering is suitable for applications where pregenerating all images is feasible and limited interactivity is acceptable. Real-time rendering, either client-side or remote, is a suitable choice for applications where the number of possible configurations is high and interactivity is important. The choice between client-side and remote rendering depends on the network infrastructure and the complexity of the scene. Generally, client-side rendering setups can be extended to support remote rendering and offline rendering, while offline rendering setups, as well as remote rendering setups, are possibly not capable of client-side rendering due to technical constraints imposed by the available technology. Based on this assessment, this work focuses on client-side rendering, with the possibility of extending it to other paradigms in the future.

1.4 Prior Work

This section highlights related work for web-based client-side real-time rendering. A rough overview of the different approaches is given, but more detailed information on concepts, technology, strengths and weaknesses, and the benefits ray tracing provides over rasterization will be discussed in section 2.3.
Rasterization is a rendering technique that projects 3D scene geometry onto a 2D plane. The technique has been widely adopted in real-time rendering due to its efficiency. However, rasterization has limitations in achieving photorealism. Effects such as shadow casting and reflection do not require additional techniques.
Ray tracing is a powerful technique for rendering photorealistic scenes, which inherently supports effects such as shadow casting and reflection. Historically, ray tracing has been used in offline rendering due to its computational complexity. However, advancements in hardware have enabled real-time ray tracing in various domains.

Web-based Real-Time Renderers

Different web-based real-time rasterizers exist. The main options include:

- *Three.js* [6] — most widely used web rendering engine.
- *Babylon.js* [7] — popular web rendering engine.
- *PlayCanvas* [8] — game engine for the web.
- *A-Frame* [9] — web framework for building virtual reality experiences.

In addition, *Unity*, a common game engine for desktop and mobile applications, also supports *WebGL*[5] [10].
These engines focus on rasterization techniques and are widely used for web-based applications. The focus of these engines lies in real-time rendering performance as used in games, advertising campaigns, virtual reality (VR), augmented reality (AR), medical imaging, scientific visualization, and educational applications. However, they lack support for ray tracing.

Web Path Tracers

Path tracing is a specific ray tracing technique. The first experiments of using *WebGL* for path tracing were implemented as early as 2010. One such example is the Google experiment demonstrating a Cornell Box [11] with basic primitive shapes such as spheres and planes [12].
Since then, various open-source implementations for the web have been developed, accompanied by related research efforts [13, 14]. While closed-source options exist, this work focuses on open-source variants. The inherent advantages of open-source software

[5] *Web Graphics Library*, since 2011 the de-facto standard API for rendering 3D graphics on the web

lie in the possibility of inspecting and modifying the code, as well as in permissive license agreements, which often allow for free use in perpetuity. Consequently, closed-source alternatives are not considered in this context.
Notable open-source path tracers for the web include:

- `three-gpu-pathtracer` [15] — One of the most widely known path tracers for the web developed by Johnson. This path tracer is implemented as a *Three.js* plugin and uses *WebGL* for rendering. It is bundled as a library and can be used to render scenes in the browser.
- `Three.js PathTracer` [16] — Path tracer based on *Three.js* developed by Loftis. This path tracer is also implemented as a *Three.js* plugin and uses *WebGL* for rendering. It does not provide a library but can serve as a basis for further development.
- `dspbr-pt` [17] — Path tracer by Dassault Systèmes implemented in *WebGL*. It implements the Dassault Systèmes Enterprise PBR Shading Model (*DSPBR*). However, the implementation has not been updated in recent years.

All of these path tracers are based on *WebGL* and are either plugins for *Three.js* or support *Three.js* scene description. A more detailed comparison of these path tracers is provided in section 4.1.

WebGPU

As of 2024, *WebGPU* is a new web *API*[6] that leverages the power of *GPUs* and serves as a successor to *WebGL*. Various applications of *WebGPU* have been investigated in recent years. Examples include `Dynamical.JS`, a framework for visualizing graphs [18]; RenderCore, a research-oriented rendering engine [19]; and demonstrations of how to use *WebGPU* for client-side data aggregation [20].
Research on the performance comparison between existing 3D engines and *WebGPU* engines has been conducted as part of the development of FusionRender, which concluded that measurable performance gains can be achieved by using *WebGPU*, but only when effectively leveraging its novel design principles [21]. Similar findings have emerged from other independent studies, showing that *WebGPU* can be faster than *WebGL* [22, 23, 24].

Conclusion

Work has been conducted in related fields; this includes research into the applicability of *WebGPU* as well as writing web-based path tracers using *WebGL*. However, the research suggests that no open-source path tracing library using *WebGPU* has been developed. In light of these findings, *WebGPU* presents a transformative opportunity. It is particularly well-suited for the development of a new real-time path tracing library for the web, which provides an alternative approach to existing rasterization-based rendering engines.

[6] *Application Programming Interface*

2 Theory

This section provides an overview of the theoretical background required to develop a path tracer. It covers the basic concepts of the underlying algorithms and data structures. Additionally, details about the technology involved, as well as a historical overview, are provided. Furthermore, data exchange formats and physical approximations for rendering are introduced.

2.1 Mathematics

This section highlights some basics about the mathematics involved in computer graphics and establishes a common understanding of concepts and notations referenced in the following sections. The section covers notation on vectors, matrices, and Bachmann-Landau. Additionally, it provides an overview of probability theory and integral calculus.

Vectors

Euclidean vectors are fundamental in computer graphics and are generally defined by a magnitude and a direction. In a three-dimensional space, a vector can be defined as $v = (x, y, z)$. This definition can be used to represent points in space (vertex) as well as directions.
The magnitude, or length, of the vector can be calculated using the Euclidean norm:

$$||v|| = \sqrt{x^2 + y^2 + z^2} \tag{2.1}$$

The cross product (vector p) of two vectors $v = (x_1, y_1, z_1)$ and $w = (x_2, y_2, z_2)$ is defined as:

$$p = v \times w = (y_1 \cdot z_2 - z_1 \cdot y_2, z_1 \cdot x_2 - x_1 \cdot z_2, x_1 \cdot y_2 - y_1 \cdot x_2) \tag{2.2}$$

As visualized in Figure 2.1, the cross product gives a vector p which is orthogonal to the two input vectors v and w.

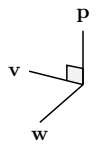


Figure 2.1: Cross product of two vectors v and w, resulting in a vector p orthogonal to v and w.

Matrices

Matrices are used to represent transformations in computer graphics. A matrix can be defined in row-major order as:

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{2.3}$$

Ray

A ray can be defined as $r = (Q, d)$, where Q is the origin vertex of the ray and d is the direction.

Triangle

A triangle can be defined as $t = (Q, u, v)$, where Q is the position of the triangle and u and v are vectors defining the triangle. See Figure 2.2 for a visual representation.

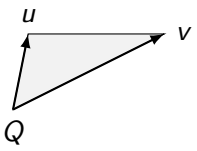


Figure 2.2: Triangle defined using three vertices, Q as the position and u,v as direction vectors starting at Q.

An alternative way to define a triangle is using three vertices, each in world space. The vertices can be defined as $v_1 = (x_1, y_1, z_1)$, $v_2 = (x_2, y_2, z_2)$ and $v_3 = (x_3, y_3, z_3)$. Converting between the two systems can be done using the following formulas:

$$Q = v_1 \tag{2.4}$$

$$u = v_2 - v_1 \tag{2.5}$$

$$v = v_3 - v_1 \tag{2.6}$$

In computer graphics, a triangle has a normal associated with each of its three vertices. Per default, the normal of a vertex is orthogonal to the two adjacent edges of the vertex and can be calculated using the cross product.

Frustum

The frustum is a geometric shape that defines the camera's view. For a visual representation, see Figure 2.3.

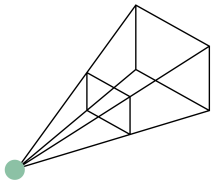


Figure 2.3: Frustum shape, the green dot indicates the position of a camera in perspective projection.

Integral

An integral determines the area under a curve, as visualized in Figure 2.4. The integral of a function $f(x)$ over an interval $[a, b]$ is defined as:

$$\int_a^b f(x)dx \tag{2.7}$$

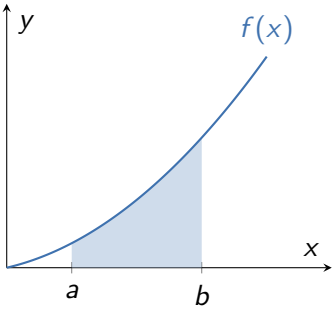


Figure 2.4: Visualization of integral, the area under the curve $f(x)$ between a and b.

In order to take an integral over a different domain S, which may represent a set of vectors representing directions, the integral can be defined as:

$$\int_S f(x)dx \tag{2.8}$$

Probability Theory

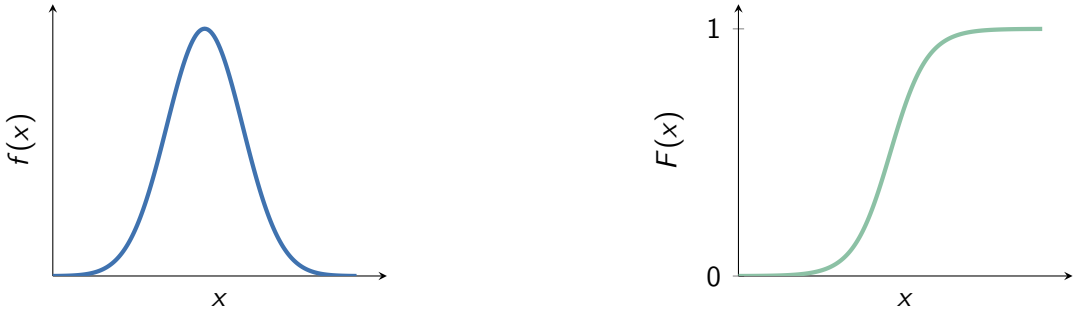
Variance is a measure of dispersion, defining how large the difference between the average and the individual values in a set of numbers is. It is calculated as the average of the squared differences from the mean. The variance of a set of numbers $x_1, x_2, ..., x_n$ is defined as:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{2.9}$$

where μ is the mean of the set of numbers.
The standard deviation (σ) is the square root of the variance (σ^2). It is a measure of the dispersion of a set of numbers and can be used to determine a confidence interval. This interval is a range of values that has a certain probability of containing the value. High variance leads to a wide confidence interval, which indicates that the data is spread out. The confidence interval (CI) as a $\pm$ margin of error deviation from the mean is then defined as:

$$CI = \mu \pm z\frac{\sigma}{\sqrt{n}} \tag{2.10}$$

where z is the z-score of the desired confidence level, for example, 1.96 for a 95% confidence interval, and n is the number of samples.
This can be used to assess the quality and reliability of measurements such as required for benchmarking.
Another important concept is the probability density function (*PDF*). It describes the likelihood of a random variable to take on a specific value. It is related to the cumulative distribution function (*CDF*), which describes the probability that the random variable will be less than or equal to a specific value. The *CDF* can be expressed as the integral of its *PDF*. See Figure 2.5 for a visualization.



(a) *PDF* visualized (b) *CDF* visualized

Figure 2.5: *PDF* and *CDF* of normal distribution visualized

Bachmann-Landau Notation

The Bachmann-Landau notation, or more specifically, the Big O notation, is used to describe the behavior of a function as the input size grows. The notation is used to describe the upper bound of a function. For example, if a function $f(n)$ is $O(n^2)$, it means that the function grows at most quadratically with the input size n. The Big O notation is used to describe the growth of algorithms in terms of run time or space requirements. See Figure 2.6 for a visualization.

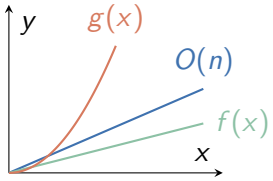


Figure 2.6: Example of Big O notation, the function $f(x)$ is $O(n)$, but $g(x)$ is not.

2.2 Physics

In order to generate images using computer graphics, understanding the physics is crucial. More specifically, optics, the study of light and its perception, is essential. The field encompasses topics such as light propagation and optical properties of matter. Maxwell's equations are a set of equations in electromagnetism that describe the behavior of electromagnetic fields. Formulated by James Clerk Maxwell in the 19th century, they provide a mathematical framework for understanding the propagation of electromagnetic waves, including, but not limited to, visible light. Light can be described as a wave or particle, generally known as wave-particle duality. [25] In a simplified model, the particles, called photons, are emitted by a light source and travel in straight lines until they hit a surface. The interaction of light with surfaces can be described as a combination of absorption, the law of reflection, and the law of refraction. Absorption is the process of light being converted into other forms of energy, such as heat.

Reflection

Reflection is the process of light bouncing off a surface. The angle of incidence equals the angle of reflection, as visualized in Figure 2.7. An example of such an effect can be seen when looking at a mirror.
This can be described using the law of reflection [25]:

$$\theta = \theta' \tag{2.11}$$

where θ is the angle of incidence and θ' is the angle of reflection.

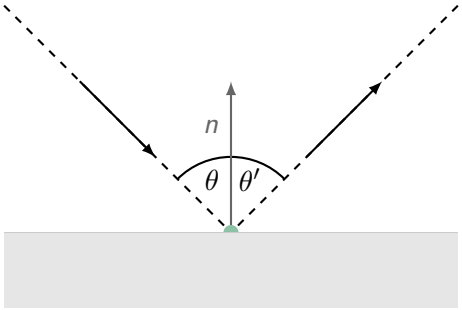


Figure 2.7: Reflection of light when hitting a surface.

Refraction

Refraction is the bending of light when passing through a medium as visualized in Figure 2.8. The distortion of objects underwater is an example of such an effect.

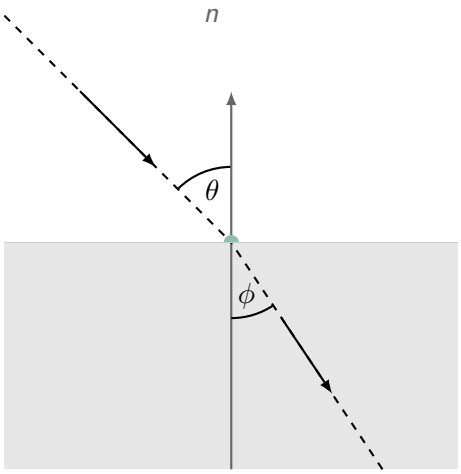


Figure 2.8: Refraction of light when passing through a medium.

It can roughly be defined by using Snell's law [25]:

$$\frac{\sin\theta}{\sin\phi} = n \tag{2.12}$$

or in a more general form:

$$n_1 \sin\theta = n_2 \sin\phi \tag{2.13}$$

where n_1 and n_2 are the refractive indices of the two media, and θ and ϕ are the angles of incidence and refraction, respectively.
Leveraging Snell's law, Fresnel equations can be derived which describe not only the refraction but also the reflection of light when passing through a medium [25].

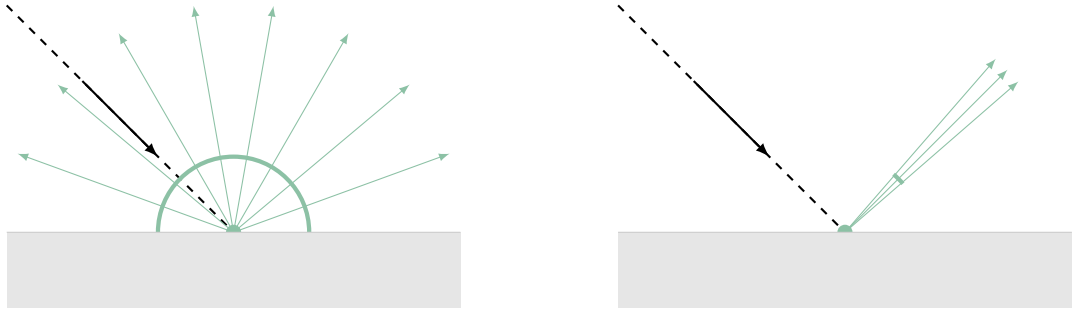
Material Properties

Common terminology used to describe the optical properties of materials is helpful in understanding the behavior of light and the challenge of simulating it. Some of the most common terms are:

- Scattering — the material scatters light. This can be observed in fog or clouds.
- Diffuse Reflection — light is scattered in all directions, independent of the viewing angle. An example of such a material is paper or chalk. As early as the 18th century, Lambert described the reflection of light on a surface [26]. Lambertian reflection is still used as a term for diffuse reflection.
- Specular Reflection — light is reflected in a specific direction. This leads to highlights on the surface.
- Roughness — defines how smooth the surface of the material is. A rough surface scatters light in many directions, while a smooth surface reflects light in a specific direction.
- Subsurface Scattering — light penetrates the material and is scattered. An example of such a material is skin or marble.
- Metallic — the material conducts electricity and typically reflects light in a specific direction, which results in a shiny appearance.
- Dielectric — materials that do not conduct electricity. These materials are not using metallic reflection and often exhibit a combination of diffuse and specular reflection.
- Absorption — defines how much of the light a material absorbs. In general, darker colors absorb more light than lighter colors. One prominent example is Vantablack, a highly absorbent material.
- Emission — the material emits light. An example of such a material is a light bulb or the sun.
- Isotropy — materials that have the same properties in all directions. An example of such a material is glass or water. Anisotropy, on the other hand, is a property of materials that have different properties in different directions. An example of such a material is wood or brushed metal. In everyday life, polarized sunglasses are an example of the practical application of anisotropy.
- Radiance — the amount of light a surface emits in a specific direction.

Generalization

Many surfaces combine these effects and need to consider the incident and outgoing directions of light, necessitating a bidirectional reflection model. This model defines where the most significant contribution to overall radiance comes from, commonly called the reflection lobe. Simplified, the main contribution can be visualized per incoming direction using an arc indicating the direction of the most significant contribution. For example, a diffuse surface scatters light in all directions (Figure 2.9a), while a mirror-like surface reflects light in a specific direction (Figure 2.9b). Reflection lobes can be combined to simulate different effects.



(a) Diffuse surface with light scattering in all directions.

(b) Mirror-like surface with a small reflection lobe.

Figure 2.9: Bidirectional reflection models for two different types of surfaces with green arc indicating the direction of the largest contribution to reflection.

2.3 Computer Graphics

This section provides an overview of computer graphics, including its history, key concepts, and applications.
Since the early days of computing, researchers have explored ways to process visual information using computers. While the field also encompasses aspects such as image processing or two-dimensional graphics, this thesis mainly focuses on three-dimensional computer graphics. Topics such as animation and geometry processing will not be covered. The main topics for this thesis are geometry representation, rendering, and shading.
A 3D scene can be defined as a collection of objects in a three-dimensional space. The basic building blocks of a 3D scene are the triangles, which form the geometry of the objects. The goal is to get this abstract representation of a scene onto a 2D screen. However, to obtain photorealistic images, the rendering needs to consider advanced lighting effects.

2.3.1 Global Illumination

Research to render advanced lighting effects using computer graphics was conducted as early as the 1960s. One of the earliest papers describing approaches to render shadow casting was written in 1968 by Appel [27]. In order to describe the phenomena, a more advanced shading model was required.
To describe these phenomena, the term global illumination was coined by Whitted in 1979 [28]. It describes a complete shading model that simulates real lighting and reflection as accurately as possible [29].
While global illumination commonly refers to a subset of effects, the term is used here to encompass a wide range of effects. Simulating real lighting includes effects such as:

- Shadow Casting — The absence of light as obstructed by other objects.
- Reflection
- Color Bleeding — Special type of reflection where a surface is colored by reflection of colored light.
- Ambient Occlusion — Special type of shadow casting, where light transport as impacted by nearby objects is considered. This is frequently encountered on objects with complex topology.
- Refraction
- Caustics - Effect produced by either reflection or refraction of light.

The importance of achieving these effects depends on the use case. For the use case of this thesis, shadows and reflections are pivotal for the realism and perceived quality of the images. Refraction and caustics were not encountered in the use case of this thesis and are, therefore, not covered in detail.
Figure 2.10 shows a visualization of the most notable effects. The image shows a floor with a white diffuse material, two walls with a red diffuse material, and a sphere with specular reflection. The sphere casts a shadow onto the floor, the walls are reflected onto the sphere, the prominent red color bleeds onto the white floor.

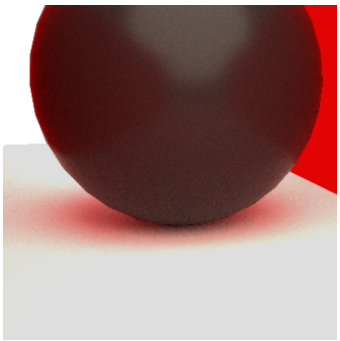


Figure 2.10: Image showing the different kinds of effects that can be achieved using global illumination.

2.3.2 Rasterization

Different rendering approaches have been developed to visualize 3D scenes using a computer. Two of the most common approaches are rasterization and ray tracing. Rasterization will be described in more detail in this section; ray tracing will be discussed in the next section.
Rasterization is a rendering technique that maps the geometry of a 3D scene onto a 2D plane. It is well-suited for real-time rendering due to its efficiency. The rasterization process can be broken down into multiple steps, commonly referred to as the graphics pipeline. The stages of the pipeline are shown in Figure 2.11.

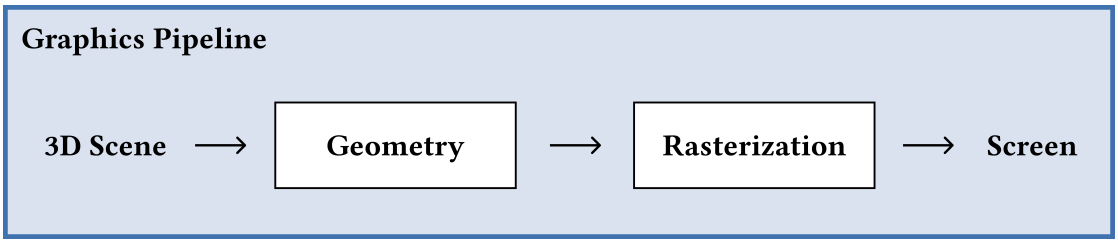


Figure 2.11: The main steps of the graphics pipeline used for rasterization.

As a pre-requisite, the 3D scene is defined. The scene consists of vertices grouped into triangles, forming the geometry. The geometry stage iterates over all triangles and uses the camera configuration to perform view projection as described in subsubsection 2.3.4, changing the position of the vertices to the camera's perspective. The rasterization stage converts the continuous geometry into discrete fragments. When having overlapping triangles, the depth buffer is used to determine which fragment is closer to the camera and, therefore, visible. The fragment processing determines the color of every pixel. Finally, the frame buffer is output to the screen.
One of the main limitations is the lack of support for global illumination. For example, shadows are generally not cast, reflections are not visible, color bleeding is not considered, and ambient occlusion is not simulated.

Advanced Techniques

To address these limitations, the graphics pipeline can be extended with additional steps such as post-processing effects or advanced rendering techniques. Various methods have been developed, including pre-baked shadow, environment [30] and light maps; screen space reflections (SSR) [31]; screen space ambient occlusion (SSAO) [32]; and screen space directional occlusion (SSDO) [33].

Environment Maps

Environment maps can be used to simulate reflections. The technique uses a texture that is mapped to the environment of the scene and is used to simulate reflections.

Environment maps not support reflections occurring dynamically in the scene, such as self-reflections or reflections of nearby objects.

Shadow Maps

Shadow maps are employed to store shadow information for a scene based on the available light sources. These shadows are pregenerated, before being employed in the rendering process. While this technique is suitable for static scenes, there are limitations in dynamic lighting situations and scenes where different components of an assembly cast shadow on one another.

Screen Space Methods

Screen space methods operate on the rendered image. Their input varies, but they frequently rely on the depth buffer, surface normals, and the rendered image. The main drawback of these methods is their limitation to the information available in the rendered image. Occlusion or out-of-view objects are not considered.

Multi-pass Rendering

Multi-pass rendering is a technique that, among other effects, can be used to simulate reflections. Instead of rendering the scene in a single pass, the scene is rendered multiple times with different configurations. For example, to simulate reflections, one first renders from the perspective of the reflective object and then renders a second pass from the camera while leveraging the first pass as a texture. When having multiple reflective objects, the technique can become computationally expensive.

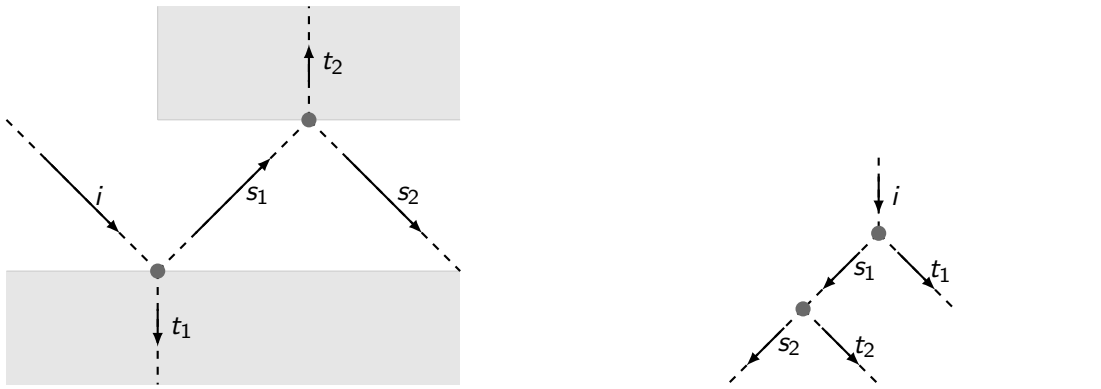
Conclusion

Each of these methods addresses only a specific limitation of rasterization. Some of them only partially alleviate the issue. These approaches induce complexity, may need to be computed at the assembly level, and can be computationally expensive. An alternative technique that resembles reality more closely could alleviate these issues: ray tracing.

2.3.3 Ray Tracing

Ray tracing is a rendering technique which simulates light transport in a scene. Forward ray tracing traces the photons, commonly referred to as rays, from the light source onto the objects. These algorithms are inefficient as most rays do not contribute to the illumination visible in the viewport. Therefore, backward ray tracing is more commonly used. Rays are cast from the camera into the scene, and the object's color with which the ray intersects is computed. In contrast to a rasterizer, which iterates over the triangles of the scene, ray tracing iterates over the pixels of the image. For each pixel, a ray is cast into the scene. By adding additional bounces of the light ray, the technique can simulate global illumination.
Early algorithms were based on recursive ray tracing [28]. These algorithms determine intersection with geometry and iteratively generate branches as shown in Figure 2.12.

For each pixel, a primary ray is cast into the scene. When intersecting with a surface, the algorithm considers the light sources and generates secondary rays. These rays are then recursively traced and radiance is calculated until a maximum depth is reached.



(a) Incident ray hitting a surface generating two branches, s_1 and t_1.

(b) Recursive ray tracing visualized as a tree structure

Figure 2.12: Recursive ray tracing as described by Whitted [28].

Due to the computational complexity of these early algorithms, ray tracing was not widely adopted until the 1980s. During these years, one of the main contributions was the introduction of the rendering equation by Kajiya in 1986 in a paper that also coined the term path tracing. Generally, path tracing can be seen as a Monte Carlo integration of the rendering equation, which is defined as:

$$I(x, x') = g(x, x')[\epsilon(x, x') + \int_S p(x, x', x'')I(x', x'')dx''] \tag{2.14}$$

where $I(x, x')$ is the radiance from point x to point x'. x for example being the camera position, and x' the intersected object. $g(x, x')$ is a geometry term determining how much light is transmitted. This depends on the distance and possibly occlusions such as transparent surfaces. $\epsilon(x, x')$ is the emitted radiance, generally used for light sources. The integral term is taken over $S = \cup S_i$ which is the union of all surfaces. $p(x, x', x'')$ is the bidirectional reflection function, which describes how light from all possible directions x'' is reflected at the surface. [34]
Further research into light transport techniques such as bidirectional light transport or Metropolis light transport has been conducted in the 1990s [35]. Concrete techniques will be described in the next section.

Monte Carlo Integration

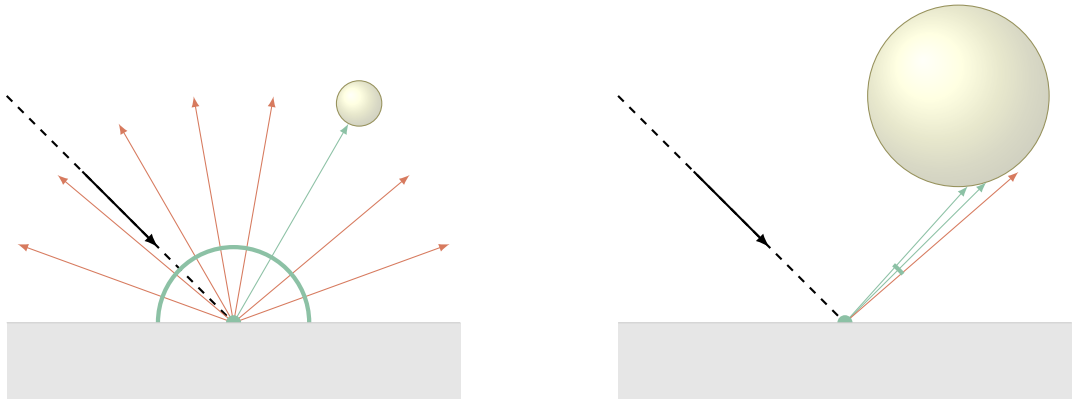
Monte Carlo integration is a statistical technique to approximate the solution of an integral numerically for cases where the integral cannot be solved analytically. At the core of the integration lies an estimator (G), which can be defined as:

$$G = \frac{1}{M} \sum_{i=1}^{M} g(X_i) \tag{2.15}$$

where G is the estimate of the solution, g is the function to be approximated, and X_i is the random sample. Taking M samples and averaging the results provides an estimate of the solution. [36]
In the context of ray tracing, a path tracing algorithm leverages Monte Carlo integration to approximate the rendering equation, as defined in Equation 2.14. The radiance of a pixel can be approximated by iteratively casting rays into the scene and averaging the results.
While Monte Carlo integration is a powerful technique, it heavily relies on the quality of the estimator that is employed. Using importance sampling can improve the rate of convergence. Importance sampling is a technique that leverages a proposal *PDF*, different from the target *PDF*, to estimate the expected value of a target function. By sampling more frequently in regions where the function has a higher value, the variance of the estimator can be reduced. In order to account for the probability distribution and prevent bias in the estimator, the estimator can be defined as:

$$G = \frac{1}{M} \sum_{i=1}^{M} \frac{g(X_i)}{p(X_i)} \tag{2.16}$$

where $p(X_i)$ is the proposal *PDF* [37].
To do the sampling, different strategies can be considered. For example, one can use the material properties to determine where to cast the rays. A highly reflective surface, such as a mirror, will apply the law of reflection, while a diffuse surface will scatter the rays in all directions. This strategy yields solid results for scenes with large light sources and reflective surfaces, as illustrated in Figure 2.13b. However, for scenes with small light sources and diffuse surfaces, the strategy can lead to high variance due to rays that miss the emissive surfaces most of the time, as shown in Figure 2.13a. The variance is visible to the human eye as noise in the image.

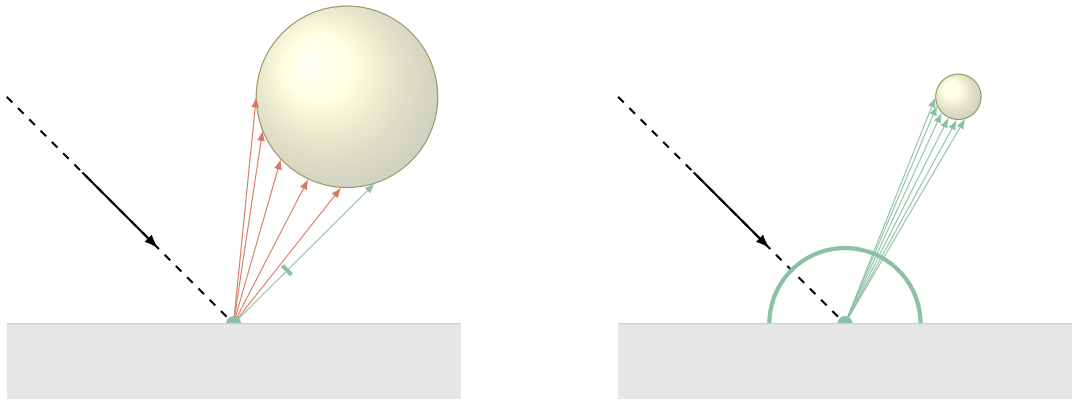


(a) Ray hitting a diffuse surface with bad-performing material sampling.

(b) Ray hitting a mirror-like surface with well-performing material sampling.

Figure 2.13: Material sampling strategy weakness and strength

Light source sampling is an alternative sampling strategy that can alleviate this issue. As shown in Figure 2.14b, it can significantly reduce the variance of the estimator when having diffuse surfaces and small light sources. Conversely, the strategy can lead to high variance when having large light sources and reflective surfaces, as shown in Figure 2.14a.



(a) Ray hitting a mirror-like surface with bad-performing light source sampling.

(b) Ray hitting a diffuse surface with well-performing light source sampling.

Figure 2.14: Light source sampling strategy weakness and strength

Both strategies are good candidates under different circumstances. To leverage the benefits of both strategies, Veach extended importance sampling in 1997 by introducing multiple importance sampling (*MIS*) [35]. The idea is to combine multiple sampling strategies to improve the overall estimator. The *MIS* estimator can be defined as:

$$G = \sum_{h=1}^{k} \frac{1}{M_h} \sum_{i=1}^{M_h} w_h(X_{h,i}) \frac{g(X_{h,i})}{p_h(X_{h,i})} \tag{2.17}$$

where w_h is the weight of the h-th sampling strategy, and $p_h(X_{h,i})$ is the probability distribution of the h-th sampling strategy. w_h is the weighting function for the estimator such that the expected value of the estimator is unbiased.

Russian Roulette

Russian roulette is a technique for improving efficiency by employing probabilistic path termination. It selectively discards paths with low expected radiance contribution. If a path contributes zero radiance to the final image, it can be terminated early without affecting the radiance of the final image. However, if the path has a small but non-zero contribution, discarding it introduces bias into the calculation. Russian roulette solves this by introducing a probability of termination which is inversely proportional to the radiance of the path. Paths with low radiance are more likely to be terminated. Paths that are continued are scaled by the inverse probability of termination. This ensures to account for the bias introduced by the termination.
Mathematically, Russian roulette is commonly formulated as shown in equation 2.18 [37].

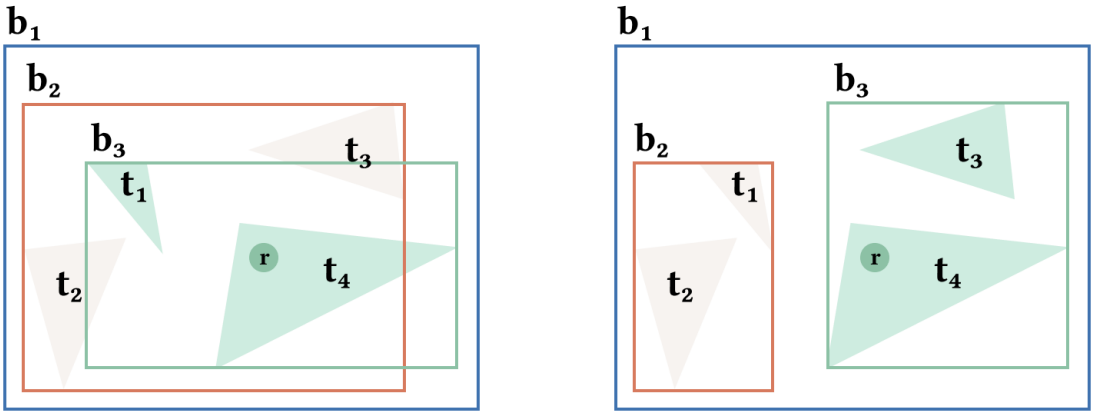
$$I' = \begin{cases} \frac{I - qc}{1-q} & \xi > q \\ c & otherwise \end{cases} \tag{2.18}$$

where I is the radiance of the path, and q is the probability of termination, which can, for instance, be calculated based on the radiance of the path. ξ is a random number between 0 and 1. c is the contribution of the path which is terminated, commonly set to zero. I' is the scaled radiance of the path.

Intersection Testing

When casting a ray into the scene, the first step in a ray tracer is to determine where the ray intersects with the geometry of the scene. A brute force approach could determine the intersection distance for each object to the ray and pick the one with the lowest distance. Given the fact, that common objects consist of thousands of triangles (n) and require tracing many rays, this approach is not efficient as it would require $O(n)$ intersection tests for each ray.
In order to accelerate the intersection testing, various approaches have been developed. The bounding volume hierarchy (BVH) is an acceleration structure leveraged to reduce the number of ray intersection tests per ray. This method is widely used in ray tracing and has been studied for a long time [38]. The core idea is to group adjacent objects in a bounding volume and structure the hierarchy in a way that all child elements of a node are contained in the bounding volume of the parent node. The nodes, employed for storing the bounding volume are often axis-aligned bounding boxes ($AABB$). This allows for early rejection of branches which are not intersected by the ray. By employing a BVH, the number of intersection tests (n) can be reduced from $O(n)$ to $O(log(n))$.
An $AABB$ (b) can be defined as $b = (min, max)$, where min and max are the minimum and maximum vertices of the bounding box. The boxes are then grouped into a hierarchy

where the maximum and minimum vertices of the children define the bounding box of the parent, as visualized in Figure 2.15. Good performance of the *BVH* depends on the quality of the hierarchy. A poorly constructed *BVH* can lead to more intersection tests, as shown in Figure 2.15a. This is because intersections need to be tested against all elements: three bounding boxes and four triangles. For a well-constructed *BVH*, the algorithm can reduce the number of intersection tests as visualized in Figure 2.15b by doing the minimum amount of intersection tests: b_1, which intersects; b_2, which can be discarded after checking; and b_3, which necessitates further intersection tests for the two contained triangles. For such a small number of triangles, the overhead of the *BVH* outweighs its benefits. However, for larger scenes, the performance gain is substantial. In the case of well-balanced trees, the number of tests can be halved at each level of the hierarchy.



(a) Poorly constructed *BVH*, which requires seven intersection tests for r, essentially testing against all objects.

(b) Well-constructed *BVH*, which reduces the required intersection tests for r to four.

Figure 2.15: Sample 2D *BVH* visualization, which contains four triangles (t_1 to t_4) and their corresponding bounding boxes b_1 as parent and b_2 as well as b_3 as the children. b_1 is visualized bigger than needed to improve readability. The color of the triangles indicates to which bounding box they belong. r indicates the ray to be tested for intersection.

After the intersection tests on the *AABB*, the actual intersection tests with the triangles contained in the bounding box need to be performed by using an algorithm such as the Möller-Trumbore algorithm [39].

Production Ray Tracing

Early ray tracing applications were primarily for research purposes or special effects in movies. Adoption expanded as computers became more powerful. One of the first widely

used ray tracing software was *POV-Ray*[1], released in 1991 [40]. Blue Moon Rendering Tools (*BMRT*) [41], a renderer compliant with *RenderMan*[2], was released in the mid-1990s and was one of the first ray tracing renderers to be used in the industry. In 2003 *RenderMan* included ray tracing capabilities [42]. Nowadays, ray tracing is widely used in industries such as film, architecture, and automotive.

Hybrid Approaches

Ray tracing and rasterization are not mutually exclusive. Hybrid approaches have been developed to leverage a multi-paradigm approach. One such example is PICA PICA, a real-time ray tracing experiment [43]. The rendering pipeline is split into multiple stages: global illumination, shadow, reflection, and direct lighting. Each stage could be implemented using different techniques, and the strategy could be chosen based on the scene's requirements and the available computational resources.

2.3.4 Common Techniques

While rasterization and ray tracing are inherently different techniques, specific tasks are similar. The following sections discuss common challenges and their differences in rasterization and ray tracing.

View Projection

In order to render an image, a virtual camera needs to be defined. Given a vertex as $p = (x, y, z)$, the view projection is responsible for transforming the vertex into 2D normalized device coordinates. The view projection matrix is a $4x4$ matrix which depends on the projection type and the camera parameters.
Different types of view projection can be used depending on the use case. The most common types are perspective and orthographic projection. Perspective projection simulates the way the human eye perceives the world, objects which are further away appear smaller. On the other hand, orthographic projection does not consider distance, and objects appear the same size regardless of their distance. Orthographic projection is frequently used in domains such as architecture.
These two projection types differ in the view projection matrix. For perspective projection, the following parameters need to be defined:

near (n) The distance from the camera to the near clipping plane.

far (f) The distance from the camera to the far clipping plane.

fov (θ) The *field of view* angle, measured in degrees.

aspect ratio (r) The ratio of the viewport's width to its height.

[1] *Persistence of Vision Ray Tracer*, cross-platform ray tracing renderer officially released in 1991
[2] A rendering software for photorealistic 3D rendering developed by Pixar

The clipping plane parameters n and f are primarily used in rasterization to determine the camera frustum. Objects outside of the frustum are not rendered. Using equations 2.19 and 2.20, the view projection matrix (P) for perspective projection is defined in equation 2.21.

$$h = 2n \cdot \tan(\frac{\theta}{2} \underbrace{\pi/180}_{\text{deg to rad}}) \tag{2.19}$$

$$w = rh \tag{2.20}$$

$$P = \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.21}$$

The elements $P_{1,1}$ and $P_{2,2}$ adjust the field of view and therefore the perspective effect. The elements $P_{3,3}$, $P_{3,4}$ and $P_{4,3}$ adjust the near and far clipping planes. In comparison to rasterization, the view projection matrix is used differently in ray tracing. In rasterization, every vertex is transformed by the view projection matrix. In ray tracing, the view projection matrix is only applied to the camera rays.
For orthographic projection, no such projection matrix is required in ray tracing. Instead, parallel rays are cast from the camera grid.

Random Number Generator

To simulate light transport, the path tracer needs to pick random directions. Generating a random direction vector, defined as $v = (x, y, z)$, can be achieved by picking three random numbers independent of one another. This is done using a random number generator (*RNG*). In rasterization, the necessity for *RNG* is less common but may still be required for specific effects. There are a variety of *RNGs* available which differ in essential characteristics. Some of the characteristics to consider when choosing a suitable *RNG* are:

- Statistical Quality – The *RNG* should generate numbers that are statistically random. This means that the numbers should be uniformly distributed and have a low correlation between them.
- Period – The length of the cycle before the sequence of generated numbers starts to repeat itself.
- Time Performance – The time it takes to generate a random number. This is influenced by the algorithm as well as the hardware, which may support dedicated instructions.
- Space Usage – The amount of memory required to store state.

Based on the use case, other factors such as prediction difficulty or others such as highlighted by O'Neill [44] may be important. Cryptography, for example, requires randomness of the *RNG* to prevent an adversary from predicting secret keys. Options to generate the randomness require additional signals [45] or may require special hardware, such as *Lavarand*[3] [46]. Many generators are pseudorandom generators. These generators are deterministic and require a seed to start the sequence. For cryptographically secure generators, the seed must be random. However, for a path tracer, there is no need for cryptographic security. Instead, it is more important to have high performance and to have a long period to avoid repetition. Therefore, a pseudorandom number generator with non-cryptographically secure seed is used.
One option for such a pseudorandom generator is using Xorshifts, such as described by Marsaglia [47]. The algorithm can be defined as shown in Figure 2.16. x is the state of the *RNG* and a, b and c are constants which are chosen to achieve good statistical properties. Frequent choices for these constants are $a = 13$, $b = 17$, and $c = 5$.

```
1  x ^= x << a;
2  x ^= x >> b;
3  x ^= x << c;
```

Figure 2.16: Xorshift *RNG* implementation.

The Mersenne Twister [48] is an alternative *RNG*. It is a pseudorandom number generator which has a long period and good statistical properties. However, it is slower than Xorshifts [44]. More recently, *PCG*[4] has been proposed as an alternative to these options. *PCG* is a family of fast generators well-suited for ray tracing use.

Anti-aliasing

Aliasing is a common issue in computer graphics. It occurs when the resolution of the screen is not sufficient to represent the scene accurately. One such example is jagged edges on diagonal lines. Anti-aliasing techniques can be employed to reduce the effect. The negative effect of aliasing is demonstrated in Figure 2.17a.

(a) Rendering showing aliasing. (b) Rendering with anti-aliasing applied.

Figure 2.17: The two images show the effect of aliasing and anti-aliasing.

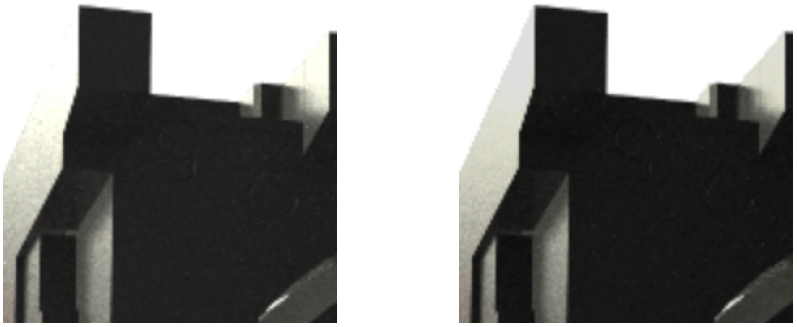
In ray tracing, aliasing can be alleviated by randomizing the direction of the rays for the different samples taken. The result of applying anti-aliasing can be seen in Figure 2.17b.

[3] Hardware *RNG* based on pictures taken of lava lamps to generate randomness
[4] *Permuted Congruential Generator*, a family of random number generators

In rasterization, similar solution strategies can be employed. Supersampling is a common technique that renders the scene at a higher resolution and then downsamples the image by averaging the colors of the pixels. More elaborate techniques, such as deep learning super sampling (DLSS) implemented by NVIDIA [49], can be used to further improve the quality of the image.

Tone Mapping

When simulating light transport, the radiance of the scene can exceed the displayable range. Tone mapping is a technique used to map the high dynamic range (*HDR*) input generated by the renderer to a lower dynamic range. The basic idea is to adjust the radiance of the scene to resemble realistic lighting conditions more closely. One common algorithm to do so is the Reinhard tone mapping operator [50]. In recent years, special tone mappers have been developed for dedicated use cases. One example is the Khronos *PBR* Neutral Tone Mapper [51], which is designed to work well with rendering true-to-life colors. See Figure 2.18 for a comparison of tone mapping with and without the Khronos *PBR* Neutral Tone Mapper.



(a) Rendering without tone mapping.

(b) Rendering using tone mapper.

Figure 2.18: Comparison of tone mapping with slight changes in luminance of specular reflection. The upper part is barely visible without tone mapping as it blends into the background due to the high luminance.

2.4 Computer Graphics Technology

This section provides an overview of the technology used to render 3D scenes. It will introduce the hardware as well as the software used in the field with a focus on technology relevant for this thesis by introducing *WebGPU* and its core concepts.

Parallel Processing

Sequential processing is limited in terms of performance. As the amount of data to process grows, the time to process the data grows linearly. Computer graphics exhibits

a large number of problems which can be parallelized. In 1966, Flynn introduced a classification system for computer architectures based on the instruction streams and data streams that can be processed in parallel [52, 53]. Already in 1970, Watkins described an algorithm for computer graphics leveraging a specialized processor for parallel processing [54]. Further research, such as Chap, a *SIMD*[5] graphics processor proposed in 1984 by Levinthal and Porter as part of Lucasfilm, introduced a special-purpose processor executing a single instruction on multiple data points [55].

GPU

The first consumer graphics processing unit (*GPU*), the GeForce 256 produced by NVIDIA [56], was developed in the 1990s and extended upon research conducted in the decades prior such as the superworkstations introduced by Silicon Graphics in the 1980s [57]. These workstations were special-purpose computers optimized for computer graphics. The *GPU* on the other hand is a specialized processor that is well-suited for workloads which require parallel processing and can be installed on a standard computer. While *CPU* parallelization on consumer hardware is generally limited to a few cores, modern *GPUs* have thousands of compute units that enable *SIMD* and, more commonly, *SIMT*[6] processing. The difference is that *SIMD* operates on a single processor, while *SIMT* operates on multiple processors.
Due to computational complexity, ray tracing techniques have been limited to offline rendering for a long time. Early ray tracers such as *BMRT* were developed, focusing on leveraging the *CPU* for computations. The introduction of *GPUs* has led to further research optimizing speed for ray tracing techniques. One such example of research for a well-established technique is *BVH* construction on the *GPU* [58]. Leveraging *GPU* has been a focus of research and has enabled real-time ray tracing in recent years. Notable developments include reservoir-based spatio-temporal importance resampling (*ReSTIR*) [59] and subsequent improvements [60, 61, 62].
Nowadays, *GPUs* are prevalent in consumer hardware such as smartphones, tablets, laptops, and desktops, and their use case is not limited to computer graphics. *GPUs* are used in various applications, such as machine learning (*ML*), scientific computing, and data processing. Devices that do not have a discrete *GPU* often use an integrated *GPU*, which is part of the *CPU*.
Modern *GPU* hardware offers support for hardware-accelerated ray tracing. Examples include NVIDIA RTX [63], Apple *Metal* on M3 [64], and others. These examples demonstrate an increasing trend towards hardware-accelerated ray tracing in recent years. However, many devices and *APIs*, including *WebGPU*, do not yet support hardware-accelerated ray tracing.

[5] *Single Instruction, Multiple Data*, type of parallel processing of data points with a single instruction as defined by Flynn [52]
[6] *Single Instruction, Multiple Threads*, extension to *SIMD* which is frequently used on modern *GPUs*

Common Strategies

Specific basic definitions have established themselves as standard practices for graphics *APIs*. The term shader describes a program that runs on the *GPU*. The term shader is frequently used in conjunction with different shader types which are used as part of the graphics pipeline in rasterization. The pipeline in its basic form consists of two steps which are executed on the *GPU* in sequence:

- Vertex Shader — The vertex shader is responsible for transforming the vertices of the geometry into screen space. The output of the vertex shader is a set of vertices which are then rasterized.
- Fragment Shader — The fragment shader is responsible for determining the color of the fragments that are generated by the rasterizer. It is executed once for each fragment.

In addition to this standard graphics pipeline, many *APIs* also support custom code which does not need to be used exclusively for rendering. This code can be used for general-purpose computing and is generally referred to as *GPGPU*. The corresponding compute pipeline consists of a single stage where input and output can be configured. Shaders that are used for *GPGPU* are often called compute shaders.

Parallelization Architecture

GPUs are designed to process data in parallel using *SIMT* architecture. A single instruction is executed on multiple threads, each operating on independent data. The multiple threads are grouped into so-called warps at the hardware level. As only a single instruction is executed per warp, the threads must be synchronized. If one thread in a warp takes a different path, e.g., due to conditional branching, all threads in the warp must wait until the divergent thread completes. This divergence in control flow can lead to performance degradation as demonstrated by Laine et al. [65]. The performance can be significantly impacted for complex material workflows with highly divergent control flow. One option to address this issue is to use split the shaders from a single monolithic shader, known as a megakernel, into multiple smaller kernels. However, this approach can lead to additional overhead due to the increased coordination efforts, manifested as additional global memory and synchronization mechanisms.

OpenGL

OpenGL is an *API* for rendering 3D graphics. After its introduction in 1992, it was widely adopted. Subsequently, the standard has been ported to other platforms and has been extended with new features. The *API* has its own shading language called OpenGL Shading Language (*GLSL*).
To date, *OpenGL* is still widely used in the industry, but it has been replaced by more modern *APIs* in recent years.

Modern Graphics APIs

While *OpenGL* and derivatives such as *OpenGL ES*[7] have been widely adopted, the introduction of a number of new *APIs* has changed the landscape. The standards share an understanding of giving developers more control over the hardware and can be more efficient than *OpenGL*. Modern hardware-accelerated features such as ray tracing are more easily accessible using these new *APIs*. Some of the most notable *APIs* are:

- *Vulkan* — Developed by the *Khronos Group*[8], *Vulkan* is a low-level *API* that is supported on a wide range of platforms.
- *Metal* — Developed by Apple, *Metal* is a low-level *API* that is supported on Apple devices.
- *DirectX 12* — Developed by Microsoft, DirectX is a collection of *APIs* for Microsoft Windows.

None of these *APIs* are supported in the browser, which makes them unsuitable for the discussed use case.

WebGL

WebGL is a graphics *API* for the web based on *OpenGL ES* 2.0. It was initially released in 2011 and has since been adopted by all major browsers.
WebGL is designed to offer a rendering pipeline but does not offer *GPGPU* capabilities. There have been efforts to extend with compute shaders, but efforts by the *Khronos Group* have been halted in favor of focusing on *WebGPU* instead.
Workarounds have been developed to provide *GPGPU* capabilities. The basic idea is to render the output of a fragment shader to a texture and interpret the output as binary data instead of color information. There are libraries such as `GPU.js` which provide *GPGPU* capabilities using *WebGL*. `Tensorflow.js`, a library for training and deploying *ML* models, uses similar techniques in the *WebGL* backend.
Since its introduction, *WebGL* has been extended with new features. *WebGL* 2.0, based on *OpenGL ES* 3.0, was released in 2017. Of the major browsers, Safari was the last to support it out of the box in 2021. Since *WebGL* 2.0, no major versions have been released, but new features have been added. This makes it evident that while *WebGL* will remain a viable option for the foreseeable future, *WebGPU* is likely to become the preferred choice in the long term.

[7] *OpenGL for Embedded Systems*, subset of OpenGL designed for embedded systems like smartphones
[8] consortium for developing interoperability standards for 3D graphics

WebGPU

WebGPU is a new web standard developed by *W3C*[9] [66]. Unlike its predecessor, *WebGL*, it is no longer based on *OpenGL*. One of the main capabilities is support for *GPGPU* by design. While all major browser vendors have announced intent to support *WebGPU*, to date only Chrome has shipped *WebGPU* for general use on desktop as well as mobile. The standard is still in development and new features are being added. Common 3D engines such as *Babylon.js*, *Three.js*, *PlayCanvas* and *Unity* have announced support for *WebGPU*.

Implementations

Three major implementations of *WebGPU* exist:

- *Dawn* — Developed by Google, *Dawn* is a C++ library which provides a *WebGPU* implementation. It is used in Chromium-based browsers [67].
- *wgpu* — A Rust library which provides a *WebGPU* implementation. It is used in Firefox and *Deno*[10] [68].
- WebKit *WebGPU* — Developed by Apple, WebKit is used in Safari [69].

As the standard is under active development, there are differences between the implementations as well as the specification [70]. However, the implementations are largely compatible thanks to alignment efforts and the extensive conformance test suite [71]. Generally, these implementations run natively on modern graphics *APIs* such as *Vulkan*, *Metal*, and *DirectX 12*. Some also support *OpenGL* and *WebGL* as a fallback.

Core Concepts

WebGPU is a low-level *API* when compared to *WebGL*. Users get access to the *GPU* via adapters, which abstract the underlying hardware. *WebGPU* fully supports common *GPU* strategies as described in paragraph 2.4. Most of the commands are then executed on a logical device which is created using the adapter.
For a compute pipeline, one needs to define a pipeline layout that consists of multiple bind group layouts. These define the memory layout of the data that is passed to the *GPU*. The code to be executed is defined in a shader module. The compute pipeline is then created by combining the shader module and the pipeline layout. See Figure 2.19 for a visual representation of the core concepts in *WebGPU*. Setting up a graphics pipeline is similar, but two shader module configurations are required.

[9] *World Wide Web Consortium*, a non-profit organization dedicated to developing web standards
[10] JavaScript runtime, alternative to Node.js

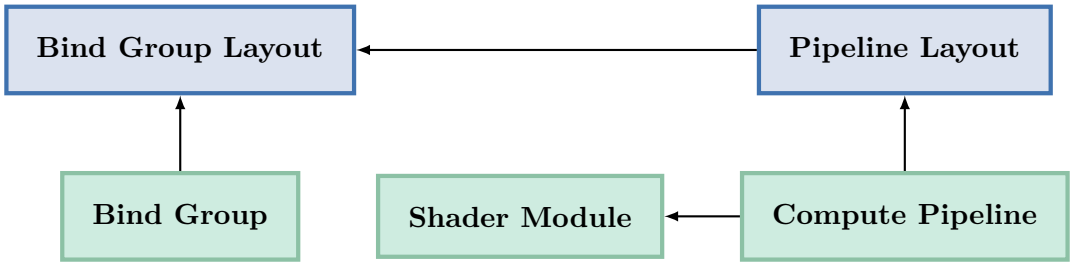


Figure 2.19: Core concepts in *WebGPU*, the arrow direction indicates references to. Blue elements are independent of concrete references to data and code, while green elements are concrete references.

Having set up the compute pipeline, one needs to define the bind groups which contain the data. Using a command encoder, the compute pipeline can be dispatched to the *GPU* for execution. The shader describes the expected size of the workgroup, which is a group of threads executed in parallel. The dispatch coordinates how many workgroups are executed. The total number of threads is, therefore, given by the product of the workgroup size and the dispatch size. For a visual representation of a workgroup of size 3x3 and dispatch size 4x4, refer to Figure 2.20.

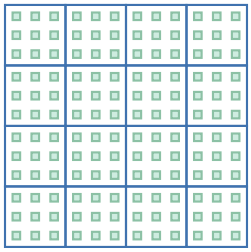


Figure 2.20: *GPU* workgroup layout. Each green square represents a thread within a workgroup. Each blue square represents a workgroup.

Shading Language

WebGPU introduces a new shading language called *WebGPU* Shading Language (*WGSL*) [72]. The language is statically typed and supports a variety of data structures required for computer graphics and general-purpose computing.
Every compute pipeline, or render pipeline, requires an entry point, as shown in Figure 2.21. This entry point can then be defined as shown in Figure 2.22. Note that the return type definition is optional for functions.

```
device.createComputePipeline({
  layout: computePipelineLayout,
  compute: {
    module: computeShaderModule,
    entryPoint: "computeMain",
  },
});
```

Figure 2.21: JavaScript code defining the entry point for a compute pipeline.

```
@compute
@workgroup_size(1, 1, 1)
fn computeMain(@builtin(global_invocation_id) global_id: vec3<u32>) {
  // ...
}
```

Figure 2.22: *WGSL* code defining the entry point for a compute pipeline.

Attributes such as `@compute` are used to provide additional information to the compiler. In this case, it denotes the entry point for the compute pipeline and is mandatory as it enables usage of specific attributes such as `@builtin`, which may only be used on entry points. In terms of limitations, it is similar to other shading languages. For example, it does not support recursion because cycles are not permitted in declarations. It also does not have features such as an *API* for *RNG*. The shading language is designed for common use cases in computer graphics and general-purpose computing. One example is support for swizzling. Swizzling is a class of operations that facilitate managing vector elements. For example, given a vector `let v = vec3f(x, y, z)`, the operation `v.xy` returns a vector `(x, y)`.

Data

Data needs to be transferred between the *CPU* and the *GPU* within bind groups. The core methods to store data are textures and buffers. Textures are optimized for n-dimensional data such as images and offer additional features such as interpolation. Buffers are the primary method to store arbitrary data. Depending on the type of data to be transferred, different buffers can be used:

- Uniforms — Uniform Buffers are optimized for data which is read-only on the *GPU* and is the same for all vertices or fragments. Such information could be the projection matrix.
- Storage — Storage Buffers are intended for read-write access on the *GPU*. The size limits of these buffers are higher than for uniform buffers.
- Vertex — Alternative to storage buffers, intended for vertex data.

- Index — Buffers which contain indices for the vertices.

Memory Alignment

WGSL supports `struct` for defining data structures. However, memory alignment must be considered when passing data between the *CPU* and the *GPU*. Alignment is a constraint, restricting the memory address at which a data structure can be stored. Having strict memory alignment enables the use of more efficient hardware instruction sets or addresses hardware requirements.
Each type in *WGSL* has specific alignment requirements independent of the size. For example, a `vec3f` has a size of 12 bytes but requires an alignment of 16 bytes. So for a struct definition such as can be seen in Figure 2.23, the struct requires 32 bytes of memory to store 20 bytes of data as illustrated in Figure 2.24. The excess 12 bytes are used for padding.

```
1  struct ThreeFields {
2    a: f32,
3    b: vec3f,
4    c: f32
5  }
```

Figure 2.23: *WGSL* `struct` containing three fields of type `f32` and `vec3f`.

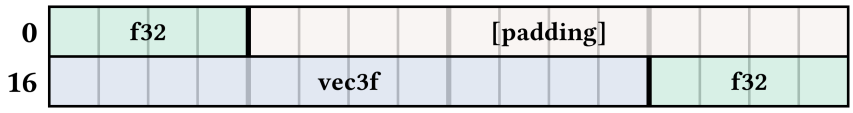


Figure 2.24: Memory alignment with padding for struct defined in Figure 2.23.

Memory alignment is opaque while writing *WGSL* code, but it is relevant when providing data to the *GPU*. The alignment requirements need to be considered when creating buffers. When using JavaScript to create buffers, the standard web *APIs* such as `Float32Array` or `Uint32Array` can be used to provide data. In order to write data to the `struct` in Figure 2.23, the views can be created as shown in Figure 2.25.

```
1  const ValueBuffer = new ArrayBuffer(32);
2  const a = new Float32Array(ValueBuffer, 0, 1);
3  const b = new Float32Array(ValueBuffer, 16, 3);
4  const c = new Float32Array(ValueBuffer, 28, 1);
```

Figure 2.25: JavaScript code providing views to access the `struct`.

2.5 Exchange Formats

In order to exchange 3D scenes between a multitude of applications, various standardized formats have been developed. These formats are optimized for different use cases depending on the requirements of the application. For this thesis, the focus is on formats well-suited for real-time rendering on the web. In comparison to other use cases, certain aspects are critical on the web. These aspects include:

- Efficiency – It should be optimized for transmission and loading. While data compression algorithms such as *Gzip*[11] can alleviate some of the issues during transmission, pure text-based formats are generally less efficient than binary formats.
- Feature Set – The format should support a wide range of features such as geometry, materials, and scene graphs.
- Extensibility — The format should be extensible to support additional features.
- Interoperability – 3D engines and applications should widely support the format.
- Openness — The format should be open and not proprietary. Additionally, it should be actively maintained.

The following section categorizes the formats based on their use case and highlights well-established formats.

General Purpose Formats

One of the most widely used formats is Wavefront *OBJ*, which was developed in the 1980s. The format is text-based and has basic support for materials and textures via a companion *MTL*[12] file. However, it lacks support for more advanced features such as animations. Additionally, due to its encoding, it is not well-suited for delivery over the web, compared to more modern alternatives.
Formats such as the proprietary *FBX*[13] by Autodesk, established in 2006, can address the shortcomings in terms of feature set. A wide range of applications supports the format. However, one of the main disadvantages is the proprietary nature of the format, which can make it challenging to integrate in applications which do not have access to the *SDK*[14].

[11] *GNU zip*, file format and software application for compression and decompression.
[12] *Material Template Library*, a companion file format to *OBJ* for material definitions
[13] *Filmbox*, proprietary 3D exchange format nowadays managed by Autodesk
[14] *Software Development Kit*

Specialized Formats

For specific use cases, specialized formats have been developed. One such example is *STEP*. The format is widely used in product manufacturing and offers support for a wide range of modeling features. The *ASCII*[15] encoding of the format is human-readable but is not well-suited for real-time rendering requirements of the web due to transmission size and parsing complexity [73]. Parsing complexity is exhibited by the large number of different entities which can be defined in the format.
Another such format is *STL*[16] which is widely used in 3D printing and supports binary and *ASCII* encoding. It lacks support for advanced features such as material representation, animations, and scene graphs.

Interoperability Formats

The aforementioned formats have shortcomings in interoperability and feature set for complex 3D scenes, such as those found in the film industry. Special formats have been developed to address issues encountered in these scenes.
Other formats such as *COLLADA*[17] have been developed to improve transporting 3D scenes between different applications. Based on *XML*[18], the format was established in 2004 and has been managed by the *Khronos Group* since then. Its release was in 2008.
One notable format is *USD*[19], which has been open-sourced by Pixar in 2016. Similar to *COLLADA*, the format is designed with interoperability in mind and is widely supported by 3D engines and applications. The main focus of *USD* is support for complex scenes and complex collaboration workflows, which are common in the film industry but are less relevant for the use case in this thesis.

Runtime Formats

For usage in end-user applications, such as for the use cases in this thesis, the *glTF* format is well-suited [74]. It was established in 2015 as an open standard and is designed to be efficient for the transmission and loading of 3D scenes. The standard is developed by the *Khronos Group* and is widely supported by 3D engines and applications. The format supports geometry, materials, animations, scene graphs, and other features.
Technically, the format offers two different encoding formats which differ in file ending:

- `.glb` – Binary format which is optimized for transmission and loading.
- `.gltf` – Text-based, human-readable format.

[15] *American Standard Code for Information Interchange*, character encoding standard for electronic communication
[16] *Stereolithography*, 3D exchange format developed by 3D Systems
[17] *Collaborative Design Activity*, 3D exchange format managed by *Khronos Group*
[18] *Extensible Markup Language*
[19] *Universal Scene Description*, 3D exchange format

glTF is designed to be extensible and a variety of extensions have been developed to support features such as mesh compression using Draco. The hierarchy structure of a *glTF* file is shown in Figure 2.26. Many of the official extensions extend the material nodes to support more advanced features. Its focus on transmission and loading efficiency makes it well-suited for the web.

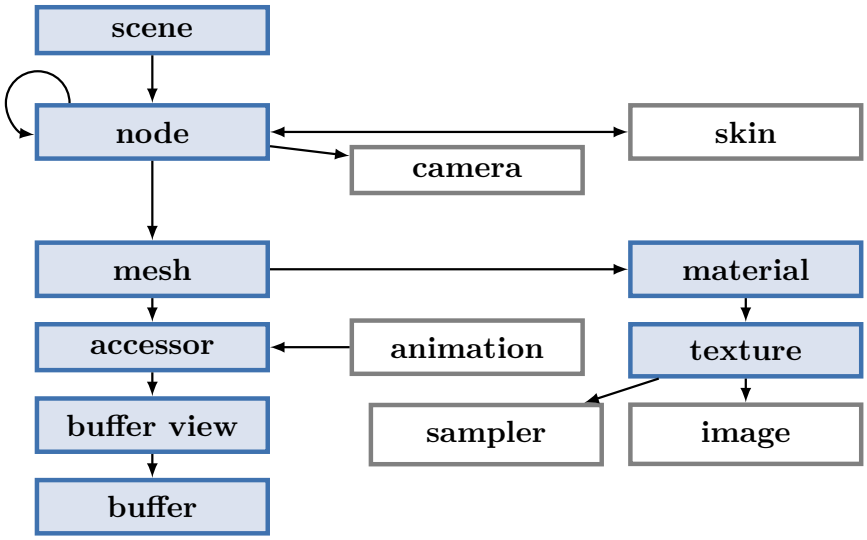


Figure 2.26: *glTF* hierarchy starting with the `scene` object. Arrows indicate parent-child relationships. Gray objects are not mentioned in this thesis.

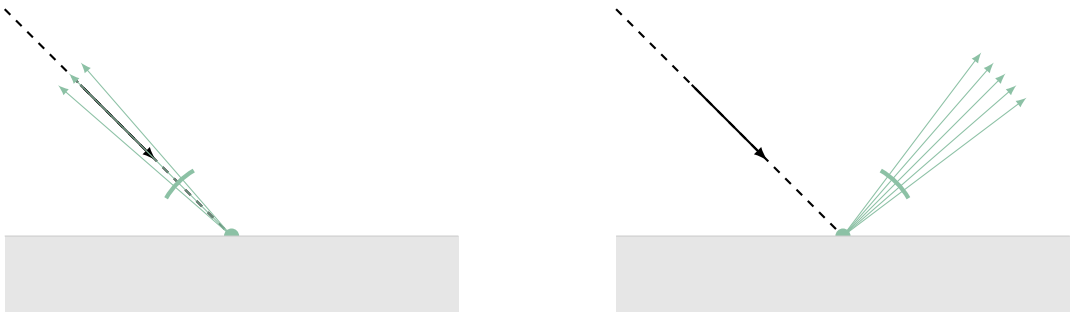
2.6 Physically-based Rendering

Defining the geometry is one part of the equation. Another major part is defining materials. Materials determine how a surface interacts with light. The core idea of physically based rendering (*PBR*) is to simulate the interaction of light using physical models. Instead of fine-tuning parameters for a specific look and feel and having to adjust based on the desired lighting conditions, *PBR* aims to define a material that behaves consistently in different lighting situations. It is, therefore, an approach based on physical principles such as those described in section 2.2.
Research into *PBR* has been conducted for a similar amount of time as ray tracing. Moreover, just like ray tracing, the computational complexity of the algorithms has limited the adoption of *PBR* in real-time rendering for a long time. *PBR* has generally coincided with the rise of ray tracing as it is a natural fit for the physics-based approach to light transport. One prominent example of a *PBR* system is *pbrt*, an open-source renderer with an associated literate programming book that has won an Academy Award for formalizing and providing a reference implementation for *PBR* [37].
PBR and ray tracing are not mutually dependent, as *PBR* can also be used in rasterization. In contrast to how ray tracing inherently addresses shortcomings of rasterization by leveraging an entirely different approach, *PBR* is a more incremental improvement

extending existing shading techniques with more sophisticated models. Many ray tracers use simple non-standardized *PBR* models that are suitable to demonstrate basic effects of global illumination but are unsuitable for high-fidelity rendering of complex scenes. Not having a clear standard makes alignment between different tools hard to manage and can lead to inconsistencies in the final rendering.
In general, there is no *PBR* standard. Instead, a variety of different models have been developed over time. In recent years, however, a convergence towards a common set of principles has been observed and certain aspects of *PBR* are found in most models. This section explains these common principles and standards which can be used to obtain consistent renderings.

BxDF

The definition of the bidirectional distribution functions (*BxDF*) is a key concept in *PBR*. The x stands for the different kinds of distributions to consider. In general, the functions describe how light is reflected or transmitted. The two functions are usually abbreviated as *BRDF*, the r for reflectance, and *BTDF*, the t for transmittance. The combination of the two is often referred to as *BSDF*, the s for scattering [37].
These functions define the distribution of light in different directions, as described in subsubsection 2.2. Samples of different kinds of *BRDFs* can be seen in Figure 2.27.



(a) Retroreflective reflection, as can be encountered in materials such as velvet.

(b) Glossy reflection, as can be encountered in materials such as plastic.

Figure 2.27: Special types of reflectance distributions using the same visualization as Figure 2.9 [37].

The *BRDF* (f_r) describes how much incident light along a differential cone of directions (ω_i) is scattered from a point (p) into a direction (ω_o) and can be defined as:

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i)\cos(\theta_i)d\omega_i} \tag{2.22}$$

where d references to the differential nature, L_o is the radiance leaving the surface in the outgoing direction, L_i is the radiance arriving at the surface from the incident direction, and $\cos(\theta_i)$ is the cosine of the angle between the normal and the incident direction that

reduces the intensity of the light for grazing angles. See Figure 2.28 for a visualization of the variables.

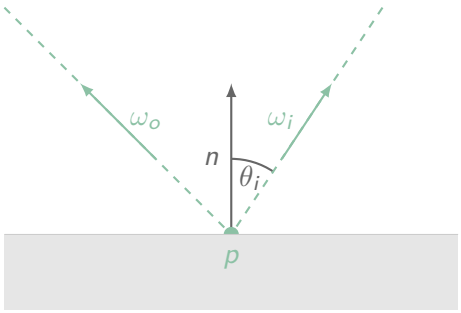


Figure 2.28: Variables of *BRDF* visualized for a sample ω_o and ω_i at p [37].

Many objects exhibit a combination of different reflectance distributions. To obtain the radiance (L_o), the *BRDF* can be integrated over the hemisphere of incoming directions. When also considering the *BTDF*, the integral to determine the combined *BSDF* (f) can be defined as:

$$L_o(p, \omega_o) = \int_S f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos(\theta_i)| d\omega_i \tag{2.23}$$

where S is the sphere of all incoming directions. The integral can be approximated using Monte Carlo integration. [37]

Microfacet Theory

While the macroscopic appearance of a surface is visible to the human eye, the microscopic structure of the surface is not. However, the microscopic structure is crucial in determining how light interacts with the surface. Microfacet theory is a model which describes this microscopic structure. Microsurfaces are aggregated to form the coarse macroscopic surface. See Figure 2.29 for a visualization of a rough surface. Theoretically, it would be possible to model the geometry as microsurfaces, but in practice, this leads to a high computational complexity due to increased storage and computation requirements. Instead, the distribution of normals is described as a normal distribution function (*NDF*). A rough surface has a wide distribution of normals, which leads to scattering, resulting in a diffuse reflection. A smooth surface has a narrow distribution of normals, which leads to a specular reflection. One of the most widely used *NDFs* is the *GGX*[20] distribution function. [37]

[20] *Ground Glass Unknown*, frequently used microfacet distribution function derived by Neyret and independently by Walter et al. [75, 76]

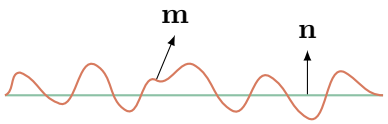


Figure 2.29: Microsurface (red) compared to macrosurface (green) and normals m for microsurface sample and n as surface normal of the underlying triangle plane. While all n on the plane are identical, the microsurface normals vary significantly. Visualized as per Walter et al. [75].

Material Description

While the principles of *PBR* are well-established, the material description in 3D formats is not uniform. A variety of different approaches have been developed. In addition to many exchange formats, such as *glTF*, having their own material definition, different shading models exist independent of these formats. Similar to exchange formats, a variety of different use cases are addressed by material description models. While some shading models are optimized for real-time rendering, others are optimized for film production.

Shader Design

Based on the focus of the material description models, they differ in how they need to be implemented. Flexible node-based systems permit the creation of custom shaders. This gives the user much flexibility, as arbitrary shaders can be created. However, this flexibility comes at a cost. The underlying implementation needs to convert the node graph into shader code that can be executed on the *GPU*. The flexibility increases the surface area of the *API* and, therefore, increases the complexity of implementation and optimization. In addition, managing a large number of custom shaders can be challenging as many shaders have similar features.
An alternative approach is to use an *uber shader*. The term does not have a strict definition, but generally, it is used to describe a shader program that can be adjusted using parameters to achieve different looks. The shader program is fixed and does not allow for customization of the shader graph. This approach often leads to a more complex shader with many parameters that can be difficult to manage, but it is easier to optimize as the shader program is fixed.
Technically, a node-based system also supports an *uber shader* approach, but an *uber shader* does not support a node-based system.

Material Description Standards

Many exchange formats, such as *glTF*, support material description as part of the standard. The material description in *glTF* supports a *PBR* approach, based on the metallic-roughness material model [74]. However, the *PBR* material description in *glTF* is limited in terms of feature set, by lacking features such as subsurface scattering, and is not intended to become an open standard across a variety of applications.

Over the years, different parties have developed their own shading models. Some prominent options include:

- Disney Principled *BRDF* [77] – Developed by Disney in 2012, the model is not necessarily physically accurate and focuses on being art-directable with limited parameters. Although the model is influential, it is not an established standard and lacks a reference implementation despite being adopted by some applications, including *RenderMan* [78].
- Autodesk Standard Surface [79] – Developed by Autodesk, the goals are similar to Disney Principled *BRDF* in that the parameter set is kept small in order to be easier to use. The model is developed as an open standard with a reference implementation in *OSL*[21]. However, the standard has not seen much development in recent years.
- Adobe Standard Material [80] – Developed by Adobe, the model is also influenced by the Disney Principled *BRDF* and Autodesk Standard Surface. It has a full specification and has been adopted by Adobe for its applications.
- *DSPBR* [81] – Developed by Dassault Systèmes, it is also influenced by the Disney Principled *BRDF*. The specification is detailed and the model is designed for industrial use cases.

In addition to these standards, many real-time rendering engines, such as *Unity* and *Unreal*[22], and modeling software, such as *Blender*[23] and *Rhino3D*[24], have their own shading models. Supporting many different shading models in a web-based real-time rendering application can be challenging, making the need for a common material description standard apparent. This is where new material description standards come into play. These are open standards developed by multiple companies to address the siloed development of prior shading models intended to be used across different applications. *MaterialX* is an open standard which was originally developed by Lucasfilm in 2012. The standard has since been adopted by the *ASWF*[25] as an open standard with support from a variety of companies. While not strictly limited to *PBR*, the standard provides a wide range of features to describe physically-based materials [82]. *MaterialX* supports extensive shader generation capabilities. This permits to re-use shaders across different applications and frameworks and aids in understanding the used approximations.

[21] *Open Shading Language*, shading language for production global illumination renderers maintained by Academy Software Foundation
[22] Unreal Engine, cross-platform game engine developed by Epic Games
[23] open-source 3D computer graphics software toolset
[24] *Rhinoceros*, proprietary 3D computer graphics and *CAD* toolset
[25] *Academy Software Foundation*, organization promoting development and adoption of open-source software in the motion picture industry

OpenPBR

OpenPBR is a surface shading model hosted by the *ASWF* as an open standard [83]. It differs from *MaterialX* in providing an *uber shader* approach instead of a node-based one. Version 1.0 of the standard was released in June of 2024 [84].
It combines Autodesk Standard Surface and Adobe Standard Material and is being worked on by both companies. The potential of this standard is to provide a common shading model which can be used across different applications. Support for the standard has already been announced in applications by Autodesk, Adobe, and NVIDIA [85], among others. *Blender* has reworked their Principled *BSDF* shader to be based on *OpenPBR* [86]. These promising signs indicate that the standard has the potential to be widely adopted.
The *OpenPBR* specification includes a reference implementation in *MaterialX*. This enables the use of shader generation of *MaterialX* in order to get a reference implementation in *GLSL* or *OSL*.
The *uber shader* approach is defined by having a fix set of inputs that can be adjusted, but it does not allow for custom node graphs as *MaterialX* does. *OpenPBR* is designed for photorealistic material description. The first release focuses on surface shading and does not support specialized workflows such as advanced hair or volumetric effects like fog, smoke, and dust.
The specification is focused on physical aspects and does not specify implementation details. Implementations have an unambiguous target in terms of how it should look like, but have flexibility in terms of the degree of fidelity. While this makes alignment between different applications harder, it also allows for flexibility in the implementation of the standard, which is especially important for real-time rendering.

Formalism

One core tenet of *OpenPBR* is energy conservation. This means that the *BSDF* must not reflect more light than it receives. The structures consists of multiple layers to describe the material as visualized in Figure 2.30. The layers are:

- Base — Substrate of the material, which can be a mixture of metal or dielectric.
- Coat — An optional layer of dielectric material.
- Fuzz — An optional layer representing reflection from micro-fibers.

The layers are set up to be adjacent mediums where the *BSDFs* of the individual interfaces are combined.
The substrate is defined as a mixture of metal, translucent base, subsurface, and the glossy-diffuse layer. These so-called slabs are combined using a mix operation. The operation is defined as a weighted sum of the different layers. The weights are defined by the user and can be adjusted to achieve different looks.

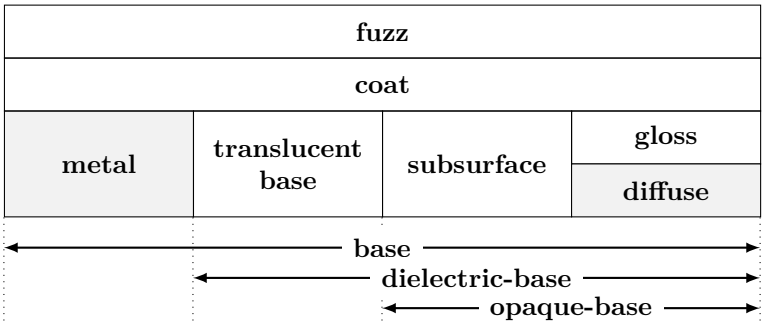


Figure 2.30: The different layers of the *OpenPBR* surface shading model. The workflow supports various features such as metallic, glossy, and diffuse reflection renderings that are important for industrial use cases.

Assessment

While there is a large variety of material description standards, *OpenPBR* is a promising candidate. The focus on surface shading is well-suited for the use case of this thesis. In comparison to other standards, it has significant industry backing, being adopted by major parties in the computer graphics industry such as Adobe, Autodesk and NVIDIA. Its design principles such as focusing on physical aspects, not specifying implementation details, and the *uber shader* approach make it well-suited for real-time rendering.

3 Results

As discussed in the introduction and theory chapters, the goal of this work is to implement a web-based path tracer. The path tracer is designed to be used for product visualization based on production *CAD* data, leveraging the *OpenPBR* surface shading model and *WebGPU*.
The concrete result of this work consists of multiple parts. The core is a path tracing library called `strahl`. The code as well as documentation is published under the *MIT license*[1] on GitHub with an accompanying website. See `https://www.github.com/StuckiSimon/strahl` for details. This also includes detailed instructions on how to set up and configure the renderer. The library is published on the *npm*[2] registry as `strahl`.
This report serves as the primary documentation of the work but does not contain library usage documentation. In addition, a dedicated short paper has been published for WEB3D '24: The 29th International ACM Conference on 3D Web Technology [1]. The short paper includes the main insights and results of this work.
This section focuses on the implementation details and highlights design decisions of the path tracer. It also provides an overview of the available documentation, insights into the performance of the renderer, and its applicability for the described use case.
A consistent notation is used for references to the implementation of the path tracer. For example, *CODE#ABC* refers to the code with the same comment. All relevant places are marked in code using the same reference. This enables the use of search features to find the specific code section.

3.1 Implementation

The goal of the implementation is to be compatible with a large variety of devices and to make the setup for consumers straightforward. Therefore, it is implemented and tested mainly in Chrome, which uses *Dawn* as the underlying implementation of *WebGPU*. Most notably, this means that dedicated features from other implementations, such as *wgpu*, cannot be used. Neither can experimental extensions be leveraged.
Figure 3.1 illustrates the procedure of the path tracer. Scene preparation is performed on the *CPU*. This setup needs to be done once per visualization. Subsequent scene sampling is carried out repeatedly on the *GPU*, which constitutes the most computationally intensive part of the process.

[1] Permissive license originating at the Massachusetts Institute of Technology (MIT)
[2] package manager for JavaScript

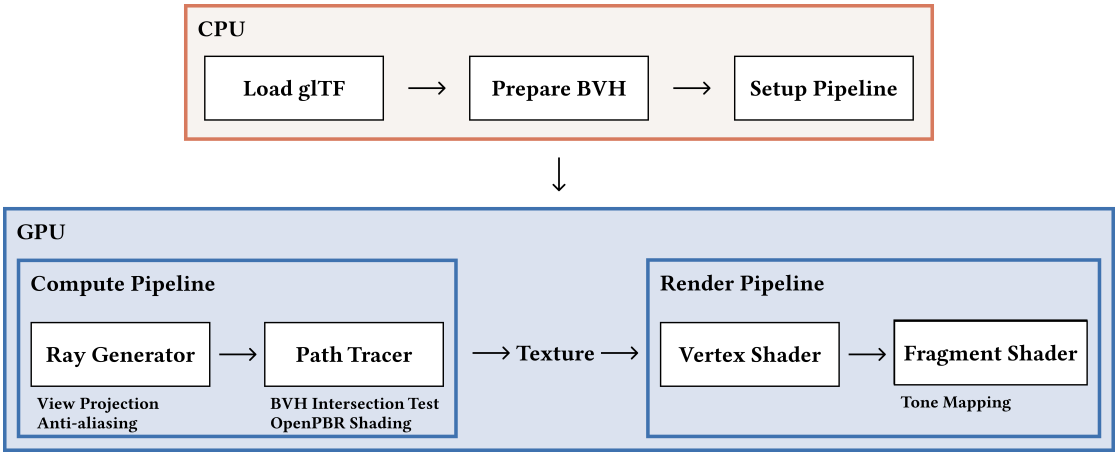


Figure 3.1: Path tracer pipeline, distinguishing *GPU* and *CPU* tasks. The stages of the compute pipeline and render pipeline are executed sequentially.

Scene Description

In order to be easy to integrate for developers familiar with existing web-based rendering engines, the renderer utilizes many of the scene description constructs provided by *Three.js*. This includes the representation of geometry using `BufferGeometry`, camera using `PerspectiveCamera`, and arbitrary camera controls such as `OrbitControls`.
This also enables the use of a variety of loaders for different file formats, such as *OBJ* or *glTF*. However, due to its advantages for transmission, it's advised to use *glTF*, which can be imported using the loader provided by *Three.js*.

Scene Preparation

The primary process involved with setting up the scene is the preparation of the *BVH*. For *BVH* construction, well-established solutions for the web are available. The path tracer uses `three-mesh-bvh` [87]. This method builds the *BVH* on the *CPU*; the code for transferring the *BVH* to the *GPU* is in *CODE#BVH-TRANSFER*.
In order to address memory alignment, as described in subsubsection 2.4, the path tracer uses `webgpu-utils` [88]. The library enables a straightforward way to map data to buffers and align them correctly. See *CODE#MEMORY-VIEW* for the creation of the type definition and *CODE#BUFFER-MAPPING* for the mapping of data to buffers.
A high-level view of the data structures is shown in Figure 3.2. Using the *BVH* it is possible to obtain the indices via indirect indices. Indirect indices are used because the *BVH* re-orders the indices of the triangles, but the original indices are needed to get the correct object definition which stores the material information of the surface. See *CODE#BUFFER-BINDINGS* for definition. *WebGPU* enforces limits on the maximum number of storage buffers that can be employed per compute pipeline. Therefore, buffers for geometry data, which contain position and normal information per vertex, are interleaved in order to reduce the amount of storage buffers. Interleaving is the practice

of putting multiple data types into the same buffer. The following sections will discuss more details on the specific buffer contents.

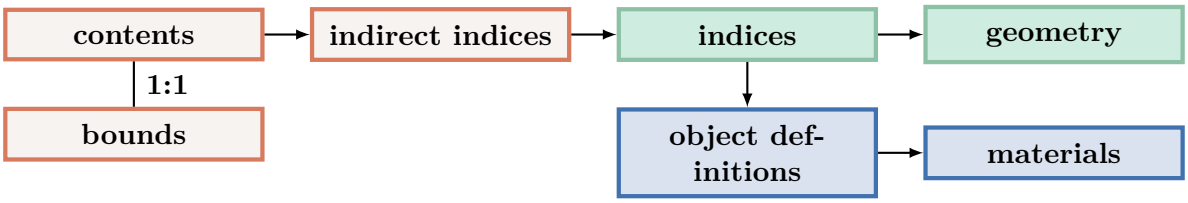
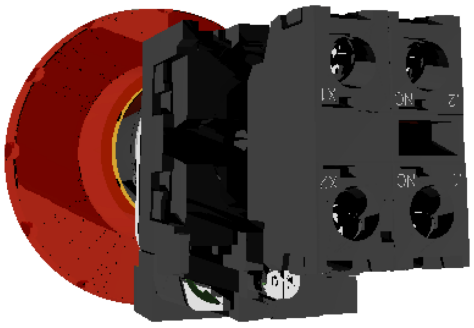


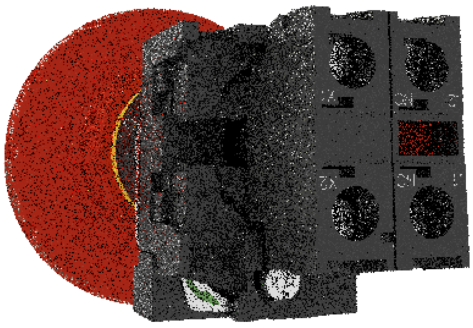
Figure 3.2: Main data buffers used by the path tracer. Data structures associated with the *BVH* are red, geometry information is green, and material or object information is blue. Arrows indicate the pointers between the buffers.

Ray Generator

The ray generator is responsible for casting rays into the scene according to the view projection. The path tracer employs a backward ray tracing approach, tracing rays from the camera into the scene.
For many applications, especially photorealistic rendering, perspective projection is prevalent. Therefore, based on the assessed use case, the path tracer uses perspective projection. See *CODE#VIEWPROJECTION* for implementation.
For *RNG*, the path tracer uses *PCG*, specifically the `PCG-RXS-M-XS` variant, as described by O'Neill [44]. See *CODE#RNG* for implementation.
In order to set up the Monte Carlo method, the *RNG* needs to be employed in a suitable manner. As a pseudorandom generator, it necessitates a seed to start the cycle. If the seed is identical for all pixels, the results of a single sample will frequently share similar patterns in adjacent surfaces, as shown in Figure 3.3a. The result for independent seeds differs in a pronounced manner as demonstrated in Figure 3.3b.



(a) Identical seed for all pixels.



(b) Independent seed for every pixel.

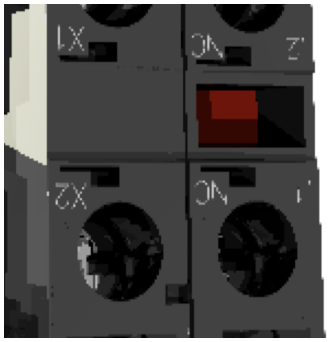
Figure 3.3: Both images consist of only one sample.

When increasing the sample count, the differences in the setup remain visible. Adjacent surfaces show similar patterns, as shown in Figure 3.4a. These patterns resemble image

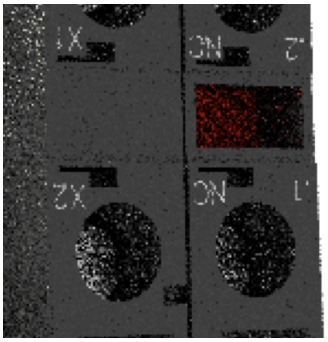
compression artifacts encountered in aggressively compressed *JPEGs*[3]. In contrast, the renderings with independent seeds show stark differences in adjacent pixels akin to noise as shown in Figure 3.4b. As shown in Figure 3.4c compared to Figure 3.4d, the anti-aliasing is less noticeable when using independent seeds.

(a) Renderings which resemble lossy image compression artifacts.

(b) Noisy renderings with stark differences in adjacent pixels.



(c) On the right side of the image, anti-aliasing is rather noticeable.



(d) On the right side of the image, anti-aliasing is less noticeable.

Figure 3.4: Magnified images of renderings with low sample count showing difference based on seed setup. The left column has identical seed for all pixels of a sample, but varying seeds for different samples. The right column has independent seeds for every pixel of a sample as well as across samples.

The implementation of the anti-aliasing strategy indicated in 2.3.4 is implemented in *CODE#ALIASING* and is highly dependent on the *RNG* setup.

[3] *Joint Photographic Experts Group*, common method for lossy image compression

Path Tracer

The path tracer is the core of the library and is responsible to test for intersections, sample the scene, and calculate the final radiance. The basic procedure can be seen in Figure 3.5. The ray, cast by the ray generator, is tested for intersections. If it misses, the path will be terminated. If a surface hit is detected, the shading will be generated based on the *OpenPBR* specification. The shading generates a new scattered ray and determines the throughput of the ray. Russian roulette uses the throughput to perform probabilistic path termination. If it should be continued, the ray is cast again. The max depth of the ray determines the end of the path. During termination, the radiance contribution is collected in *RGB*[4] color space.

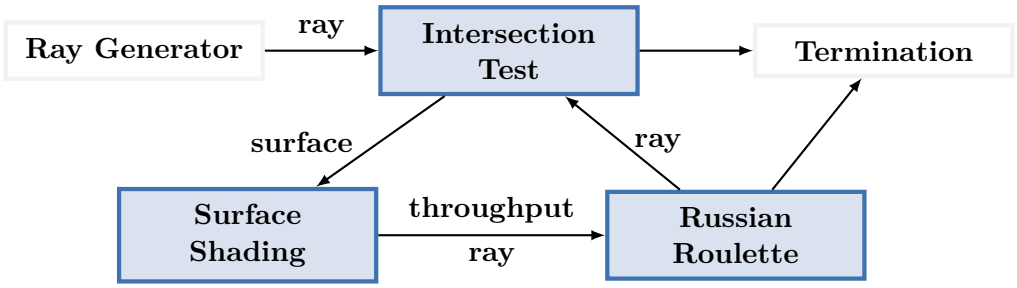


Figure 3.5: High-level workflow of path tracing core routine. Blue are the main parts of the recursive path tracing algorithm, gray are the pre- and post-processing steps. The arrows indicate the flow of data, describing the primary information passed between the steps.

Intersection Test

Intersection tests based on the *BVH* are implemented in *WGSL*, see *CODE#BVH-TESTS* for details. The intersection tests for triangles are implemented in *CODE#TRIANGLE-INTERSECTION*. The boundary data is stored as *AABB* requiring two vectors for the minimum and maximum bounds. The *BVH* node content depends on whether it is a leaf or an inner node. Each node uses 64 bytes of memory, as visualized in Figure 3.6. The split axis determines which child node is traversed first. The left index is the index of the sibling node. Therefore, only the right index is stored. For leaf nodes, the triangle offset determines the index in the indirect indices buffer. The triangle count is the number of triangles in the leaf node.

[4] *red, green, and blue*, common system for color representation in computer graphics

(a) *BVH* inner node information

(b) *BVH* leaf node information

Figure 3.6: Content information stored in *BVH* nodes. The information in the first two bytes is used to discriminate between inner and leaf nodes.

Surface Shading

For each intersection, sampling is done according to the *MIS* scheme using a combination of light source sampling and *BSDF* sampling as described in subsubsection 2.3.3. For light source sampling, an additional shadow ray is cast to determine the visibility of the light source. See *CODE#RAY-COLOR* for implementation. The surface shading method is based on *OpenPBR* reference implementation in *MaterialX* and the reference viewer by Portsmouth [89]. Shading starts by sampling the reflection functions based on definitions of *OpenPBR*, which reference the Monte Carlo sampling scheme described in the *pbrt* book [37] for sampling *BSDFs*, as described in section 2.6. See *CODE#REFLECTION-LOBE-WEIGHTS* and subsequent methods for implementation. Using these probabilities, one specific reflection lobe is sampled, and an outgoing direction is determined as implemented in *CODE#BSDF-SAMPLE*. These different lobes correspond to different workflows, such as diffuse, specular, or metal. The implementation does not constitute a complete conformance to the *OpenPBR* specification but instead covers a subset of the features.
The outgoing path and radiance, determined by the surface shading, are used for Russian roulette. At this stage, it is either terminated or continued based on the throughput. Once the path is terminated, the radiance is written to a texture. A ping-pong technique is used for writing and reading. This means that the output of the previous frame is used as the input for the current frame and vice versa.

Render Pipeline

The output of the path tracing compute shader is a texture, which is then passed to a rasterizer. The rasterizer renders the texture to the canvas using a fullscreen quad consisting of two triangles. Tone mapping is done using the Khronos *PBR* Neutral Tone Mapper described in subsubsection 2.3.4. See *CODE#TONE-MAPPER* for implementation. Progressive rendering is a technique that renders an image in multiple passes. Each render pass improves the quality of the image. Per default, a pass consists of a single sample per pixel. This enables the user to see the rough image quickly while it gets refined over time.

Denoise Pipeline

As rendering progresses, the image becomes increasingly less noisy. However, as the number of samples increases, the noise reduction achieved by each additional sample diminishes. At this stage, denoising can significantly enhance image quality by further reducing noise. The renderer allows for an optional denoising step to be performed after sampling is complete. The renderer supports two mutually exclusive denoising modes:

- Denoise filtering — A filter to reduce noise based on a circular Gaussian kernel [90]. The denoise filter is integrated into the render pipeline. See *CODE#DENOISE-FILTERING* for implementation.
- *OIDN* — Open Image Denoise [91], developed by Intel, is a state-of-the-art denoising library that uses *ML* and is built for noise reduction in ray tracing. The web library is developed by Shen [92] and based on *WebGPU*, making it well-suited for integration into the path tracer as a subsequent step. After denoising, the image is rendered using the render pipeline.

Results of simple renderings with and without denoising can be seen in Figure 3.7. Note that *OIDN* can effectively denoise images even with a low number of samples, as shown in Figure 3.7d. The difference in denoising on 100 samples, as shown in Figure 3.7c for *OIDN* and Figure 3.7b for the denoise filter, is less pronounced in local areas. However, *OIDN* is more effective in reducing noise in global regions.

(a) No denoising applied, 100 samples.

(b) Denoise filtering applied, 100 samples.

(c) *OIDN* applied, 100 samples.

(d) *OIDN* applied, 10 samples.

Figure 3.7: Renderings with a limited number of samples demonstrating the effect of denoising.

3.2 Documentation

The path tracer is designed to be integrated into existing web projects. The package is installable via *npm*, but could also be downloaded and included manually. The complete documentation is available on the website. The website is designed to demonstrate the path tracer, provide a quick start guide, and offer detailed information on how to set up `strahl`. The website is shown in Figure 3.8.

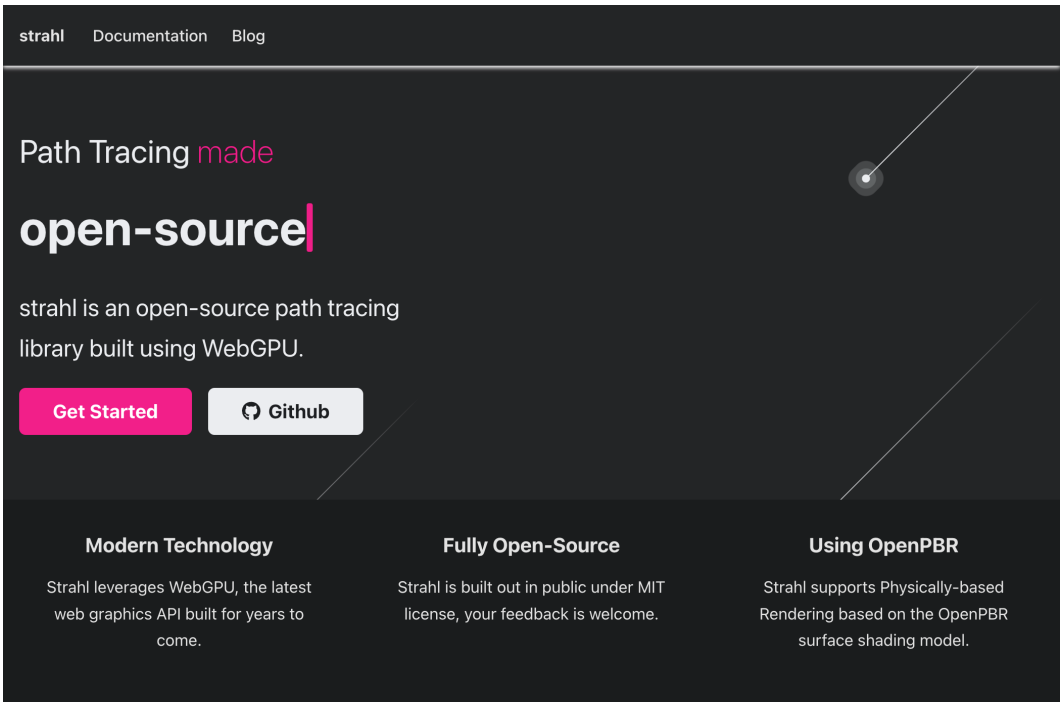


Figure 3.8: Homepage of the `strahl` website.

Documentation on how to configure the renderer is provided. The library uses custom exceptions that the user can catch and handle. It provides different exceptions to enable the user to react appropriately to different error conditions. This includes basic exceptions, where the action is limited, such as missing browser support for *WebGPU*, as well as transparent information on what the user misconfigured. The use of *TypeScript*[5] enables code completion and type checking. The documentation describes how to control sampling, denoising, environment lighting, and more. See Figure 3.9 for an example.

[5]Typed superset of JavaScript developed by Microsoft

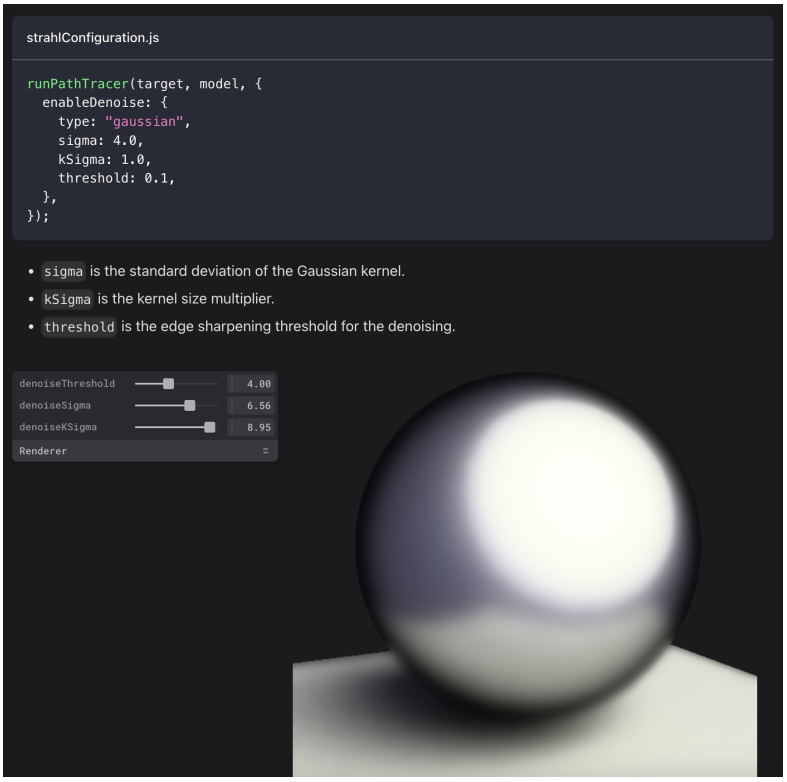
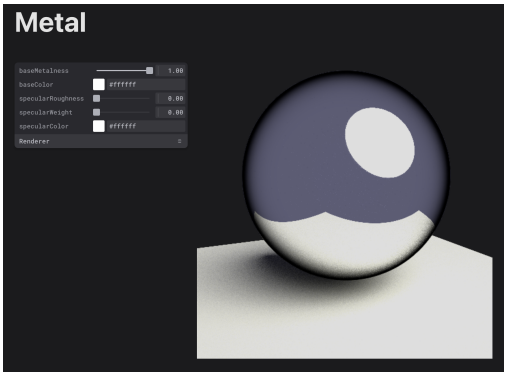
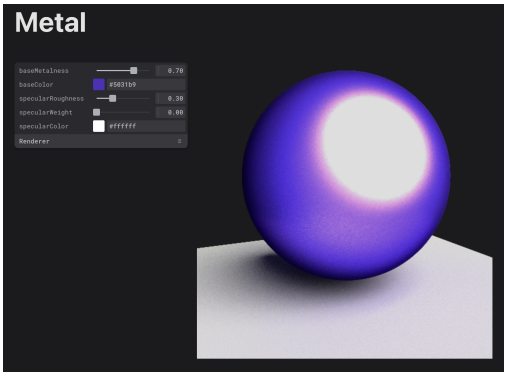


Figure 3.9: Part of documentation on how to setup denoising, with interactive example.

The documentation includes showcases for the *OpenPBR* surface shading model. The goal is to enable users to understand the parameter set and how to configure them to get the desired result. The parameters can be adjusted in real-time to see the effect on the rendering as shown in Figure 3.10.



(a) Mirror-like metal surface



(b) Rougher metal surface

Figure 3.10: Configurable showcase for *OpenPBR* parameters.

3.3 Benchmark

In order to assess the effectiveness of specific measures, a benchmark is defined to measure the performance of the path tracer. This benchmark is used for quantitative evaluation of the path tracer. Prior sections, such as anti-aliasing as described in Figure 3.1 focused on qualitative aspects of the renderer. Depending on the use case, the benchmark can be adjusted to focus on different aspects. However, the core design of the benchmark remains the same. Generally, measurements are taken for the entirety of a stage instead of focusing on individual routines within a stage. This gives a more holistic view of the performance of the path tracer. The results for *CPU* performance were recorded using the Performance *API* and exclude time for loading the model. The results for *GPU* performance were recorded using two measurement systems. The first is based on *WebGPU* timestamp queries that only account for the compute pipeline. The second is taking wall-clock time using the Performance *API* for the entire *GPU* part. The results presented are based on wall-clock time.
A total of 100 samples per pixel with a ray depth of five was used. The image was rendered in Chrome 126/127 at a resolution of 512×512 pixels. Experiments were conducted with different model complexities. The simplified versions are decimated meshes of the original, which consists of roughly one million triangles. The *LOD* artifacts are shown in Figure 3.11. The first two levels are intended to be visually similar, while the third level is a simplified version intended to demonstrate the effect of non-manifold geometry for ray tracing.
Unless otherwise specified, the benchmarks are conducted on a MacBook Pro with Apple Silicon M1 Max. The measurements were calculated using the mean time of 30 runs with a confidence interval of 95% given as ± standard deviation from the mean time in milliseconds (ms) as described in subsubsection 2.1. The different measurement sections should not be compared directly, as different factors influence them and may have been taken on different development stages of the renderer. They are only intended for comparison within the same measurement category.

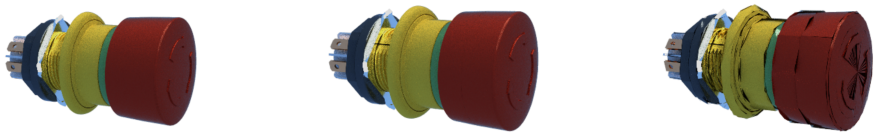


Figure 3.11: The three *LOD* artifacts and their short names in brackets, from left to right: 1,068,735 triangles (high), 106,873 triangles (mid), and 10,687 triangles (low). The left and middle figures share similar visual fidelity characteristics.

Russian Roulette

As described in subsubsection 2.3.3, Russian roulette is a technique for probabilistic path termination. The code is implemented in *CODE#RUSSIAN-ROULETTE*. The benchmark results for the path tracing routine are shown in Table 3.1.

	With Russian roulette	Without Russian roulette
High	2,361.77 ms ± 11.27 ms	2,583.40 ms ± 12.84 ms
Mid	2,018.03 ms ± 8.76 ms	2,135.94 ms ± 8.14 ms
Low	2,028.04 ms ± 10.78 ms	2,200.00 ms ± 8.91 ms

Table 3.1: Benchmark results for Russian roulette optimization.

BVH Split Axis Heuristic

For the intersection tests using the *BVH*, the split axis is used as a heuristic for node traversal, as described in subsubsection 3.1. In order to assess the effectiveness of the heuristic, the benchmark results for the path tracing routine are shown in Table 3.2.

	With split axis heuristic	Without split axis heuristic
High	3,244.30 ms ± 45.71 ms	4,039.87 ms ± 33.51 ms
Mid	2,663.14 ms ± 35.57 ms	3,181.89 ms ± 24.92 ms
Low	2,698.65 ms ± 27.60 ms	3,124.27 ms ± 36.02 ms

Table 3.2: Benchmark results for *BVH* split axis heuristic.

Overall Performance

A final performance test was conducted on two machines. The AMD/NVIDIA machine is a desktop computer with an AMD Ryzen 5 5600X *CPU* and NVIDIA GeForce RTX 3080 (10 GB) *GPU*. The *CPU* results to setup the *BVH* are presented in Table 3.3. The *GPU* results for the core path tracing are presented in Table 3.4. Note that the number of samples required for high-quality renderings is dependent on the scene. Effects such as rough metallic reflection require more samples to converge than a diffuse surface. However, for many use cases, the number is in the range of 100 to 1000 samples per pixel, with an additional denoising step.

	Apple M1 Max	AMD/NVIDIA
High	327.57 ms ± 0.56 ms	383.10 ms ± 4.06 ms
Mid	45.49 ms ± 0.31 ms	41.77 ms ± 0.43 ms
Low	10.53 ms ± 0.24 ms	8.43 ms ± 0.22 ms

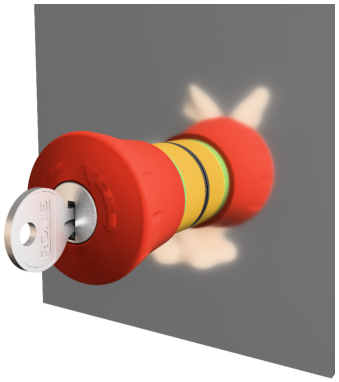
Table 3.3: *BVH* setup time based on model complexity

	Apple M1 Max	AMD/NVIDIA
High	2,319.45 ms ± 11.14 ms	1,058.73 ms ± 59.47 ms
Mid	1,992.11 ms ± 8.49 ms	790.20 ms ± 3.79 ms
Low	2,031.38 ms ± 9.87 ms	790.83 ms ± 4.18 ms

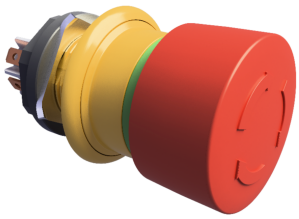
Table 3.4: *GPU* path tracer time based on model complexity

3.4 Use Case Scenarios

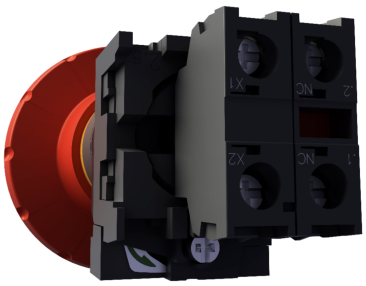
Different assemblies of production *CAD* data used by EAO as rendered by the path tracer can be seen in Figure 3.12. Note that artistic liberties were taken to highlight specific effects. The path tracer can render full assemblies without being impeded by the complexity of the geometry. The renderer offers flexibility by providing configuration of *OpenPBR* parameters, environment lighting, denoising setup, and more. This enables the user to obtain high-fidelity renderings of *CAD* data without the need for pregenerated artifacts, making the renderer suitable for product visualization.



(a) Pushbutton mounted on a metallic surface, including the reflection of the Stanford bunny model [93] positioned in the scene.



(b) Front view of pushbutton with specular highlights.



(c) Rear view exhibiting slight color bleeding, visible as a red tint on the black surface.

(d) Rear view demonstrating ambient occlusion.

Figure 3.12: Path-traced renderings of pushbutton *CAD* models.

4 Discussion and Outlook

This section discusses the findings and experience gained by implementing a project using *WebGPU*. In addition, a comparison of the results is provided and possible future work is outlined.
The path tracer focuses on the given use case and is, therefore, not a general-purpose rendering engine. For example, it does not offer a physics engine, support for animations, or other features that are common in general-purpose rendering engines, such as *Three.js*, *Babylon.js*, or alternatives. Generally, the ray tracing technique is slower than rasterization-based approaches and is therefore not a silver bullet for all use cases. The results indicate that the developed path tracer is a viable option in the discussed use case of e-commerce.

4.1 Comparison

In order to contextualize the results, the presented work is compared to existing solutions. This includes a comparison to other open-source path tracers for the web, as well as a comparison to two alternative rendering strategies: web-based client-side real-time rendering using rasterization and offline ray-traced rendering.

Comparison to Prior Work

The three open-source path tracers introduced in section 1.4 are alternatives to the renderer developed in this work. Comparing them based on quantitative measures such as *FPS*[1] is not meaningful as the number of samples required for high-fidelity renderings differs across the renderers. Nevertheless, defining a set of criteria to compare them is still possible.

WebGPU Support

WebGPU is a new technology and its support is likely to become more important in the future. The three alternatives currently do not support *WebGPU* and still rely on *WebGL*.

PBR Standard

As discussed in paragraph 2.6, a variety of standards exist for *PBR*. While *OpenPBR* is a new standard, it has already seen adoption by the industry and offers interoper-

[1] *Frames per second*, measure of the rate at which consecutive images are rendered and displayed

ability. `three-gpu-pathtracer` and `Three.js PathTracer` currently use custom *PBR* standards or partially support *glTF PBR* extensions. `dspbr-pt` uses the *DSPBR* standard.

Documentation

For developers to use the library, the renderer should be well-documented. `strahl` and `three-gpu-pathtracer` provide documentation. `Three.js PathTracer` and `dspbr-pt` provide minimal documentation. None of the three alternatives provide a material editor.

Benchmark Setup

A benchmark setup is important to assess the performance of the renderer. `strahl` provides a reproducible benchmark setup. None of the alternatives provide a comparable setup or measurements. This further complicates the comparison of the renderers.

Maintenance

The availability of an *npm* package can simplify the integration of the renderer into existing projects. Additionally, it enables updating libraries using a standardized process. `strahl`, `three-gpu-pathtracer`, and `dspbr-pt` provide an *npm* package, while `Three.js PathTracer` does not. `strahl`, `three-gpu-pathtracer`, and `Three.js PathTracer` have been updated in 2024, while `dspbr-pt` has not been updated since 2022. Therefore, maintenance for `strahl` and `three-gpu-pathtracer` is provided while `Three.js PathTracer` and `dspbr-pt` are lacking in this regard.

Assessment

Table 4.1 contains a high-level comparison between the four renderers. The renderer developed in this work is the only one that supports *WebGPU* and uses the *OpenPBR* standard. The extensive documentation and benchmark setup further distinguish the developed solution. Therefore, `strahl` provides an alternative to the existing renderers.

	`strahl`	`three-gpu-pathtracer` [15]	`Three.js PathTracer` [16]	`dspbr-pt` [17]
WebGPU Support	Yes	No	No	No
PBR Standard	*OpenPBR*	Custom	Custom	*DSPBR*
Documentation	yes	yes	minimal	minimal
Benchmark Setup	yes	no	no	no
Maintenance	provided	provided	lacking	lacking

Table 4.1: High-level comparison between four open-source path tracers for the web.

Comparison to Alternative Strategies

The chosen architecture paradigm serves as an alternative to offline rendering solutions. In addition, the decision to implement ray tracing techniques contrasts with rasterization-based rendering engines. This section shows the comparison between these different options. For rasterization, *Three.js* is used as an example, based on a minimal setup without pregenerated artifacts or advanced rendering techniques. For offline rendering, *RealityServer*[2] renderings used by EAO are shown as a representative example. The three different renderings are visualized in Figure 4.1.

(a) Scene rendered by *Three.js* using rasterization.

(b) Offline rendering with *RealityServer*, used by EAO [94].

(c) Scene rendered with the developed path tracer.

Figure 4.1: Comparison of different rendering techniques.

Visually, differences in global illumination effects between the two ray-traced renderings and the rasterization rendering are apparent. Ambient occlusion on the terminal block is more pronounced in the ray-traced images. Reflection on the metal part is also more noticeable, and the surrounding environment is considered. Additionally, the shadows are more realistic and consider all light sources. The differences between the two ray-traced images are less apparent due to the similar rendering techniques. Certain differences, such as the color of the metal part and the reflection on the red part, are mainly due to changes in the material properties. This demonstrates the potential of the proposed solution to provide high-fidelity renderings in near real-time.

4.2 Findings

The main novelty introduced in this work is the development of a path tracer with *WebGPU* using the *OpenPBR* surface shading model. *WebGPU* and *OpenPBR* are promising endeavors for the future of 3D rendering, but as of 2024, they are relatively new and have yet to be widely adopted.

[2]Platform for 3D rendering, which integrates the NVIDIA Iray global illumination rendering technology

State of WebGPU

WebGPU is a promising technology for *GPGPU* computations in the browser. The design reduces global state, enforces asynchrony, and is more explicit than *WebGL*. However, to date, there are certain limitations regarding support and features.

Browser Support

Due to the current state of support for *WebGPU* in Safari and Mozilla Firefox, the production readiness of *WebGPU* is still limited. Safari has announced plans to support *WebGPU* and has launched a preview version [95]. Firefox also has plans to support WebGPU [96]. Thanks to the extensive conformance test suite [71], it is more likely that the different implementations will be compatible with each other.
The main browser that supports *WebGPU* to date is Chrome. *WebGPU* has shipped to general use on desktops in May of 2023 [97]. Since January 2024, *WebGPU* is also supported on modern Android devices [98].
This means that using *WebGPU* is straightforward on most modern devices, with the notable exception of Apple iOS and iPadOS devices.

Ray Tracing

To date, *WebGPU* does not support some features common in modern rendering *APIs* and would be beneficial for the path tracer. The most prominent example is hardware-accelerated ray tracing. *APIs* such as *Vulkan* support hardware-accelerated ray tracing [99]. This entails helpers for building common acceleration structures, such as *BVH* and ray querying functions to determine intersections. *WebGPU* does not yet support these features, but discussions are ongoing to add extensions [100] as well as a demonstration implemented in a modified version of *Dawn* [101].

Debuggability

None of the major browser engines, Chrome, Firefox, and Safari, provide debugging as part of the developer tools. Tools for inspecting *WebGPU* applications exist [102, 103] but are limited in terms of feature set. While they are helpful in inspecting the pipelines, capturing frames, and inspecting resources, they do not provide debugging capabilities such as breakpoints, stepping, or variable inspection. For specific setups, there are methods to setup profiling [104], but these are not integrated into the browser developer tools and are dependent on the concrete machine hardware. Improvements in this area would facilitate troubleshooting.

Stability

To date, there are reports of stability issues with *WebGPU*. This includes browser crashes, but on macOS using Chrome, even system-wide crashes have been observed with faulty *WebGPU* code. Such issues can look like shown in Figure 4.2. Due to the

early stage of the technology and the complexity of the underlying system, such stability issues ought to be expected. As implementations mature, these issues are likely to be resolved.

Figure 4.2: Example of a provoked *WebGPU* crash on macOS; note the black squares which correspond in size to the dispatched workgroup size.

OpenPBR

The integration of the *OpenPBR* surface shading model enables realistic material representation. As it is a new standard, and no stable version was released when implementation started, aligning the path tracer with the standard was challenging. After the standard reached a stable version, the alignment effort between the implementation and the standard was reduced.
To date, only a few open-source projects have implemented the standard. Neither *Three.js* nor *Babylon.js* have direct *OpenPBR* support. However, there is a *MaterialX* loader for *Three.js* that could be used. For use in real-time rendering, alignment between *OpenPBR* and *glTF PBR* extensions could be beneficial. This alignment could reduce the computational resources required for surface shading and improve interoperability by establishing *glTF* extensions for *OpenPBR*.
Using *PBR* and the latest industry standards is beneficial and provides high-quality results. The adoption track of *OpenPBR* is promising, and initiatives such as NVIDIA's *OpenPBR* material library [85] can facilitate material creation. The standard is likely to be extended in the future. Such extensions could be specifications to support hair or volumetric effects.

4.3 Future Work

In order to accommodate other use cases, the renderer could be extended in various ways. The following section outlines possible future work, which includes extending *PBR* capabilities, improving performance, extending to other rendering architecture paradigms, general improvements for the *WebGPU* community, and aligning the renderer with other standards.

Rendering Improvements

The focus on the defined use case means that certain rendering effects are not implemented. These include:

- Volumetric Effects
- Hair and Fur
- Refraction
- Caustics
- Depth of Field — Effect of objects at different distances from the camera being in focus or out of focus.
- Motion Blur — Effect of objects moving quickly appearing blurred.

While refraction, caustics, depth of field, and motion blur are questions of light transport, volumetric effects, hair and fur are features that are not part of the *OpenPBR* specification. Other features in terms of rendering could be implemented. This section outlines some of the possibilities.

Spectral Rendering

Instead of using *RGB* color space, spectral rendering could leverage the full spectrum of visible light by modeling the light as a function of wavelength. This can improve the realism of the rendered images. Spectral rendering is well-suited for *PBR* and is a natural extension of the current implementation. It is also possible to support both *RGB* and spectral rendering.

Texture Support

While not critical for the given use case, texture support is a common feature in rendering engines. The current implementation does not use textures. The renderer could be extended to support sampling *OpenPBR* parameters from textures. This would enable more complex material variations and improve the realism of the rendered images.
This is distinct from texture mapping frequently used in rasterization-based rendering engines. Texture mapping is a technique that applies images to surfaces to simulate surface details. In *PBR*, textures are used to sample material properties such as base color, roughness, or metallicness.

Environment Mapping

The current implementation offers a environment light setup. However, no details about the environment are visible in the reflections. By adding environment mapping, more complex lighting scenarios can be achieved. This could be done by using a skybox or a *HDR* environment map.

Full OpenPBR support

The current implementation only supports the *OpenPBR* surface shading model features that are required for the given use case. The full *OpenPBR* specification includes additional features that could be implemented to improve the realism of the rendered images. These include:

- Coat - Secondary layer of specular highlight, often used for car paint.
- Fuzz – Layer for fuzzy or dusty surfaces.
- Thin-film iridescence - Effect of thin-films of material on the surface of an object. This can be seen in objects covered in grease, oil, or alcohol.
- Subsurface Scattering and Translucency
- Opacity and Transparency

Technical Improvements

The renderer uses *GPU* parallelization to accelerate the rendering process. However, the current implementation is not optimized for performance. This includes possible improvements on the *CPU* and *GPU* sides and the data transfer between the two. The following sections outline possible improvements and extensions of the current setup.

TypeScript Support for Memory Management

While `webgpu-utils` [88] is helpful for memory management, it does not provide TypeScript support for the generated definitions based on the underlying *WGSL* code. Type safety could reduce the likelihood of errors in the code. As an alternative, runtime checks could be implemented to ensure the data is correctly mapped to all fields of the underlying structure.

Web Worker Support

Web Workers are a web technology that allows to run scripts off the main thread. This can be used to offload *CPU*-heavy tasks to a separate thread to prevent blocking the main thread, which is responsible for rendering the user interface.

BVH Construction

The current implementation builds the *BVH* on the *CPU* and transfers it to the *GPU*. Corresponding research [58] suggests that moving parts of the construction to the *GPU* directly could improve performance. This would reduce the amount of data that needs to be transferred between the *CPU* and the *GPU*. The new *GPGPU* capabilities of *WebGPU* further enable this approach.

Independence of Three.js

For ease-of-use for developers, the renderer uses *Three.js* helpers. This aids developers familiar with *Three.js* in getting started with the renderer, as the configuration for scene loading and camera configuration is similar. However, the renderer does not depend on *Three.js* and could be used independently. The main drawback of the dependence is the increased bundle size. `three-mesh-bvh` [87] could also be exchanged for an alternative library. By removing the dependency, the bundle size would be reduced. To support developers, a binding to *Three.js* could be provided on top of the independent renderer.

Offline and Remote Rendering

As highlighted in section 1.3, it is possible to extend a real-time client-side renderer for offline and remote rendering scenarios. In order to implement offline rendering, one could opt to use a headless browser such as `Puppeteer`, a *Node.js*[3] library that provides a high-level *API* to control browsers. An alternative is to use *wgpu*, but this would necessitate a rewrite of the renderer. Possibly, the rewritten renderer could also be used in the web context by using *WebAssembly*[4].
For remote rendering, the renderer could be extended to render images on demand and encode them as video streams.

WebGPU Compatibility Mode

There is a proposal under active development that aims to extend the reach of *WebGPU* by providing a slightly restricted subset of *WebGPU* [105]. Considering the suggested limits of the compatibility mode, it could be possible to deploy the renderer onto a broader range of devices. However, it is important to consider that path tracing is a computationally expensive task and might not be suitable for all devices. Therefore, increasing the reach might only be beneficial in some cases.

Automatic Shader Conversion

During the specification phase of *WebGPU*, the relation to *SPIR-V*[5] was discussed [106]. In general, many of the modern shading languages can be compiled to one another. Projects such as `Tint`, which is part of *Dawn* [67] or `Naga` [107] could be used to compile shaders from different frontends to different backends. Similarly, other engines, such as *Three.js* with *TSL*[6], have their own shading languages that support a variety of backends [108]. Parts of *MaterialX* shader generation could be used to generate shaders for *WebGPU* and update them automatically as *OpenPBR* is updated.

[3] JavaScript runtime, frequently used for executing JavaScript outside of the browser
[4] Portable Binary-code format for executable programs available in modern browser engines
[5] *Standard Portable Intermediate Representation*, intermediate language for parallel computing and graphics developed by Khronos Group
[6] *Three.js Shading Language*, shading language used in *Three.js* which supports *GLSL* as well as *WGSL*

Low-Level Performance Optimizations

The current implementation is not optimized for performance. Therefore, optimizing the *WGSL* code could improve the performance of the renderer. Due to the nature of the design of *OpenPBR*, it would be possible to optimize and improve real-time rendering performance. One approach to do so is to establish so-called wavefront path tracing [65]. Instead of a single megakernel, the renderer could be split into multiple smaller kernels. One kernel would be responsible for ray generation, another for intersection testing, and multiple kernels for shading - one per workflow. This would reduce the divergence of the shader programs and could improve performance. Further investigation of potential performance improvements is required [109, 110].
This objective may be incompatible with the goal of providing automatic shader generation. Automatic shader generation is helpful but likely not as optimized as a carefully tuned implementation. However, both endeavors are of interest for potential benefits.

Quality Improvements

This section highlights changes to improve the quality of the rendered images, reduce the amount of samples required, and enable better comparison of the renderer to baseline renderers.

Sampling Performance Optimizations

Path tracing is computationally expensive and requires multiple samples per pixel to achieve accurate results. Consequently, the sampling process is noticeable during interactions with the scene. One technique to improve perceived interaction quality is to overlay the rendering with a rasterization preview during interactions. To reduce the number of samples required for high-quality renderings, techniques like neural radiance caching (NRC) [111] or *ReSTIR* [59] could be employed in future work.

Denoisers

The renderer currently provides two optional strategies for denoising: *OIDN* [91] and Gaussian filtering. However, the quality of the results varies depending on the scene. In certain setups, the changes introduced by *OIDN* are rather pronounced, which may necessitate disabling the denoiser. By employing alternative denoising algorithms, applied as post-processing steps, the quality of the results could be enhanced without introducing new artifacts into the renderings. Options include blockwise multi-order feature regression (BMFR) [112] and non-local means (NLM) denoising [113].

Qualitative Assessment

The provided results highlight the quantitative performance of the renderer. However, due to the nature of a renderer, qualitative assessment based on visual inspection is also used to determine the quality of the rendered images. This could be extended to include

more advanced metrics such as peak signal to noise ratio (PSNR), structural similarity (SSIM) [114], or learned perceptual image patch similarity (LPIPS) [115].
Such a comparison could be based on reference scenes such as the Cornell Box [11] or the Sponza Atrium [116]. To assess the differences, different comparisons could be conducted:

- Offline Renderer - Comparison to other offline renderers such as Mitsuba [117], *pbrt* [37], or Cycles which is used by *Blender* [118].
- Rasterization Renderer - Comparison to rasterization-based web renderers such as *Three.js* or *Babylon.js*.
- Web-based Path Tracer - Comparison to other web-based path tracers such as `three-gpu-pathtracer` [15], `Three.js PathTracer` [16], or `dspbr-pt` [17].

4.4 Conclusion

While there are a variety of areas to improve on, the proposed solution constitutes a fully functional path tracer encompassing technical features such as using *BVH*, parallelization on *GPU* based on *WebGPU*, and supporting *MIS*; rendering features including anti-aliasing, denoising options, tone mapping, generating a wide range of global illumination effects, and supporting the *OpenPBR* standard; usability features such as incremental rendering, camera controls, and scene loading; benchmarking setup for reliable performance measurements; and extensive documentation to facilitate use of the renderer.
These features fulfil the requirements of the use case and present an alternative to existing offline rendering solutions by permitting a higher degree of interactivity and alleviating the need to pregenerate all images, which facilitates offering more complex product families. Compared to remote rendering services, the approach reduces infrastructure cost and network dependency. The fidelity gained by using ray tracing techniques in combination with the *OpenPBR* standard provides a high-quality rendering solution without the need for pregenerated assets. The path tracer developed in this thesis is a suitable choice for the given use case of using production *CAD* data with manifold assembly configurations and customer-specific materials.
As shown in section 3.3, the performance is sufficient for near real-time renderings of assembled *CAD* models on the web. The use of *WebGPU* over *WebGL* as well as incorporating *OpenPBR* provides a distinction to existing open-source path tracers for the web. *WebGPU* has significant potential for the years to come and adoption of *OpenPBR* by the wider industry is a promising sign of the possible longevity of the chosen technology and standards. The open-source nature of the project facilitates extension and serves as inspiration for future initiatives.

5 Index

5.1 Glossary

A-Frame Open-source 3D engine for the web based on WebGL 11

AABB *Axis-Aligned Bounding Boxes*, bounding volume aligned with the coordinate axes 27, 28, 53

API *Application Programming Interface* 12, 33–36, 38, 39, 45, 58, 65, 69, 73, 74

ASCII *American Standard Code for Information Interchange*, character encoding standard for electronic communication 41

ASWF *Academy Software Foundation*, organization promoting development and adoption of open-source software in the motion picture industry 46, 47

Babylon.js Open-source 3D engine for the web based on WebGL and WebGPU 11, 36, 62, 66, 71

Blender open-source 3D computer graphics software toolset 46, 47, 71

BMRT *Blue Moon Rendering Tools*, early free and open-source ray tracing software which was RenderMan compatible 29, 33

BRDF *Bidirectional Reflectance Distribution Function*, describes the reflection of light off a surface 43, 44, 46, 87

BSDF *Bidirectional Scattering Distribution Function*, describes the interaction of light with surfaces 43, 44, 47, 54, 72

BTDF *Bidirectional Transmittance Distribution Function*, describes the transmission of light through a surface 43, 44

BVH *Bounding Volume Hierarchy*, common tree-based acceleration structure 27, 28, 33, 50, 51, 53, 54, 59, 60, 65, 68, 71, 86–88

BxDF *Bidirectional Distribution Functions*, describe the interaction of light with surfaces, commonly used interchangeably with *BSDF* 43

CAD *Computer-Aided Design*, use of computers to aid in modeling of a design 4–6, 46, 49, 61, 71, 75, 88

CDF *Cumulative Distribution Function*, describes the probability that a random variable is less than or equal to a given value 16, 86

CDN *Content Delivery Network*, distributed group of servers for caching content near end users. 8

COLLADA *Collaborative Design Activity*, 3D exchange format managed by *Khronos Group* 41

CPU *Central Processing Unit*, the main processing unit of a computer 33, 38, 39, 49, 50, 58, 60, 68, 87

Dawn Open-source and cross-platform WebGPU implementation developed by Google 36, 49, 65, 69

Deno JavaScript runtime, alternative to Node.js 36

DirectX 12 *API* developed by Microsoft for 3D graphics and computing 35, 36

DSPBR *Dassault Systèmes PBR Shading Model*, specification for PBR shading model 12, 46, 63

FBX *Filmbox*, proprietary 3D exchange format nowadays managed by Autodesk 40

FPS *Frames per second*, measure of the rate at which consecutive images are rendered and displayed 62

GGX *Ground Glass Unknown*, frequently used microfacet distribution function derived by Neyret and independently by Walter et al. [75, 76] 44

GLSL *OpenGL Shading Language*, shading language used in *OpenGL* 34, 47, 69, 76

glTF *Graphics Library Transmission Format, common 3D exchange format optimized for transmission and real-time rendering.* 7, 41, 42, 45, 50, 63, 66, 87

GPGPU *General-Purpose GPU* Programming, using GPUs for non-graphics tasks 34–36, 65, 68

GPU *Graphics Processing Unit*, specialized processor for parallel computation 8, 12, 33, 34, 36–39, 45, 49, 50, 58, 60, 68, 71, 75, 87, 88

gzip *GNU zip*, file format and software application for compression and decompression. 40

HDR *High Dynamic Range*, signal with a large range of luminance values 32, 67

JPEG *Joint Photographic Experts Group*, common method for lossy image compression 52

Khronos Group consortium for developing interoperability standards for 3D graphics 35, 41, 73

Lavarand Hardware *RNG* based on pictures taken of lava lamps to generate randomness 31

LOD *Level of Detail*, defines finer or coarser representations of a model 6, 58, 88

MaterialX Open standard for rich material definition 46, 47, 54, 66, 69

Metal *API* developed by Apple for 3D graphics and computing 33, 35, 36

MIS *Multiple Importance Sampling*, technique to combine multiple sampling strategies in Monte Carlo integration 26, 54, 71

MIT license Permissive license originating at the Massachusetts Institute of Technology (MIT) 49

ML *Machine Learning*, field of computer science that uses data and algorithms to give computer systems the ability to improve accuracy without being explicitly programmed 33, 35, 55

MTL *Material Template Library*, a companion file format to *OBJ* for material definitions 40

NDF *Normal Distribution Function*, describes the distribution of microfacet normals on a surface 44

Node.js JavaScript runtime, frequently used for executing JavaScript outside of the browser 69

npm package manager for JavaScript 49, 56, 63

OBJ OBJ geometry file format, a simple text-based format for 3D models 40, 50, 74

OIDN *Open Image Denoise*, open-source library for denoising images 55, 70

OpenGL *Open Graphics Library*, cross-platform API for rendering 2D and 3D graphics 34–36, 73

OpenGL ES *OpenGL for Embedded Systems*, subset of OpenGL designed for embedded systems like smartphones 35

OpenPBR Open specification of surface shading model 47–49, 53, 54, 57, 61–64, 66–71, 87, 88

OSL *Open Shading Language*, shading language for production global illumination renderers maintained by Academy Software Foundation 46, 47

PBR *Physically Based Rendering*, rendering technique that simulates real-world materials 32, 42, 43, 45, 46, 54, 62, 63, 66, 67

pbrt open-source ray tracing renderer, described in the book Physically Based Rendering [37] 42, 54, 71

PCG *Permuted Congruential Generator*, a family of random number generators 31, 51

PDF *Probability Density Function*, describes the likelihood of a random variable 16, 25, 86

PlayCanvas Open-source 3D engine for the web based on WebGL 11, 36

POV-Ray *Persistence of Vision Ray Tracer*, cross-platform ray tracing renderer officially released in 1991 29

RealityServer Platform for 3D rendering, which integrates the NVIDIA Iray global illumination rendering technology 64

RenderMan A rendering software for photorealistic 3D rendering developed by Pixar 29, 46

ReSTIR *Reservoir-based Spatio-Temporal Importance Resampling*, technique to optimize sampling in path tracing 33, 70

RGB *red, green, and blue*, common system for color representation in computer graphics 53, 67

Rhino3D *Rhinoceros*, proprietary 3D computer graphics and *CAD* toolset 46

RNG Random Number Generator 30, 31, 38, 51, 52, 74, 86

SDK *Software Development Kit* 40

SIMD *Single Instruction, Multiple Data*, type of parallel processing of data points with a single instruction as defined by Flynn [52] 33, 75

SIMT *Single Instruction, Multiple Threads*, extension to *SIMD* which is frequently used on modern *GPUs* 33, 34

SPIR-V *Standard Portable Intermediate Representation*, intermediate language for parallel computing and graphics developed by Khronos Group 69

STEP *Standard for the Exchange of Product model data*, a standard formalized in ISO 10303 for product manufacturing information. 5, 41

STL *Stereolithography*, 3D exchange format developed by 3D Systems 41

Three.js Open-source 3D engine for the web based on WebGL 11, 12, 36, 50, 62, 64, 66, 69, 71, 76

TSL *Three.js Shading Language*, shading language used in *Three.js* which supports *GLSL* as well as *WGSL* 69

TypeScript Typed superset of JavaScript developed by Microsoft 56

uber shader Shader design that offers parameter configuration but no custom shader code 45, 47, 48

Unity Proprietary, cross-platform game engine developed by Unity Technologies 11, 36, 46

Unreal Unreal Engine, cross-platform game engine developed by Epic Games 46

USD *Universal Scene Description*, 3D exchange format 41

Vulkan Open standard for cross-platform 3D graphics and computing 35, 36, 65

W3C *World Wide Web Consortium*, a non-profit organization dedicated to developing web standards 36

WebAssembly Portable Binary-code format for executable programs available in modern browser engines 69

WebGL *Web Graphics Library*, since 2011 the de-facto standard API for rendering 3D graphics on the web 11, 12, 35, 36, 62, 65, 71

WebGPU A new standard for 3D graphics on the web 12, 32, 33, 35–37, 49, 50, 55, 56, 58, 62–66, 68, 69, 71, 87, 88

wgpu Open-source and cross-platform WebGPU implementation 36, 49, 69

WGSL *WebGPU Shading Language* 37–39, 53, 68–70, 76, 87

XML *Extensible Markup Language* 41

5.2 Bibliography

[1] S. Stucki and P. Ackermann, „Physically-based Path Tracer using WebGPU and OpenPBR", *Proceedings WEB3D '24: The 29th International ACM Conference on 3D Web Technology*, 2024.

[2] J. Stjepandić, H. Liese and A. J. C. Trappey, *Intellectual Property Protection*. Cham: Springer International Publishing, 2015, pp. 521–551. [Online]. URL: https://doi.org/10.1007/978-3-319-13776-6_18

[3] D. Luebke, *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[4] S. Shi and C.-H. Hsu, „A Survey of Interactive Remote Rendering Systems", *ACM Comput. Surv.*, Vol. 47, No. 4, may 2015. [Online]. URL: https://doi.org/10.1145/2719921

[5] L. Corneo, M. Eder, N. Mohan, A. Zavodovski, S. Bayhan, W. Wong, P. Gunningberg, J. Kangasharju and J. Ott, „Surrounded by the Clouds: A Comprehensive Cloud Reachability Study", in *Proceedings of the Web Conference 2021*, Series WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 295–304. [Online]. URL: https://doi.org/10.1145/3442381.3449854

[6] Three.js. (2024, July) *Three.js*. [Online]. URL: https://threejs.org/

[7] Babylon.js. (2024, July) *Babylon.js*. [Online]. URL: https://www.babylonjs.com/

[8] PlayCanvas. (2024, July) *PlayCanvas*. [Online]. URL: https://playcanvas.com/

[9] A-Frame. (2024, July) *A-Frame*. [Online]. URL: https://aframe.io/

[10] Unity. (2024, April) *Unity - Web browser compatibility*. [Online]. URL: https://docs.unity3d.com/2023.2/Documentation/Manual/webgl-browsercompatibility.html

[11] C. M. Goral, K. E. Torrance, D. P. Greenberg and B. Battaile, „Modeling the interaction of light between diffuse surfaces", *ACM SIGGRAPH computer graphics*, Vol. 18, No. 3, pp. 213–222, 1984.

[12] E. Wallace. (2011, August) *WebGL Path Tracing*. [Online]. URL: https://experiments.withgoogle.com/webgl-path-tracing

[13] A. Jaškauskas. (2020, February) *Porting a CUDA Path Tracer to WebGL*. [Online]. URL: https://github.com/driule/webgl-path-tracer

[14] M. Nilsson and A. Ottedag, „Real-time path tracing of small scenes using WebGL", 2018.

[15] G. Johnson. (2024, May) *three-gpu-pathtracer*. [Online]. URL: https://github.com/gkjohnson/three-gpu-pathtracer

[16] E. Loftis. (2024, May) *THREE.js-PathTracing-Renderer*. [Online]. URL: https://github.com/erichlof/THREE.js-PathTracing-Renderer

[17] B. Sdorra and T. Häußler. (2022, July) *dspbr-pt*. [Online]. URL: https://github.com/DassaultSystemes-Technology/dspbr-pt

[18] R. L. Dotson, „Dynamical.JS: A composable framework for online exploratory visualization of arbitrarily-complex multivariate networks", Master's thesis, Harvard University Division of Continuing Education, 2022.

[19] C. Bohak, D. Kovalskyi, S. Linev, A. Mrak Tadel, S. Strban, M. Tadel and A. Yagil, „RenderCore – a new WebGPU-based rendering engine for Root-eve", *EPJ Web of Conferences*, Vol. 295, p. 03035, 2024.

[20] G. Kimmersdorfer, D. Wolf and M. Waldner, „WebGPU for Scalable Client-Side Aggregate Visualization", *Proceedings of Eurographics - The European Association for Computer Graphics*, pp. 1–3, 2023.

[21] W. Bi, Y. Ma, Y. Han, Y. Chen, D. Tian and J. Du, „FusionRender: Harnessing WebGPU's Power for Enhanced Graphics Performance on Web Browsers", in *Proceedings of the ACM on Web Conference 2024*, Series WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2890–2901. [Online]. URL: https://doi.org/10.1145/3589334.3645395

[22] Z. Usta, „WebGPU: A new Graphic API for 3D WebGIS Applications", *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. XLVIII-4/W9-2024, pp. 377–382, 2024. [Online]. URL: https://isprs-archives.copernicus.org/articles/XLVIII-4-W9-2024/377/2024/

[23] E. Fransson and J. Hermansson, „Performance comparison of WebGPU and WebGL in the Godot game engine", 2023. [Online]. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-24706

[24] S. Chickerur, S. Balannavar, P. Hongekar, A. Prerna and S. Jituri, „WebGL vs. WebGPU: A Performance Analysis for Web 3.0", *Procedia Computer Science*, Vol. 233, pp. 919–928, 2024. [Online]. URL: https://www.sciencedirect.com/science/article/pii/S1877050924006410 5th International Conference on Innovative Data Communication Technologies and Application (ICIDCA 2024).

[25] G. R. Fowles, *Introduction to modern optics*. Courier Corporation, 1989.

[26] J. H. Lambert, *Photometria*, 1760.

[27] A. Appel, „Some techniques for shading machine renderings of solids", in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*,

Series AFIPS '68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, p. 37–45. [Online]. URL: https://doi.org/10.1145/1468075.1468082

[28] T. Whitted, „An improved illumination model for shaded display", in *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, Series SIGGRAPH '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 14. [Online]. URL: https://doi.org/10.1145/800249.807419

[29] T. Whitted, „Origins of Global Illumination", *IEEE Computer Graphics and Applications*, Vol. 40, No. 1, pp. 20–27, 2020.

[30] N. Greene, „Environment mapping and other applications of world projections", *IEEE computer graphics and Applications*, Vol. 6, No. 11, pp. 21–29, 1986.

[31] T. Stachowiak, „Advances in real time rendering, part I, Stochastic Screen-Space Reflections", in *ACM SIGGRAPH 2015 Courses*, Series SIGGRAPH '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. URL: https://doi.org/10.1145/2776880.2787701

[32] L. Bavoil and M. Sainz, „Screen space ambient occlusion", *NVIDIA developer information: http://developer.nvidia.com*, Vol. 6, No. 2, 2008.

[33] T. Ritschel, T. Grosch and H.-P. Seidel, „Approximating dynamic global illumination in image space", in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, Series I3D '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 75–82. [Online]. URL: https://doi.org/10.1145/1507149.1507161

[34] J. T. Kajiya, „The rendering equation", in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[35] E. Veach, *Robust Monte Carlo Methods for Light Transport Simulation*. Stanford University, Department of Computer Science, 1998.

[36] M. H. Kalos and P. A. Whitlock, *Monte carlo methods*. John Wiley & Sons, 2009.

[37] M. Pharr, W. Jakob and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. The MIT Press, 2023.

[38] S. M. Rubin and T. Whitted, „A 3-dimensional representation for fast rendering of complex scenes", *SIGGRAPH Comput. Graph.*, Vol. 14, No. 3, p. 110–116, jul 1980. [Online]. URL: https://doi.org/10.1145/965105.807479

[39] T. Möller and B. Trumbore, „Fast, minimum storage ray/triangle intersection", in *ACM SIGGRAPH 2005 Courses*, Series SIGGRAPH '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 7–es. [Online]. URL: https://doi.org/10.1145/1198555.1198746

[40] POV-Ray Team, *POV-Ray: Documentation.* [Online]. URL: https://www.povray.org/documentation/view/3.6.1/10/

[41] L. Gritz and J. K. Hahn, „BMRT: A Global Illumination Implementation of the RenderMan Standard", *Journal of Graphics Tools*, Vol. 1, No. 3, pp. 29–47, 1996. [Online]. URL: https://doi.org/10.1080/10867651.1996.10487462

[42] Pixar, „RenderMan 11 Release Notes", Technical Report. [Online]. URL: https://renderman.pixar.com/resources/RenderMan_20/rnotes-11.0.html

[43] C. Barré-Brisebois, H. Halén, G. Wihlidal, A. Lauritzen, J. Bekkers, T. Stachowiak and J. Andersson, *Hybrid Rendering for Real-Time Ray Tracing.* Berkeley, CA: Apress, 2019, pp. 437–473. [Online]. URL: https://doi.org/10.1007/978-1-4842-4427-2_25

[44] M. E. O'Neill, „PCG: A family of simple fast space-efficient statistically good algorithms for random number generation", *ACM Transactions on Mathematical Software*, 2014.

[45] R. Gennaro, „Randomness in cryptography", *IEEE Security & Privacy*, Vol. 4, No. 2, pp. 64–67, 2006.

[46] J. Liebow-Feeser. (2017, June) *LavaRand in Production.* [Online]. URL: https://blog.cloudflare.com/randomness-101-lavarand-in-production

[47] G. Marsaglia, „Xorshift rngs", *Journal of Statistical software*, Vol. 8, pp. 1–6, 2003.

[48] M. Matsumoto and T. Nishimura, „Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", *ACM Trans. Model. Comput. Simul.*, Vol. 8, No. 1, p. 3–30, jan 1998. [Online]. URL: https://doi.org/10.1145/272991.272995

[49] NVIDIA Corporation, „NVIDIA DLSS". [Online]. URL: https://www.nvidia.com/en-us/geforce/technologies/dlss/

[50] E. Reinhard, M. Stark, P. Shirley and J. Ferwerda, „Photographic tone reproduction for digital images", *ACM Trans. Graph.*, Vol. 21, No. 3, p. 267–276, jul 2002. [Online]. URL: https://doi.org/10.1145/566654.566575

[51] E. Lalish, „Neutral Tone Mapping for PBR Color Accuracy", in *ACM SIGGRAPH 2024 Talks*, Series SIGGRAPH '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. URL: https://doi.org/10.1145/3641233.3664313

[52] M. J. Flynn, „Very high-speed computing systems", *Proceedings of the IEEE*, Vol. 54, No. 12, pp. 1901–1909, 1966.

[53] M. J. Flynn, „Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, Vol. C-21, No. 9, pp. 948–960, 1972.

[54] G. S. Watkins, „A real time visible surface algorithm", Ph.D. thesis, 1970, AAI7023061.

[55] A. Levinthal and T. Porter, „Chap - a SIMD graphics processor", *SIGGRAPH Comput. Graph.*, Vol. 18, No. 3, p. 77–82, jan 1984. [Online]. URL: https://doi.org/10.1145/964965.808581

[56] W. J. Dally, S. W. Keckler and D. B. Kirk, „Evolution of the Graphics Processing Unit (GPU)", *IEEE Micro*, Vol. 41, No. 6, pp. 42–51, 2021.

[57] K. Akeley, „The Silicon Graphics 4D/240GTX superworkstation", *IEEE Computer Graphics and Applications*, Vol. 9, No. 4, pp. 71–83, 1989.

[58] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke and D. Manocha, „Fast BVH construction on GPUs", in *Computer Graphics Forum*, Vol. 28, No. 2. Wiley Online Library, 2009, pp. 375–384.

[59] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn and W. Jarosz, „Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting", *ACM Trans. Graph.*, Vol. 39, No. 4, aug 2020. [Online]. URL: https://doi.org/10.1145/3386569.3392481

[60] Y. Ouyang, S. Liu, M. Kettunen, M. Pharr and J. Pantaleoni, „ReSTIR GI: Path Resampling for Real-Time Path Tracing", *Computer Graphics Forum*, Vol. 40, No. 8, pp. 17–29, 2021. [Online]. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14378

[61] D. Lin, M. Kettunen, B. Bitterli, J. Pantaleoni, C. Yuksel and C. Wyman, „Generalized resampled importance sampling: Foundations of restir", *ACM Transactions on Graphics (TOG)*, Vol. 41, No. 4, pp. 1–23, 2022.

[62] S. Zhang, D. Lin, M. Kettunen, C. Yuksel and C. Wyman, „Area ReSTIR: Resampling for Real-Time Defocus and Antialiasing", Vol. 43, No. 4, July 2024.

[63] NVIDIA Corporation, „NVIDIA RTX Ray Tracing". [Online]. URL: https://developer.nvidia.com/rtx/ray-tracing

[64] Apple Developer, „Explore GPU advancements in M3 and A17 Pro - Tech Talks", 2023. [Online]. URL: https://developer.apple.com/videos/play/tech-talks/111375

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

[66] W3C group for GPU web standards. (2024, August) *WebGPU*. [Online]. URL: https://www.w3.org/TR/webgpu/

[67] Google. (2024, July) *Dawn source code.* [Online]. URL: https://dawn.googlesource.com/dawn

[68] gfx-rs. (2024, July) *wgpu source code.* [Online]. URL: https://github.com/gfx-rs/wgpu

[69] Apple. (2024, July) *WebKit source WebGPU implementation.* [Online]. URL: https://github.com/WebKit/WebKit/tree/main/Source/WebGPU

[70] gfx-rs. (2024, April) *Differences between our subgroup implementation and the WebGPU proposal.* [Online]. URL: https://github.com/gfx-rs/wgpu/issues/5555

[71] W3C group for GPU web standards. (2024, May) *WebGPU Conformance Test Suite.* [Online]. URL: https://github.com/gpuweb/cts

[72] W3C group for GPU web standards. (2024, July) *WebGPU Shading Language.* [Online]. URL: https://www.w3.org/TR/WGSL

[73] S. Marjudi, M. M. Amran, K. A. Abdullah, S. Widyarto, N. A. Majid and R. Sulaiman, „A Review and Comparison of IGES and STEP", in *Proceedings Of World Academy Of Science, Engineering And Technology*, Vol. 62, 2010, pp. 1013–1017.

[74] Khronos Group. (2021, October) *glTF 2.0 Specification.* [Online]. URL: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html

[75] B. Walter, S. R. Marschner, H. Li and K. E. Torrance, „Microfacet Models for Refraction through Rough Surfaces." *Rendering techniques*, Vol. 2007, p. 18th, 2007.

[76] E. Heitz, „Sampling the ggx distribution of visible normals", *Journal of Computer Graphics Techniques (JCGT)*, Vol. 7, No. 4, pp. 1–13, 2018.

[77] B. Burley, „Physically-based Shading at Disney", in *Acm Siggraph*, Vol. 2012. Walt Disney Animation Studios, 2012, pp. 1–7.

[78] Pixar. (2015, July) *RenderMan PxrDisney Material.* [Online]. URL: https://renderman.pixar.com/resources/RenderMan_20/PxrDisney.html

[79] I. Georgiev, J. Portsmouth, Z. Andersson, A. Herubel, A. King, S. Ogaki and F. Servant. (2023, Februar) *Autodesk Standard Surface.* [Online]. URL: https://autodesk.github.io/standard-surface/

[80] Adobe. (2021, October) *Adobe Standard Material.* [Online]. URL: https://helpx.adobe.com/substance-3d-general/adobe-standard-material/asm-specifications.html

[81] Dassault Systèmes. (2024, June) *Enterprise PBR Shading Model.* [Online]. URL: https://dassaultsystemes-technology.github.io/EnterprisePBRShadingModel/spec-2025x.md.html

[82] N. Harrysson, D. Smythe and J. Stone. (2019, July) *MaterialX Physically-Based Shading Nodes Introduction.* [Online]. URL: https://materialx.org/assets/MaterialX.v1.37REV2.PBRSpec.pdf

[83] Z. Andersson, P. Edmondson, J. Guertault, A. Herubel, A. King, P. Kutz, A. Machizaud, J. Portsmouth, F. Servant and J. Stone, „OpenPBR Surface Specification", Academy Software Foundation (ASWF), Tech. Rep., 2024. [Online]. URL: https://academysoftwarefoundation.github.io/OpenPBR/

[84] Academy Software Foundation (ASWF). (2024, June) *Academy Software Foundation Releases OpenPBR 1.0.* [Online]. URL: https://www.aswf.io/blog/academy-software-foundation-releases-openpbr-1-0/

[85] A. Langlands. (2024, March) *Unlock Seamless Material Interchange for Virtual Worlds with OpenUSD, MaterialX, and OpenPBR.* [Online]. URL: https://developer.nvidia.com/blog/unlock-seamless-material-interchange-for-virtual-worlds-with-openusd-materialx-and-openpbr/

[86] A. Langlands. (2023, November) *Unlock Seamless Material Interchange for Virtual Worlds with OpenUSD, MaterialX, and OpenPBR.* [Online]. URL: https://docs.blender.org/manual/en/4.0/render/shader_nodes/shader/principled.html

[87] G. Johnson. (2024, July) *three-mesh-bvh.* [Online]. URL: https://github.com/gkjohnson/three-mesh-bvh

[88] G. Tavares. (2024, July) *webgpu-utils.* [Online]. URL: https://github.com/greggman/webgpu-utils

[89] J. Portsmouth. (2024, June) *OpenPBR-viewer.* [Online]. URL: https://github.com/portsmouth/OpenPBR-viewer

[90] M. Morrone. (2021, November) *Fast glsl deNoise spatial filter.* [Online]. URL: https://github.com/BrutPitt/glslSmartDeNoise

[91] A. T. Áfra, „Intel® Open Image Denoise", 2024, https://www.openimagedenoise.org.

[92] Y. Shen. (2024, August) *Open Image Denoise on the Web.* [Online]. URL: https://github.com/pissang/oidn-web

[93] G. Turk and M. Levoy, „Zippered polygon meshes from range images", in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, Series SIGGRAPH '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 311–318. [Online]. URL: https://doi.org/10.1145/192161.192241

[94] EAO. (2024, August) *Emergency stop switch, Series 45.* [Online]. URL: https://eao.com/product/45-2C36.1920.000~45-2300.1000.000~45-311.1X10~45-

312.1X10/emergency_stop_switch/en/nothalt-taste-baureihe-45-40-mm-1-oe-1-s-ip66-ip67-ip69k-schraubanschluss

[95] M. Wyrzykowski. (2023, December) *WebGPU now available for testing in Safari Technology Preview*. [Online]. URL: https://webkit.org/blog/14879/webgpu-now-available-for-testing-in-safari-technology-preview/

[96] Mozilla. (2023, April) *Mozilla Platform GFX WebGPU*. [Online]. URL: https://wiki.mozilla.org/Platform/GFX/WebGPU

[97] Google. (2023, April) *Chrome Platform Status: WebGPU*. [Online]. URL: https://chromestatus.com/feature/6213121689518080

[98] Google. (2024, January) *Chrome Platform Status: WebGPU on Android*. [Online]. URL: https://chromestatus.com/feature/5119617865613312

[99] Khronos Group. (2020, December) *Vulkan SDK, Tools and Drivers are Ray Tracing Ready*. [Online]. URL: https://www.khronos.org/news/press/vulkan-sdk-tools-and-drivers-are-ray-tracing-ready

[100] W3C group for GPU web standards. (2020, January) *Ray Tracing extension*. [Online]. URL: https://github.com/gpuweb/gpuweb/issues/535

[101] F. Maier. (2020, September) *Hardware Ray tracing extension for Chromium WebGPU*. [Online]. URL: https://github.com/maierfelix/dawn-ray-tracing

[102] B. Duncan. (2024, August) *Inspection debugger for WebGPU*. [Online]. URL: https://github.com/brendan-duncan/webgpu_inspector

[103] T. Aoyagi. (2023, May) *webgpu-devtools*. [Online]. URL: https://github.com/takahirox/webgpu-devtools

[104] B. Jones. (2024, January) *Profiling WebGPU with PIX*. [Online]. URL: https://toji.dev/webgpu-profiling/pix

[105] S. White. (2023, October) *WebGPU Compatibility Mode*. [Online]. URL: https://github.com/gpuweb/gpuweb/blob/bda661a/proposals/compatibility-mode.md

[106] W3C group for GPU web standards. (2020, February) *Define what bijective to SPIR-V means*. [Online]. URL: https://github.com/gpuweb/gpuweb/issues/582

[107] gfx-rs. (2023, October) *Naga source code*. [Online]. URL: https://github.com/gfx-rs/naga

[108] Three.js. (2024, August) *Three.js Shading Language*. [Online]. URL: https://github.com/mrdoob/three.js/wiki/Three.js-Shading-Language

[109] B. Yalçıner and A. O. Akyüz, „Path guiding for wavefront path tracing: A memory efficient approach for GPU path tracers", *Computers & Graphics*, Vol. 121, p. 103945, 2024. [Online]. URL: https://www.sciencedirect.com/science/article/pii/S0097849324000803

[110] W. Jakob. (2024, April) *GPU Rendering not that much faster.* [Online]. URL: https://github.com/mitsuba-renderer/mitsuba2/issues/72#issuecomment-610631631

[111] T. Müller, F. Rousselle, J. Novák and A. Keller, „Real-time neural radiance caching for path tracing", *ACM Trans. Graph.*, Vol. 40, No. 4, jul 2021. [Online]. URL: https://doi.org/10.1145/3450626.3459812

[112] M. Koskela, K. Immonen, M. Mäkitalo, A. Foi, T. Viitanen, P. Jääskeläinen, H. Kultala and J. Takala, „Blockwise Multi-Order Feature Regression for Real-Time Path-Tracing Reconstruction", *ACM Trans. Graph.*, Vol. 38, No. 5, jun 2019. [Online]. URL: https://doi.org/10.1145/3269978

[113] A. Buades, B. Coll and J.-M. Morel, „A non-local algorithm for image denoising", in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 2, 2005, pp. 60–65 vol. 2.

[114] Z. Wang, A. Bovik, H. Sheikh and E. Simoncelli, „Image quality assessment: from error visibility to structural similarity", *IEEE Transactions on Image Processing*, Vol. 13, No. 4, pp. 600–612, 2004.

[115] R. Zhang, P. Isola, A. A. Efros, E. Shechtman and O. Wang, „The Unreasonable Effectiveness of Deep Features as a Perceptual Metric", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[116] M. Dabrovic, „Sponza atrium", 2002.

[117] W. Jakob, S. Speierer, N. Roussel and D. Vicini, „Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering", *Transactions on Graphics (Proceedings of SIGGRAPH)*, Vol. 41, No. 4, July 2022.

[118] Blender Foundation. (2024) *Cycles: Open Source Production Rendering.* [Online]. URL: https://www.cycles-renderer.org/

5.3 List of Figures

5.4 List of Tables