Rust-V

Physically Based Spectral Rendering

Supervised by: Prof. Dr. David Bommes, Simone Raimondi
Institute of Computer Science, University of Bern

Julius Oeftiger

8th July 2022

Contents

Acknowledgement

It was during a fascinating introductory course to computer graphics by Prof. Bommes that I became interested in the modern art of ray tracing, and I thank him for stimulating my curiosity. I could not have undertaken this journey without him enabling this work outside his usual scope of bachelor theses. Furthermore, I am extremely grateful to my supervisor, Simone Raimondi, who provided a great amount of support and time. Our splendid debugging and discussion sessions span many hours, all of them enlightening.

My deepest gratitude goes to my family. My parents, Eva Maria and Uwe, and my siblings, Rebecca and Adrian Oeftiger, all supported me along this way, reigniting my passion countless times over.

Last but certainly not least, I want to thank Chenny Kate Sampiano Malfarta. Her faraway support and encouragement were not only like a pillar, they were like a lighthouse in some very dark nights.

Abstract

Physically based rendering is extensively used in the TV animation industry (like Disney [10] or Pixar [17]) and the video game industry. This rendering technique uses an advanced light model to solve the rendering equation, leading to photo-realistic representations of our real world.

An improvement thereon is to include the dispersive nature of materials like glass in the form of a spectral dependency in the ray tracing. This spectral dependency leads to the splitting of one light bundle into its individual wavelengths, each tracing its own path. However, this is computationally expensive and methods to improve performance become important.

This work compares two methods of reducing a discretised visible spectrum to increase the performance of spectral physically based rendering: the "hero" wavelength sampling and the "random" wavelength sampling. For this purpose, a Monte-Carlo ray tracing engine has been implemented in the Rust programming language. The convergence speed of both sampling methods has been assessed using several metrics for the resulting image quality (namely the Mean Squared Error, the Peak Signal to Noise Ratio, and the Structural Similarity Index Measure).

This study finds that the hero sampling approach consistently outperforms random wavelength sampling. In the scenes presented, the hero wavelength sampling can be measured to converge up to

- 2.04x faster according to the Mean Squared Error, and
- 1.13x faster according to the Peak Signal to Noise Ratio, and have
- 1.23x less variance,

which demonstrates the significant advantages of hero wavelength sampling as a spectral sampling algorithm.

Chapter 1

Introduction

Physically based rendering (PBR) is an advanced application of photo-realistic light simulation to computer graphics. PBR treats light as rays intersecting and reflecting in the scene, modelling the flow of light and therefore achieving high render realism. Spectral PBR takes this a step further, deviating from RGB rendering by applying the dispersion phenomenon to the visible light spectrum: considering the wavelength-dependant optical refraction creates rainbow-like colour effects following the physics of optics and hence leads to an even more realistic simulation. The latter builds the core motivation of this work.

In this thesis, a spectral PBR tracer (PBRT) has been implemented in Rust [1] following the famous PBRT book[1] [16]. This new software is called Rust-*V*. Some ideas were also taken from Spectral Clara Lux Tracer (SCLT) [9] [20]. While this is not a full recreation, changes have been made to incorporate full spectral dependency and, therefore, the ability to simulate the dispersion effect, which cannot be found in the original PBRT. Rust has been chosen over `C++` as it provides fearless memory safety and concurrency. Rust-*V* uses the unbiased backwards-ray-tracing method.

Since PBRT is already based on physically based ray tracing, I decided to enhance the method to multi-wavelength tracing. Conceptually, I divided the visible spectrum into equally distributed discrete sample points. This raises the question of the computational effort required to trace such a large distribution of wavelengths individually. As a solution, instead of tracing all wavelengths individually, I chose a smaller subspace of the visible spectrum. This bundle is traced as one until a spectral-dependent scattering event occurs, potentially continuing as individual traces.

This thesis takes two methods of choosing this subspace, namely random and Hero sampling [23], and compares their convergence rate and the computational effort needed for a final noise-free image.

[1] version 3, at the time of writing version 4 is in the works

Chapter 2

Theory

To simulate a virtual scene, it is crucial to know the process of how vision works. Whether it be light rays hitting the cones and rods in a human eye or on a pixel in a camera sensor, both measure the light intensity coming from a certain incoming direction. This incoming light may either be directly from a light source like the sun or a light bulb, or it may have been reflected or refracted by objects along its way.

This is the basis of how ray tracing works and can be qualitatively expressed using the following equation.

2.1 Rendering Equation

The *rendering equation* describes the incoming light in these two parts:

1. the emission from the point of interest, and
2. all reflections from that point towards the observer.

The latter can be effectively modelled with an integral over the hemisphere of the point of all light rays coming towards it. This reflected light itself once again stems from either an emission or another reflection.
The rendering equation is therefore a recursive function, tracing back all incoming light rays to their source emitter:

$$L_o(x, \omega_o, \lambda) = L_e(x, \omega_o, \lambda) + \int_\Omega f_r(x, \omega_i, \omega_o, \lambda) L_i(x, \omega_i, \lambda) \underbrace{(n \cdot \omega_i)}_{\cos\theta_i} d\omega_i \tag{2.1}$$

where

- $L_o(x, \omega_o, \lambda)$ is the total radiance of λ from point x,
- $x \in \mathbb{R}^3$ is the point in space we are evaluating,
- $\omega_o \in \mathbb{R}^3$ is the direction of the outgoing light,
- λ is the wavelength of the light,
- $L_e(x, \omega_o, \lambda)$ is the spectral emission,
- Ω is the hemisphere described by the tuple (x, n),
- $f_r(x, \omega_i, \omega_o, \lambda)$ is the bidirectional reflectance distribution function (BRDF) to evaluate the transferred radiance of λ from ω_i to ω_o at point x,
- $\omega_i \in \mathbb{R}^3$ is the direction of the incoming light,
- $L_i(x, \omega_i, \lambda)$ is the incoming radiance of λ from direction ω_i to point x,

- $n \in \mathbb{R}^3$ is the surface normal of point x. We describe the dot product with $\cos\theta_i$, the incidence angle, which weakens the outward irradiance.

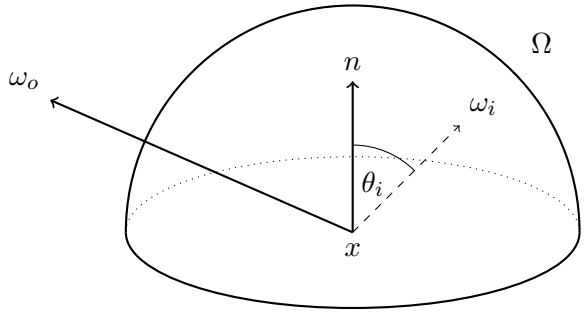
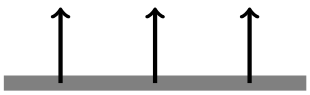


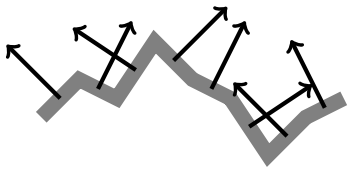
Figure 2.1: Illustrating the integral over all ω_i in the hemisphere Ω. Note that only reflection is illustrated. Transmission would enter the material instead, effectively taking the opposite hemisphere.

2.2 Bidirectional Reflection Distribution Function

The BRDF evaluates how much light gets reflected in an outgoing direction from a given incoming direction. On a given surface, the light gets reflected in a predictable manner, leading to only one possible outgoing direction for each wavelength. However, real materials have very complex surfaces. Unlike a glass pane, a mirror, and water that appear with perfectly smooth surfaces, a wooden plank, e.g., has a lot of unevenness and details to it if one looks at the microscopic level of the surface.
The former causes a specular reflection where light perfectly reflects according to the law of reflection (2.2), whereas the latter creates a diffuse reflection. An illustrative example is seeing each surface as made up of many small planes, each having its own different normal vector. The different normal vectors therefore lead to different outgoing directions of the light ray.



(a) Surface as we see or feel it with normal vectors of equivalent direction.



(b) Surface as seen at the microscopic level (example) with normal vectors pointing in various directions.

Modelling such surfaces in computer graphics is a complex endeavour with different strategies, some of which I will explain below.

2.2.1 Diffuse Reflection

The reflection on rough and uneven surfaces is called *diffuse reflection*. Incoming light bundles scatter into many different directions. Instead of using complex meshes to simulate this behaviour on the microscale, we can effectively approximate this effect by randomly choosing an outgoing direction. A visualisation is given in Figure 2.3a.

The *Lambertian Reflection* uses the hemisphere described by the surface normal to generate a random reflection. While this is a good first approximation, it is not physically accurate, as light appears equally bright from all viewing directions. The real world is much more complex.

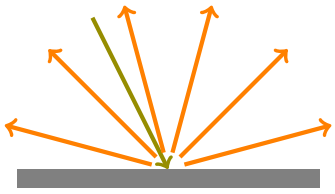
A more sophisticated method is the *Oren-Nayar*[12] reflectance model. It more accurately predicts reflections on rough surfaces and leads to more realistic looking materials.

2.2.2 Specular Reflection

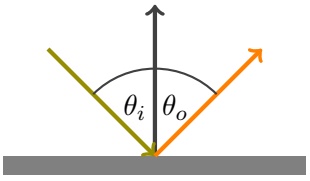
The reflection on smooth surfaces like mirrors, glass, and water is called *specular reflection*. Incoming light bundles at angle θ_i reflect perfectly along the surface normal with outgoing angle θ_o. This is the *law of reflection* [11]:

$$\theta_i = \theta_o \tag{2.2}$$

A visualisation thereof can be seen in Figure 2.3b.



(a) Diffuse reflection: rays scatter into all directions due to roughness on a microscopic level, as seen in Figure 2.2b.



(b) Specular reflection: rays get reflected perfectly.

Figure 2.3: Visualisation of the main BRDF for two different types of surfaces.

2.3 Bidirectional Transmission Distribution Function

A special case of the BRDF is the transmission of light into the material. This is called the bidirectional transmission distribution function (BTDF). Sometimes BRDF means both BRDF and BTDF. It is therefore important to understand the context.

2.3.1 Specular Transmission

If light gets transmitted into the material instead, it is called *specular transmission* or refraction. Material properties like *refractive indices* play an important role as they define in which direction the light is being transmitted [21]. Incoming light at angle θ_i relative to the surface normal in a material with refractive index η_i is then transmitted into another material with refractive index η_t at an angle θ_t relative to the inside surface normal. This is the *law of refraction*[11]:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

This refractive index depends on the wavelength of the light entering. Therefore, the dispersion effect is observed in materials with rather "large" divergent refractive indices. This is why during a sunny rain, a rainbow appears. All the different water droplets refract and disperse the incoming sunlight into many more outgoing directions. As a result, the law of refraction can be written to directly account for the wavelength dependency of λ:

$$\eta_i(\lambda) \sin\theta_i = \eta_t(\lambda) \sin\theta_t \tag{2.3}$$

Real materials are often a combination of both specular reflection and transmission (e.g., water or glass). This effect can be described with the Fresnel equations to compute the Fresnel reflectance [16]. This specifies the fraction of incoming light that is either reflected from the surface or transmitted into the material.

Rust-V only implements the Fresnel equations for dielectric materials (non-conducting materials like air, glass, and water) as they have a particularly simple form:

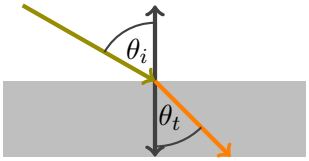


Figure 2.4: Specular transmission: rays refract into an optically more dense material ($\eta_i < \eta_t$).

$$
\begin{aligned}
r_{\parallel} &= \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}, \\
r_{\perp} &= \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}, \\
F_r &= \frac{1}{2}\left(r_{\parallel}^2 + r_{\perp}^2\right).
\end{aligned} \tag{2.4}
$$

Where $r_{\parallel}$ and $r_{\perp}$ describe the Fresnel reflectance for either parallel or perpendicular polarized light respectively. F_r then gives the Fresnel reflectance for unpolarised light. The Fresnel transmittance is the energy conserving opposite of $1 - F_r$.

A simple RGB model is not enough to model the transmitting behaviour accurately. Considering only 3 wavelengths can hardly simulate the dispersion effect correctly, as can be seen in Figure 5.8d (spectral vs. RGB rendering comparison).

2.4 Integration

Integrators are a major component in ray tracing as they compute the final incoming radiance by solving the rendering equation (2.1). They come in various forms and take different "paths". They can be described with the help of the following notation [2]:

- E is the eye,
- L is the light,
- D is diffuse reflection/transmission,
- G is glossy reflection/transmission, and
- S is specular reflection/refraction,

leading to following descriptions (with help of Regex):

- ray casting (by Arthur Appel): $E(D|G)L$,
- recursive ray tracing (by Turner Whitted): $E[S*](D|G)L$,
- path tracing (James Kajiya): $E[(D|G|S) + (D|G)]L$,
- radiosity (Cindy M. Goral): $ED * L$.

We can therefore immediately see which method traces which paths.

2.4.1 Path Tracer

To solve Equation (2.1) we can use the Monte-Carlo method to compute the integral over the hemisphere using random path creation. The basic algorithm can be described as follows [2]:

Monte-Carlo Path Tracing Algorithm

1. Create a ray from the camera at pixel (x, y). We initialise throughput $= 1$ and $L_o = 0$.
2. Trace the ray to the closest intersection in the scene.
   (a) If we hit no object, we return L_o.
   (b) If the object is an emitter, we return $L_o + \text{throughput} \cdot L_e$.
   (c) Otherwise we:
      i. Add direct illumination from emitters (if visible) $L_o = L_o + \text{throughput} \cdot L_{\text{direct}}$.
      ii. Sample the surface BRDF and set throughput $=$ throughput $\cdot f_r \cdot \cos\theta_i$.
      iii. Scatter the ray according to the surface BRDF and continue at Item 2.

One can see that this recursive algorithm terminates upon hitting either no object, or an emitter. This might be computationally expensive or never terminate in a scene where we have either

- no emitter present, or
- objects all around us with only a small emitter present.

This is why it is very beneficial to introduce a *maximum depth* for the recursion. Moreover, since the throughput might decrease so much that any further recursion proves to have too little impact, an early stop is unquestionably favourable.

To estimate the radiance of a pixel, we therefore have to shoot many rays from the camera to get a good sample space for the first hemispherical integral. We can express this estimation as follows: [23]:

$$I = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{2.5}$$

where

- I is the intensity of the pixel,
- N is the number of samples per paths for the pixel,
- X_i is a sample in the path space,
- $f(X_i)$ is the path contribution described in Section 2.4.1, and
- $p(X_i)$ is the probability density function (**PDF**) of the sample X_i.

More samples per pixel estimate the integral more precisely, reducing noise stemming from the Monte-Carlo approach. An example is given in Figure 2.5.

2.4.2 Spectral Path Tracer

In a spectral path tracer, we take into account the potential wavelength-dependant BRDF function as in transmissive materials, e.g. We can therefore reuse the Monte-Carlo path tracing algorithm 2.4.1 and adapt it to shoot only rays with a specific wavelength.
This results in a multitude of different paths, even though there may not have been a wavelength-dependant BRDF.

It is therefore better to actually trace the batch of wavelengths using the same path, splitting them only on dispersive materials. The estimator can then be expressed similar

to Equation (2.5), but with the additional wavelength dependency of λ^j at any given time and C chosen wavelengths [23]:

$$I = \frac{1}{N}\frac{1}{C}\sum_{i=1}^{N}\sum_{j=1}^{C}\frac{f(X_i, \lambda_i^j)}{p(X_i, \lambda_i^j)} \tag{2.6}$$

Tracing all possible wavelengths has a significant impact on performance on branching light paths, so we can again use random sampling to get a good approximation of the whole spectrum Λ. Tracing a batch of only C different $\lambda_i \in \Lambda$ still results in the correct estimation. For choosing the different λ_i it makes a lot of sense to just sample them uniformly from the interval $[\lambda_{\text{start}}, \lambda_{\text{end}}]$. This is a simple and unbiased approach with just one big flaw. While this leads to the correct sampling of all wavelengths involved, it also creates a lot of colour noise as the samples are not spread out but randomly chosen. It is very well possible that sampling a batch of wavelengths leads to very similar λ_i, missing potentially important $\lambda_j \in \Lambda$. This is why the **Hero Wavelength Spectral Sampling** method introduces the concept of the Hero Wavelength [23]. Instead of choosing all λ_i at random, we will only choose the first - the hero wavelength $\lambda_1 = \lambda_h$ - at random, placing the rest of the batch equidistantly around it. This hero wavelength will then also be used for path propagation decisions. The article defines a rotation function around Λ like the following:

$$r_i : \Lambda \to \Lambda \qquad r_i(\lambda_h) = \lambda_{\min} + (\lambda_h - \lambda_{\min} + \frac{i}{C}\bar{\lambda}) \ \textbf{mod} \ \bar{\lambda} \tag{2.7}$$

with $i \in 1, ..., C, \bar{\lambda} = \lambda_{\max} - \lambda_{\min}$.

This method is still randomly sampling the Λ space, but due to virtually covering the whole Λ in one batch, it results in less noise and more coherent convergence.

As the hero paper points out, the PDF to sample X_i and λ_i^j is

$$p(X_i, \lambda_i^j) = \frac{1}{C}\sum_{k=1}^{C} p(\lambda_i^k) p(X_i|\lambda_i^k) \tag{2.8}$$

If no spectral dependency was observed in the path generation, this would simply reduce to

$$\begin{aligned} p(X_i, \lambda_i^j) &= p(\lambda_i^j) * p(X_i|\lambda_i^j) \\ &= \frac{C}{|\Lambda|} * p(X_i|\lambda_h) \end{aligned} \tag{2.9}$$

as the probabilities of sampling each λ_i^j are equal, and only the hero wavelength is used for path creation (may even be omitted).

During a spectral scattering event, however, the light path would split up and the probability to choose the spatial sample X_i with a λ_i^k reduces to

$$p(X_i|\lambda_i^k) = \begin{cases} 1, & \text{if } \lambda_i^k \text{ generates this path} \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

Although (2.8) describes the correct PDF, it also means that one would essentially discard all other scattered paths, as only the hero wavelength is used for path propagation.

If the path creation is chosen with a maximum depth of d and the scattering event happens at a depth $\hat{d} < d$, the path computation for the other wavelengths is discarded instead of following the new path branches. This strikes me as a poor decision, as the path contribution of other wavelengths $\lambda \in \{\lambda_h, ...\lambda_C\} \setminus \lambda_h$ should not just be discarded. Rust-V's approach is to reuse the generated path by the light bundle until dispersion happens, after which the different path branches are traced separately, combining their contributions.

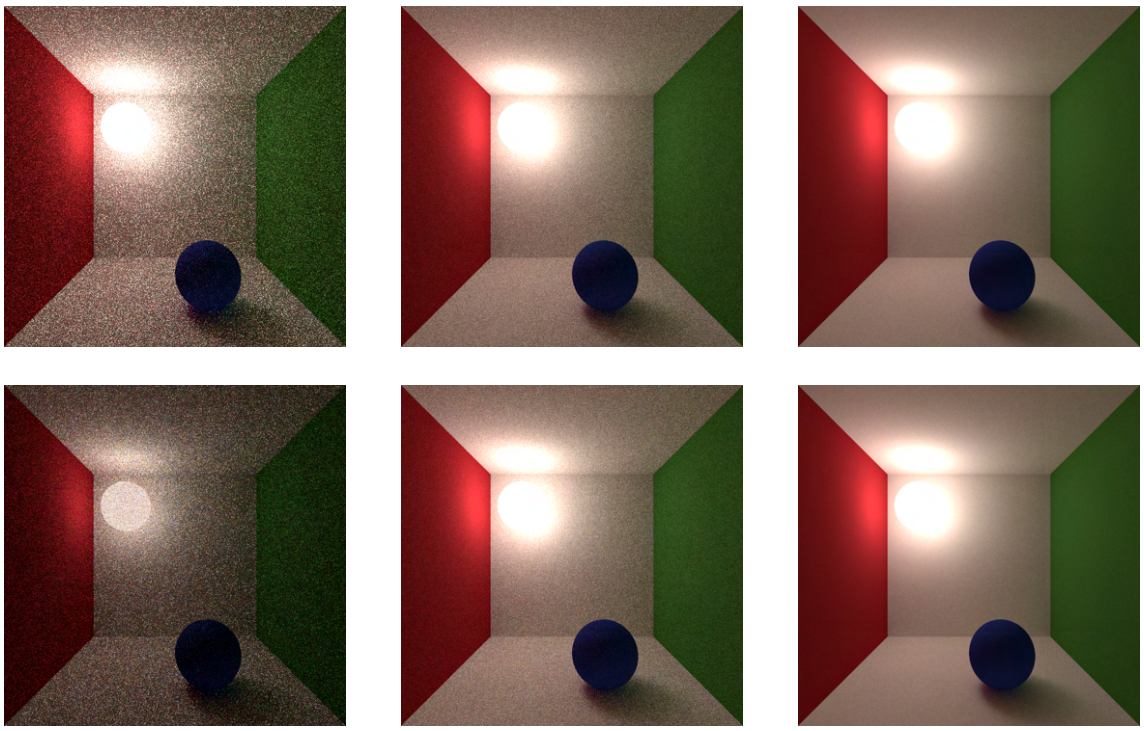


Figure 2.5: A comparison of both path (top row) and spectral path (bottom row) tracing with 4, 64 and 1024 samples per pixel (left to right). The effect of reduced brightness in spectral sampling becomes significantly visible in low samples renderings.

By the very nature of spectral sampling, tracing only a randomised subspace of Λ can result in accidentally ignored wavelengths. This reduces the overall brightness of the scene if either:

1. the random generator keeps giving unfavourable samples, or

2. not enough samples have been rendered.

With enough samples provided, the first problem has a low chance of happening. Therefore, this unwanted effect is especially visible in the second case. An example of this effect is given in Figure 2.5. As a comparison, a noise-free rendering can be seen in Figure 5.1.

2.5 Colour

The physical model describes colour as a continuous spectrum of waves, whereas computer graphics usually describe colours in RGB, CMY, HSV, XYZ or other formats. These tend to have only 3 components to describe a colour, e.g. red, blue and green in the RGB colour space. All these formats cannot describe the physical description in full, but get pretty close to it and are enough for most graphical applications. This thesis won't go into much detail on colour theory because it is such an extensive topic.

To break down a spectrum to common RGB, some operations are needed according to CIE 1931[7]. The main steps include:

1. convert to CIE XYZ colour space using colour matching functions which describe the observer's (human eye, e.g.) sensitivity towards the tristimulus values XYZ. An example of these functions is given in fig. 2.7.

2. convert XYZ to RGB with a reference white (e.g., D50 or D65)

One benefit of rendering in spectral colours is the ability to demonstrate metamerism. Metamerism is the perception of different spectra having equal colour to the observer due to different illuminants. The human eye, e.g., has three types of cone cells which reduce a perceived light spectrum into the tristimulus values of red, green, and blue. These cones have overlapping sensitivities, as shown in Figure 2.6.

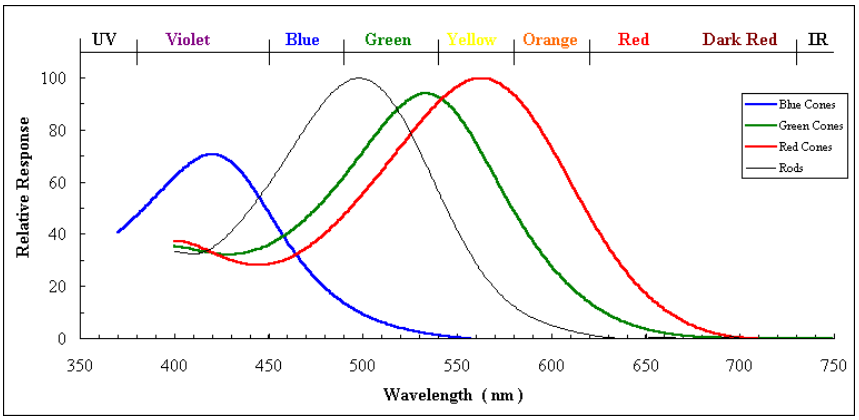


Figure 2.6: The spectral sensitivity of the eye [3]. Each cone covers a large spectrum with different sensitivity.

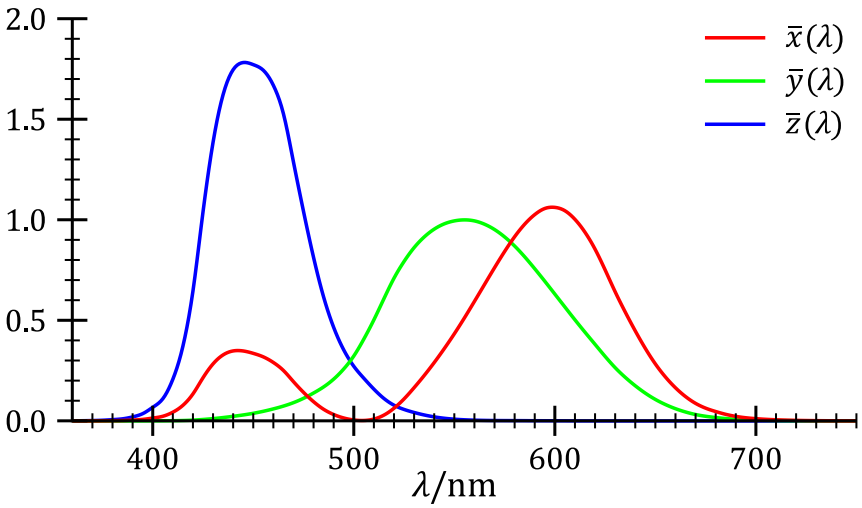


Figure 2.7: The CIE 1931 XYZ colour matching functions[4]. These are required to convert spectral colour data to XYZ format.

Chapter 3

Implementation

While ray tracers are typically written in C++ [5], I opted for the rather new language Rust. It is a systems programming language as well, but comes with a promise of automatic memory management and safety (including concurrency). This is due to heavy analysis during compilation, where the compiler enforces a set of rules programmers have to follow, unless we want to go *unsafe*.

As a reference implementation in C++, I followed the PBRT book [16] and source code [14][15]. I did, however, diverge from many things to incorporate Rust patterns and idioms and drop most space transformations for simplicity.

3.1 Colour

Rust-V supports the colour types `Srgb`, `Xyz` and `Spectrum`. Important to note is that these colour types are described as an array of either 3 or 36 floats in ascending order of wavelength. This effectively requires both `Srgb` and `Xyz` data to be in inverted order, as a low wavelength corresponds to a blue wavelength. To put it more clearly, the typical $(red, green, blue)$ gets interpreted as $(blue, green, red)$ (a pitfall that lead to long debugging hours). The conversion between those types follow

$$\begin{aligned} \texttt{Srgb} &\rightarrow \text{linear Srgb} \rightarrow \texttt{Xyz} \\ \texttt{Xyz} &\rightarrow \text{linear Srgb} \rightarrow \texttt{Srgb} \\ \texttt{Spectrum} &\rightarrow \texttt{Xyz}. \end{aligned}$$

Only the D65 white space is supported, and all colour transformations depend on this assumption.

All colour data has been taking from BabelColor [8], which contains data for 24 different colours like red, blue sky and foliage, to name a few. Their spectrum measurements range from 380nm to 730nm in steps of 10nm, reaching the 36 distinct wavelengths for the `Spectrum` type.

Rust-V allows defining the colour scheme at compile time, as dynamic interpretation would lead to a loss in overall computation speed. As explained in Section 2.4.2, batch tracing is implemented for spectral rendering, amounting to a compile-time constant of `const PACKET_SIZE: usize = 4` for auto-vectorization of the code. Both `Srgb` and `Xyz` have only 3 values, so their packet size is automatically reduced to 4.

3.2 Geometry

Rust-V supports following geometries:

- axis aligned bounding boxes (AABBs),
- spheres (no ellipsoids) and bubbles (made out of two spheres),
- planes and disks, and
- meshes.

However, it does not support texture mapping with (u, v) coordinates. The geometry interface describes the basic functions:

```
trait Geometry {
    // If we cannot do a containment test, we simply return [None].
    fn contains(&self, p: Vec3) -> Option<bool>;

    // Returns the bounding box.
    fn bounds(&self) -> Aabb;

    // Returns intersection information, if any.
    fn intersect(&self, ray: Ray) -> Option<Intersection>;

    // Returns whether an intersection occurs.
    fn intersects(&self, ray: Ray) -> bool;
}
```

where the intersection information consists of

```
struct Intersection {
    point: Vec3,
    normal: Vec3,
    incoming: Vec3,
    t: Float,
}
```

This information is needed by the integrators (Section 3.6) for the BRDF (Section 3.4) calculation.

3.3 Acceleration structure

To accelerate the ray tracing, particularly with meshes, a bounding volume hierarchy has been implemented, following the surface area heuristic (SAH) [22]. It uses the indices of geometries (be it triangles in a mesh or all objects in a scene, e.g.) to do quick intersection tests.

3.4 Bidirectional Reflectance Distribution Functions

To calculate the BRDF function of the rendering equation (2.1) we need to have an interface for different materials. While Rust-V is laid out for spectral ray tracing, it also comes with a generic path tracer (Section 3.6), meaning the interface must support both modes. Both BRDF (reflective) and BTDF (transmissive) functions have been united as `BxDF`.

To make use of vectorization in compiled code, I introduce methods to calculate a packet of multiple wavelengths at the same time with a default size of `const PACKET_SIZE: usize = 4`.

```
trait BxDF {
    // Returns the type of the bxdf.
```

```
    fn flag(&self) -> BxDFFlag;

    // Evaluates the BxDF.
    fn evaluate(&self, incident: Vec3, outgoing: Vec3) -> Spectrum;

    // Evaluates a packet of indices.
    fn evaluate_packet(..., indices: &[usize; PACKET_SIZE]) -> [Float; PACKET_SIZE];

    // Evaluates one wavelength only.
    fn evaluate_lambda(..., index: usize) -> Float;

    // Samples and evaluates the BxDF.
    // E.g. samples a random incoming direction from the
    // unit hemisphere for diffuse reflection.
    fn sample(&self, out: Vec3, sample: Vec2) -> Option<BxDFSample>;

    // Samples a packet.
    // A transmissive material introduces multiple incoming
    // directions here due to wavelength dependency.
    fn sample_packet(...) -> ...;

    // Samples one wavelength only.
    fn sample_lambda(...) -> ...;

    // Computes the PDF for the pair of directions.
    fn pdf(&self, inc: Vec3, out: Vec3) -> Float;
}
```

All BxDF do calculations inside their local space for mathematical simplifications inside the code. For this reason, Rust-V introduces the super-set called `BSDF` (bidirectional scattering distribution function), which wraps around a list of BxDFs and provides the necessary conversion between global and local space.

3.5 Objects

A scene object consists of both a geometry and a BSDF. These properties fully to describe a red cube, e.g., or a mesh with glass-like material.

However, a special case are scene objects that emit light. To be qualified as an emitter, the scene object must also have the capability of being sampled for direct illumination. This sampling is what leads to *soft shadows* inside the scene, greatly enhancing the perceived realism. The emitter interface is only implemented for the geometries `Point`, `Disk` and `Sphere`, whereas the last two are the only physically plausible ones.

3.6 Integrator

The integrator interface is fairly simple and consists of only one method:

```
pub trait Integrator: Send + Sync {
    /// Calculates the rendering equation and adds the result into the pixel.
    /// The pixel is averaging all results without specific filters.
    fn integrate(&self, scene: &Scene, primary_ray: Ray, pixel: &mut Pixel);
}
```

Rust-V comes with implementations for the Whitted model, path- and spectral path tracing.

3.7 Checkpointing

Rust-V comes with a checkpointing system that allows pausing the current rendering computation and continuing at a later time. The trapped signals are:

- SIGUSR1: saves the current rendering as a PNG, appending the number of passes done to the file name,
- SIGURS2: saves a checkpoint of the program state, and
- SIGINT and SIGTERM: corresponds to both actions described above and exiting the program.

The checkpoint is compressed using LZ4, as it might consume a lot of disk space, especially when computing high resolution spectral renderings (hundreds of megabytes).

Chapter 4

Testing Method

The speed of convergence of both *random sampling* and *hero wavelength sampling* is progressively compared frame-by-frame using various quality metrics in comparison to a *target image* (rendering with many passes, 20'000 e.g.). Each image is rendered with a resolution of 512×512 pixel.

The used image format is 16-bit RGB. Therefore all pixels consist of 16-bit red, green and blue values (r, g, b). These pixels can be represented as a 3D-vector, each colour component describing one dimension. The metric comparing between two pixels p, q can then be formulated using the standard Euclidean distance on this RGB space (also called the L_2-norm):

$$\Delta(p, q) = |q - p| = \sqrt{(q_r - p_r)^2 + (q_g - p_g)^2 + (q_b - p_b)^2} \tag{4.1}$$

4.1 Notation

Images are described as variables with either $\mathbf{x}$ or $\mathbf{y}$ and their size with $N \cdot M$ pixels. The pixels of the images are written as $\mathbf{x}_{ij}$.
The *average* of $\mathbf{x}$ is:

$$\mu_{\mathbf{x}} = \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{j=0}^{N} \mathbf{x}_{ij} \tag{4.2}$$

The *variance* of $\mathbf{x}$:

$$\sigma_{\mathbf{x}}^2 = \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{j=0}^{N} (\mathbf{x}_{ij} - \mu_{\mathbf{x}})^2 \tag{4.3}$$

And the covariance of $\mathbf{x}$ and $\mathbf{y}$ of same size:

$$\sigma_{\mathbf{xy}} = \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{j=0}^{N} (\mathbf{x}_{ij} - \mu_{\mathbf{x}}) \cdot (\mathbf{y}_{ij} - \mu_{\mathbf{y}}) \tag{4.4}$$

Additionally, the maximum possible signal power in images is denoted with L.
In 16-bit RGB images, the white pixel fulfils this property and L is equal to the dynamic range of the pixel values. It has `u16::MAX` $= 2^{16} - 1 = 65535$ in all colour channels.
The maximum possible signal is therefore the magnitude of the white pixel vector:

$$L = \left| \begin{pmatrix} \texttt{u16::MAX} \\ \texttt{u16::MAX} \\ \texttt{u16::MAX} \end{pmatrix} \right| = 113509.9497 \tag{4.5}$$

4.2 Quality indices and metrics

4.2.1 Mean Squared Error

The Mean Squared Error (MSE) is a simple quality index with a clear mathematical interpretation of error in absolute terms. It measures the average of the squared pixel error compared to the target image. Due to the nature of the squared distance value, pixels with large errors contribute significantly to the overall MSE measure. Consequently, we expect the rendering process to start with a high MSE value before quickly decreasing as the rendered image at a given iteration converges towards the target image.

The value of this error metric is more relevant on the mathematical modelling side, whereas the perceived visual quality is poorly represented via the MSE [19].

$$\mathrm{MSE}(\mathbf{x}, \mathbf{y}) = \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{j=0}^{N} (\mathbf{x}_{ij} - \mathbf{y}_{ij})^2 \tag{4.6}$$

A smaller value means smaller error and more similarity with the target image.

4.2.2 Peak Signal to Noise Ratio

The Peak Signal to Noise Ratio (PSNR) describes the ratio between the maximum possible signal power with the actual error. Different sources may have a high dynamic range, therefore PSNR is often described on the logarithmic decibel (dB) scale. In our case, the highest signal power is the white pixel described above. Compared to MSE, PSNR is closer to human perceived image quality, but still describes the errors in absolute terms [19].

$$\begin{aligned} \mathrm{PSNR}(\mathbf{x}, \mathbf{y}) &= 10 \cdot \log_{10} \left(\frac{L^2}{\mathrm{MSE}(\mathbf{x}, \mathbf{y})} \right) \\ &= \underbrace{20 \cdot \log_{10}(L)}_{=101.1007} - 10 \cdot \log_{10} \left(\mathrm{MSE}(\mathbf{x}, \mathbf{y}) \right) \end{aligned} \tag{4.7}$$

A higher value means smaller error and more similarity with the target image.

4.2.3 Structural Similarity Index Measure

Another approach to test image quality is the Structural Similarity Index Measure (SSIM), which is a more sophisticated approach at describing similarity or perceived quality[19]. It tries to evaluate perceived differences in the image structure, taking into account changes in luminance and contrast. SSIM uses some pre-defined constants where:

- $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ are a small bias for division with small denominators,
- L is the above-mentioned dynamic range of the pixels (equal to the maximum signal power), and
- $k_1 = 0.01$, $k_2 = 0.03$ are some default values.

$$\mathrm{SSIM}(\mathbf{x}, \mathbf{y}) = \frac{(2\mu_{\mathbf{x}}\mu_{\mathbf{y}} + c_1)(2\sigma_{\mathbf{xy}} + c_2)}{(\mu_{\mathbf{x}}^2 + \mu_{\mathbf{y}}^2 + c_1)(\sigma_{\mathbf{x}}^2 + \sigma_{\mathbf{y}}^2 + c_2)} \tag{4.8}$$

A higher value means less error and more similarity.

4.3 Ratio

The aim of the above quality metrics is to assess a possible benefit of hero wavelength sampling over random wavelength sampling. To this end, one may compute the ratio of the former to the latter. This provides a nice overview how both sampling methods relate to each other.

Do note however, that both PSNR and SSIM treat higher values as being more accurate towards the target image! Therefore these values have to be inverted prior to comparison.

$$\text{MSE ratio} = \frac{\text{hero MSE}}{\text{random MSE}} \tag{4.9}$$

$$\text{PSNR ratio} = \frac{(\text{hero PSNR})^{-1}}{(\text{random PSNR})^{-1}} = \frac{\text{random PSNR}}{\text{hero PSNR}} \tag{4.10}$$

$$\text{SSIM ratio} = \frac{(\text{hero SSIM})^{-1}}{(\text{random SSIM})^{-1}} = \frac{\text{random SSIM}}{\text{hero SSIM}} \tag{4.11}$$

Using this definition the following holds:

$$\text{ratio} = \begin{cases} < 1 & \text{hero sampling is better} \\ > 1 & \text{random sampling is better} \\ 1 & \text{equivalent quality} \end{cases} \tag{4.12}$$

Chapter 5

Results

This chapter analyzes the convergence rate of both hero and random sampling and talks about the benefit of spectral path tracing over normal path tracing. All target images have been rendered using the hero sampling approach with 20'000 passes per pixel. These target images are uses as a ground base to calculate the convergence rates.

Furthermore, there will be some argument made for using spectral vs. RGB representation.
The chapter closes with a visual analysis of spectral-path tracing vs non-spectral-path tracing.

5.1 Convergence

The Cornell box consists of Oren-Nayar-diffuse surfaces and forms the basis for all subsequent renderings. The left wall is red, the right wall is green, and all other surfaces are white. The spherical emitter in the top left corner of the box emits strong, white light.

5.1.1 Cornell box

The following variant of the Cornell box (Figure 5.1) features a blue sphere on the ground, showcasing soft-shadows. The ray-tracing depth is set to 8 light bounces.

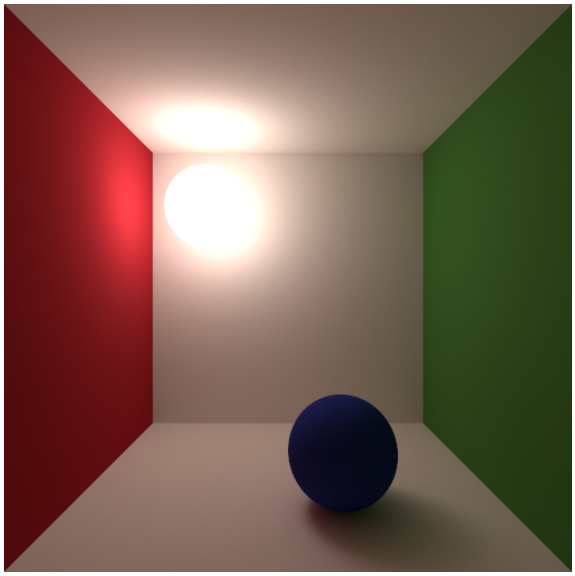
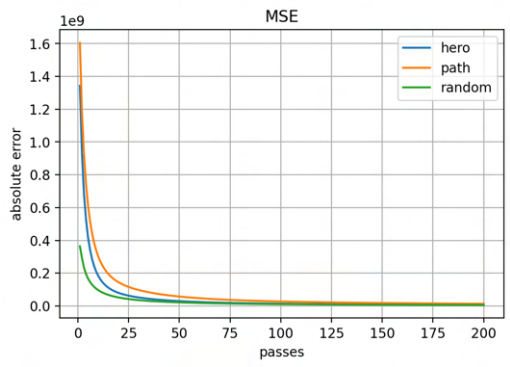
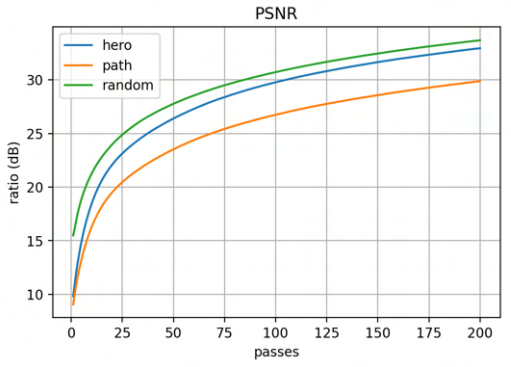


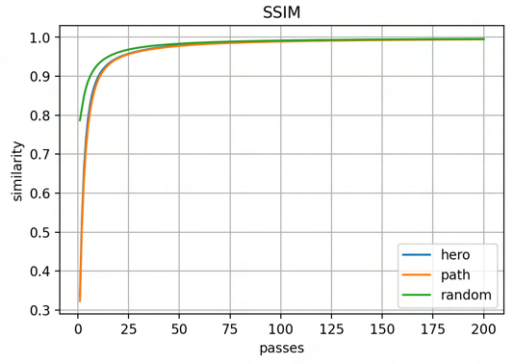
Figure 5.1: The Cornell scene after 20'000 passes per pixel.



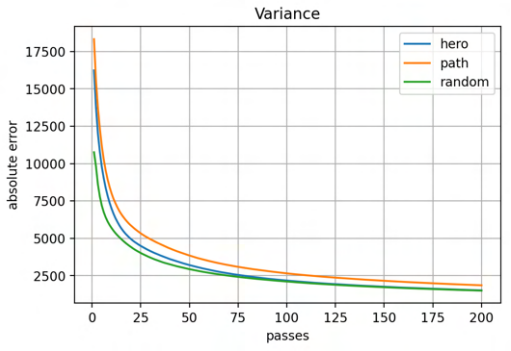
(a) MSE: original path tracing is best without spectral dependency. However, hero sampling improves rapidly.



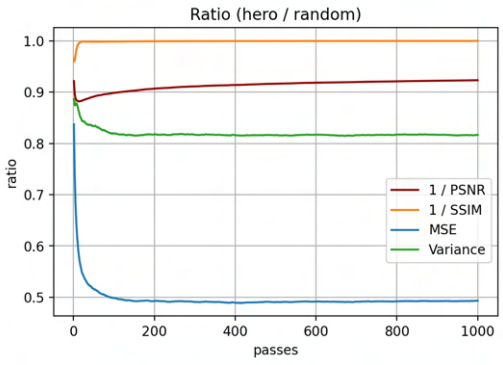
(b) PSNR: original path tracing and hero sampling are quite close.



(c) SSIM: original path tracing is best. Both hero and random sampling start poorly due to reduced spectrum.



(d) Variance: original path tracing proves best again. Hero sampling has less dispersion than random sampling.



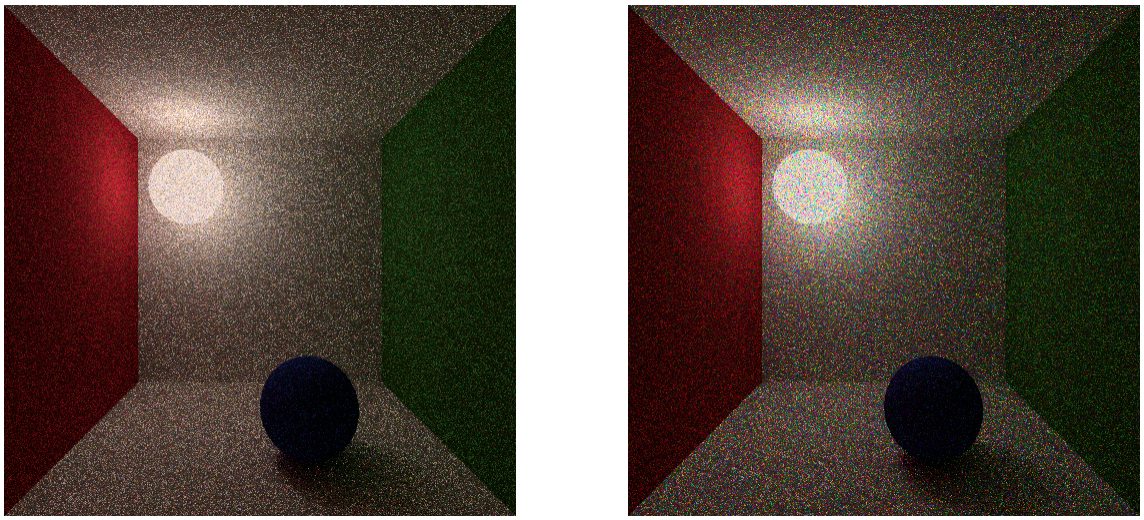
(e) Ranges (legend top to bottom): $(0.882, 0.923)$, $(0.960, 1.000)$, $(0.489, 0.837)$, $(0.815, 0.886)$.

Figure 5.2: Error plots for the Cornell scene.

In this scene there is no spectral dependency as all materials are diffuse Oren-Nayar models. No path-branching is happening inside the scene. Therefore, reducing the discretised spectrum leads to the undesirable effect of introducing more colour noise. This effect can be seen in Figure 5.2, which contains both hero and random sampling and the simpler path tracing model. Therefore it is encouraged to trace the whole spectrum together in scenes with no spectral dependency.

Apart from SSIM (Figure 5.2c), all error metrics consistently show less noise with hero sampling as compared to random sampling. In Figure 5.2e, the error ratio of hero to random sampling is visualised. After 100 passes per pixel, especially the MSE is substantially less with being below 0.5. Therefore, the better distribution of wavelength packets inside hero sampling shows a large benefit.

As hinted with Figure 2.5, both spectral tracing methods start with reduced brightness before converging towards the target image in Figure 5.1. That is the reason why the SSIM starts with a low value of 0.336 before climbing to a high similarity of above 0.9. Since the SSIM ratio remains so close to 1.0, this metric indicates no significant benefit of hero over random sampling. Visually, in a low range of passes per pixel, one may perceive the hero sampling to be much more appealing to the eye due to less colour noise. An example is shown in Figure 5.3.



(a) Hero sampling at 4 passes per pixel.

(b) Random sampling at 4 passes per pixel.

Figure 5.3: The random sampling clearly suffers from significant colour noise compared to hero sampling.

5.1.2 Refracting spheres

This scene consists of 9 strongly refracting glass spheres inside a Cornell box (Figure 5.4). The high number of refracting materials poses a challenge for physical rendering. The scenery also showcases the dispersion effect very well. The ray-tracing depth is set to 10 light bounces.

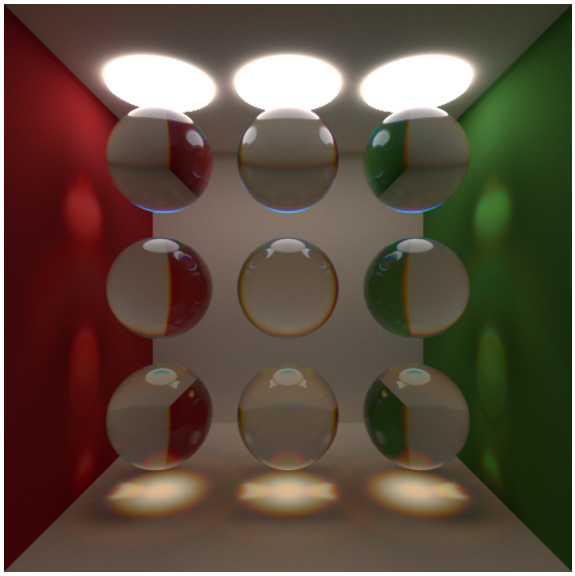
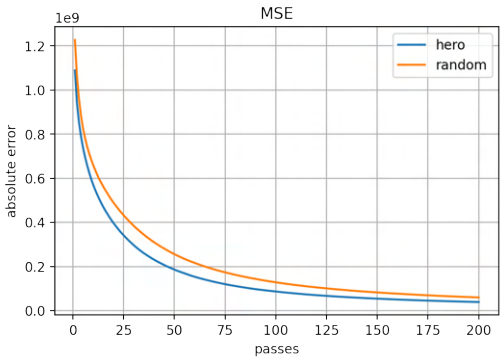


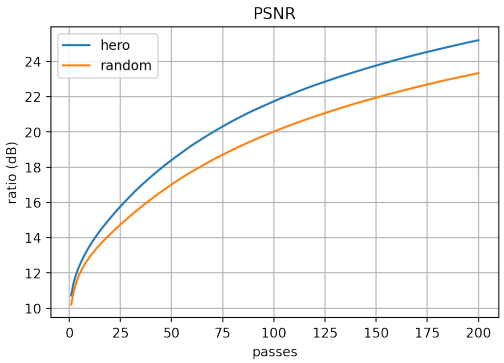
Figure 5.4: The refracting spheres scene after 20'000 passes per pixel.

Similarly to Figure 5.2, Figure 5.5 consistently converges fast in all metrics. The variance (Figure 5.5d) for random sampling forms an exception here, where the second pass over all pixels resulted in overtaking hero sampling for some time. This effect can happen, as in the early phases, each pass adds a large weight to the current image. Randomly choosing wavelengths can therefore be "lucky" to hit strongly weighted emissions which are similar to the target image. However, the reduced discretised spectrum also has a lot of trouble to find light sources in this scene, as all the refracting spheres pose a challenge.

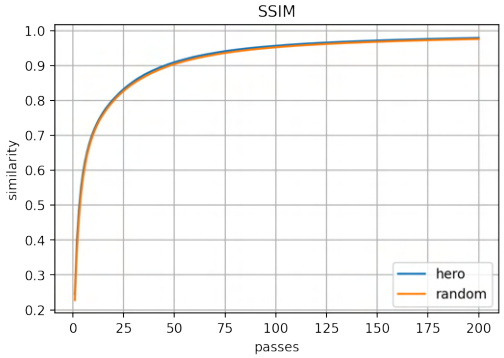
All the path-branching contributes to colour noise, therefore many passes per pixel are needed to have a good noise-free image. This effect can particularly be seen in the PSNR (Figure 5.5b), which is still in a strong climb after 200 passes per pixel. Similarly, the variance (Figure 5.5d) is still continuously decreasing.



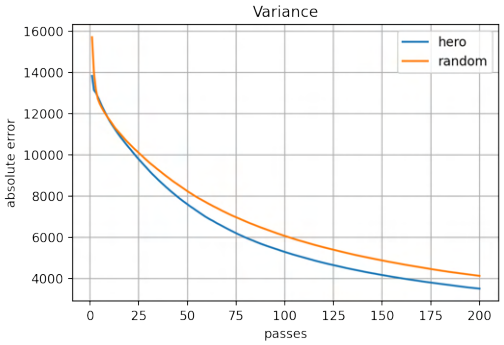
(a) MSE: hero sampling has less error than random sampling.



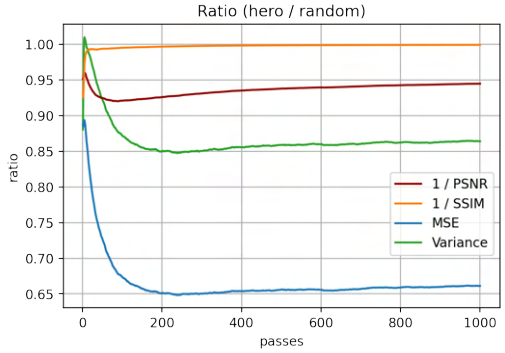
(b) PSNR: hero sampling delivers less corrupting noise.



(c) SSIM: both sampling methods prove to be equal.



(d) Variance: anomaly after 2 frames, but hero sampling again proves less dispersed.



(e) Ranges (legend top to bottom): $(0.920, 0.961)$, $(0.926, 0.999)$, $(0.648, 0.894)$, $(0.848, 1.010)$.

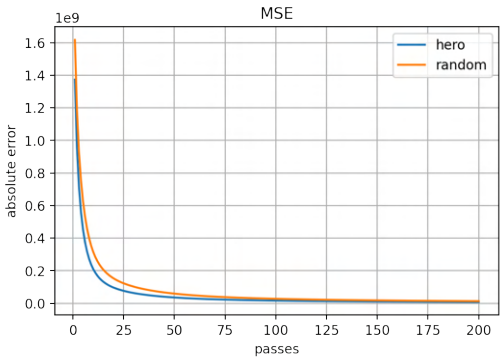
Figure 5.5: Error plots for the refracting spheres scene.

5.1.3 Chinese dragon mesh

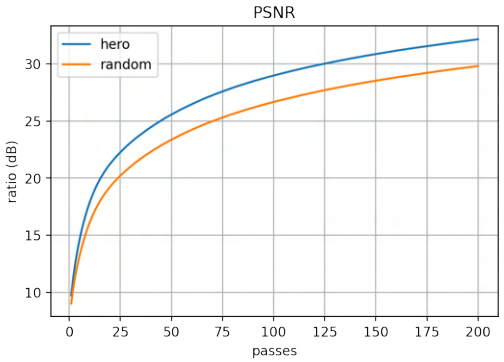
The dragon mesh consists of 1'180'060 vertices and 2'349'078 faces (after exporting with Blender), showcasing the power of the implemented acceleration structure.

Figure 5.6: The dragon scene after 20'000 passes per pixel.

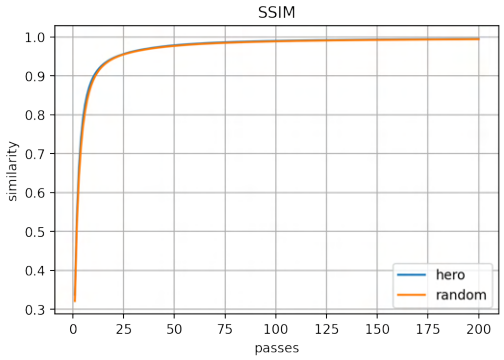
Similar to the Cornell box, we can see that the hero sampling shows improvement in all metrics in Figure 5.7. Especially in MSE (Figure 5.7a), the ratio towards random sampling consistently stays below 0.6 after ca. 50 passes per pixel. The PSNR is also still in a strong climb, similar to Figure 5.5b. Hero sampling again proves to deliver less corrupting noise. Introducing the refractive dragon is a big contributor to colour noise due to path-branching. However, compared to the refracting spheres in Section 5.1.2 it takes less volume inside the scene. Therefore the wavelength packets are less likely to hit the dragon and split into discrete paths. Consequently, the resulting colour noise is weaker than with the refracting spheres. This effect is clearly visible due to the lower error scores.



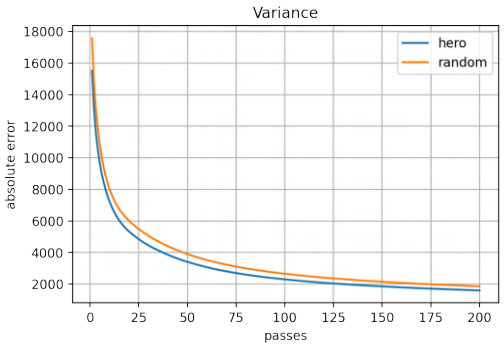
(a) MSE: hero sampling consistently has less error.



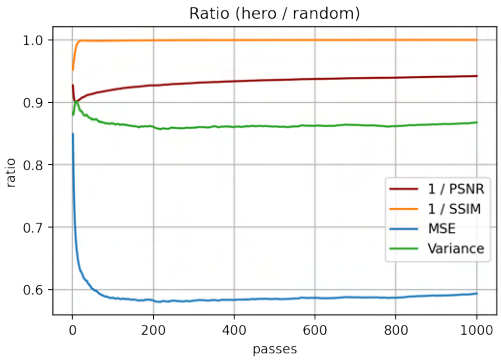
(b) PSNR: hero sampling has less noise than random sampling.



(c) SSIM: both sampling methods prove to be equal.



(d) Variance: hero sampling again proves to be less dispersed.

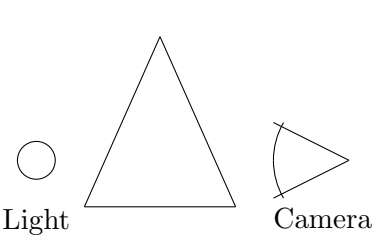


(e) Ranges (legend top to bottom): $(0.901, 0.942)$, $(0.952, 1.000)$, $(0.580, 0.849)$, $(0.857, 0.902)$.

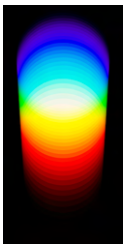
Figure 5.7: Error plots for the dragon scene.

5.2 Spectral vs. RGB

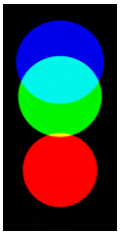
The most prominent example to show the benefits of spectral vs. RGB rendering is looking at the dispersion of a prism. As Rust-V is a backwards-tracing engine, it is very inefficient to create a projection from a light source to the wall through a prism. The statistical probability of the Monte-Carlo method to create this path in opposite direction starting from the wall is rather low. Instead, a much simpler example shows the benefit quite immediately by looking at the light source directly through the prism.



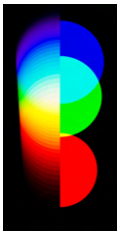
(a) The prism scene setup.



(b) Spectral rendering



(c) Srgb rendering



(d) Comparison

Figure 5.8: Looking at a spherical light source through a prism.

One can see the 36 distinct circles stemming from discretised Spectrum in Figure 5.8b. Figure 5.8c however, delivers much less detail of dispersion due to only having 3 distinct wavelengths. The RGB model uses the wavelengths $(0.4358, 0.5461, 0.7)\mu$m. Interesting to see here is the fading at both ends of the spectral rendering, as according to the CIE XYZ colour matching functions[24] (Figure 2.7).

5.3 Comparison of spectral-path and path tracing

In scenes that do not have any spectral dependency on surfaces, there is no difference between the path and spectral-path tracing method with regard to paths taken, as no splitting of light packets is happening. The only difference here is that the former uses the whole `Spectrum` for colour calculation, whereas the latter works with subset thereof. This reduced spectrum leads to a slower convergence rate, as seen in Figure 5.2.

It gets interesting, however, when transmitting materials like glass get introduced into the scene. As the light packets split when hitting such a medium, the rainbow effect can be observed.

The scenes of Section 5.1.2 and Section 5.1.3 have been retaken and compared against the simpler path tracing method in Figure 5.9 and Figure 5.10. Additionally, a focus has been set on some interesting areas of these renderings. These focused scenes have a more intense lighting to see the dispersion effect more clearly.

These examples showcase the benefit of spectral rendering as opposed to only path tracing. The missing dispersion in path tracing is a big loss in physical accuracy in these scenes.

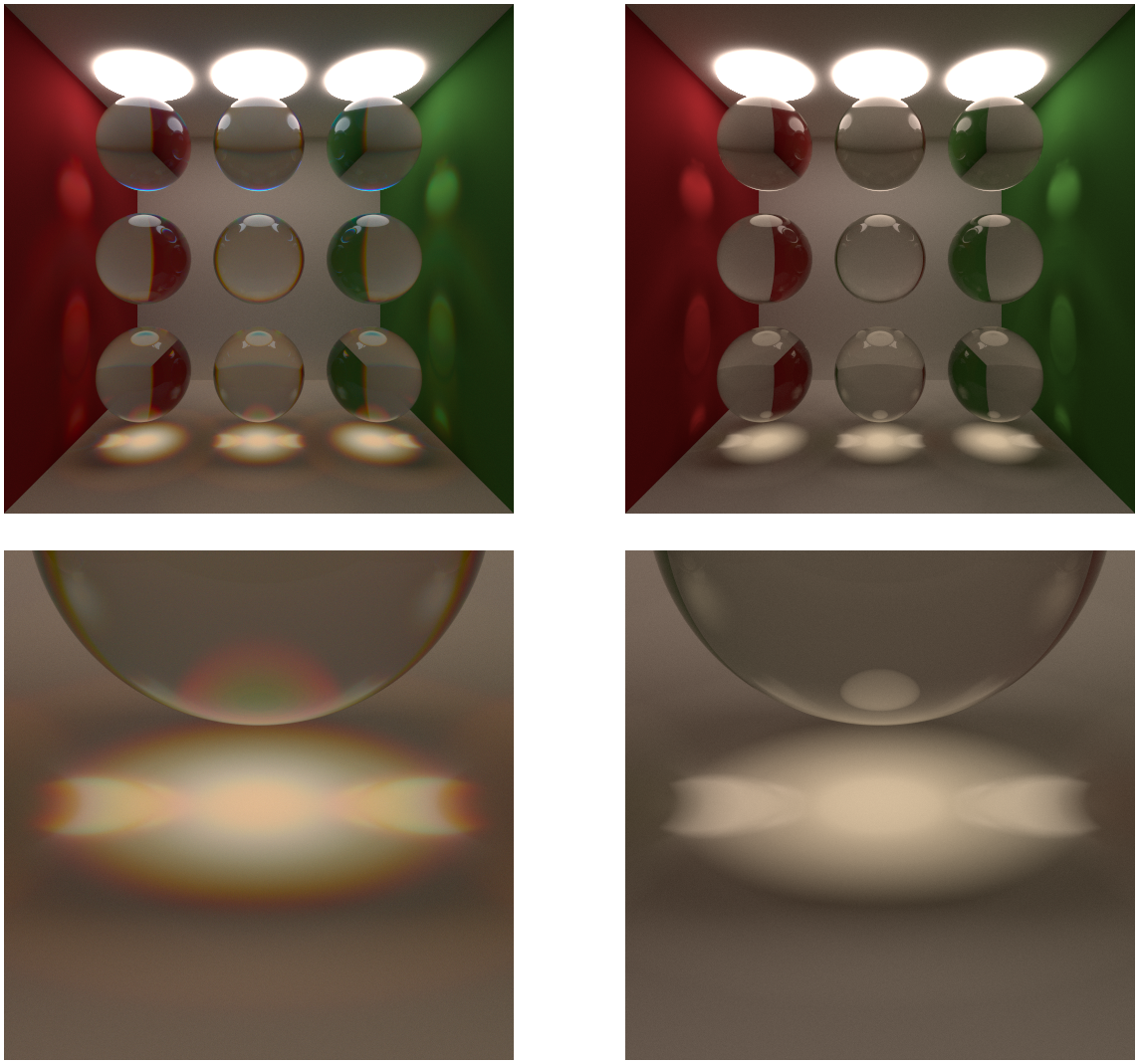


Figure 5.9: Refracting spheres: spectral-path (left) and path (right) tracing at 20'000 passes per pixel each.

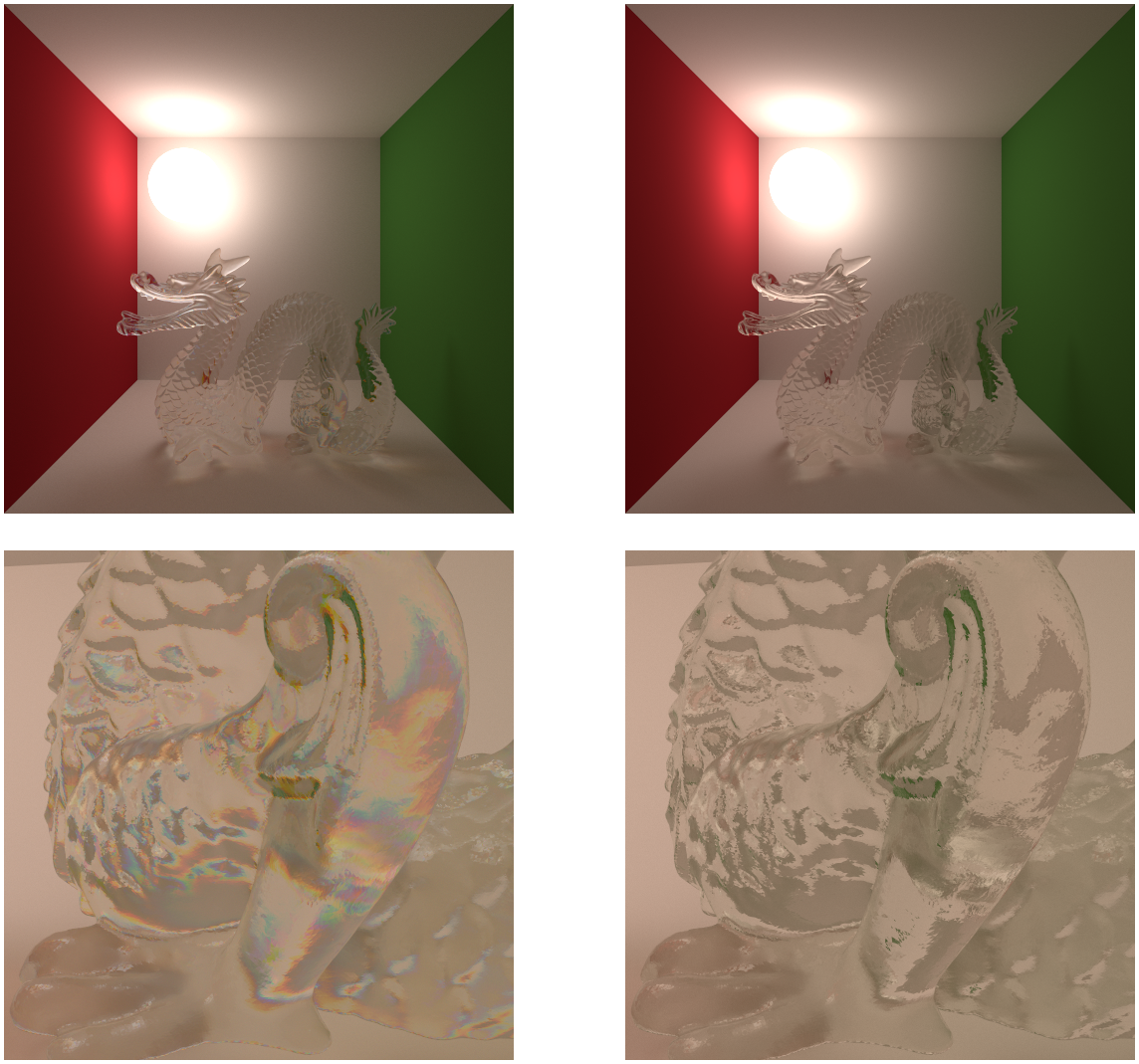


Figure 5.10: Chinese dragon: spectral-path (left) and path (right) tracing at 20'000 passes per pixel each.

Chapter 6

Conclusion

6.1 Summary

This work demonstrates two different methods of choosing a reduced discretised spectrum. The convergence rate of hero wavelength sampling and random sampling has been assessed on four different quality metrics. The results consistently prove the hero wavelength sampling technique to outperform random sampling. It has been observed that equidistant distribution of the reduced spectrum leads to significantly less colour noise as the rendering progresses. Empirical error computations have shown the hero sampling to converge up to

- 2.04x faster in Mean Squared Error,
- 1.13x faster in Peak Signal to Noise Ratio,
- 1.23x faster in variance, and
- having virtually no effect in the Structural Similarity Index Measure.

The analysis shows that random sampling results in significant colour noise as opposed to hero wavelength sampling, indicating a reason why the latter method should be favoured.

Physically based spectral rendering promises a bright future in computer graphics. The approach taken showcases which benefits towards photo-realism can be gained by incorporating the simulation of light dispersion. A new rendering engine, Rust-V has been created with guidelines from the PBRT book. Rust-V creates physically accurate rendering even in very simple scenes.

6.2 Outlook

The used colour space is currently hard-coded in Rust-V and would need an overhaul to be more generic. This would allow choosing different standard illuminants to simulate different lightings.

While the current implementation follows the principles of unbiased backwards path tracing, the performance thereof is rather low. There exist techniques like bidirectional path tracing, photon mapping, multiple importance sampling and more, which increase the performance drastically for representing caustics and dispersion effects. Rust-V needs a lot of samples per pixel as it depends on randomised paths. Bad random number sequences may therefore show diminished caustics. The approach in this work follows the principles of unbiased path tracing in order to easily and correctly represent the physics of light bundles.

To further enhance the representation of physical lighting, advanced methods like volumetrics and subsurface scattering can be implemented. The source code of Rust-*V* can easily be adapted to include more features using the `BxDF` and `Integrator` traits. Introducing more accurate camera models is also possible by implementing the `Camera` trait.

Rust-*V* can only be used to render static images. To incorporate animations and moving objects, it would require a rewrite of the `Ray` system. Simulating the shutter speed of a camera is therefore impossible unless the whole scene remains in place without moving objects.

Furthermore, a major performance drawback of Rust-*V* is the CPU-based implementation. The renderer cannot be accelerated using GPUs. This decision has been made to simplify development, debugging and keep native code safety delivered-on by the standard Rust compiler. An upcoming crate for Rust-native GPU shaders is *rust-gpu*[18] which promises GPU-acceleration without leaving the Rust language. However, at the time of concluding this thesis, the project is still in an early development stage. Some core library features required by Rust-*V* are not yet working (e.g. algebraic enums and iterators). Nonetheless, *rust-gpu* is a very interesting project that shows a lot of potential.

Modern technology advanced immensely as well, such that real-time ray-tracing is supported by hardware such as Nvidia RTX[13] and recently also AMD RDNA2[6]. Making use of hardware acceleration would greatly increase the performance of Rust-*V*. Introducing this feature would require a deep dive into the pipelines of various graphics APIs.

List of Figures

Bibliography

[1] URL: https://www.rust-lang.org/.

[2] URL: https://graphics.stanford.edu/courses/cs348b-01/course29.hanrahan.pdf (visited on 23/10/2021).

[3] URL: https://www.unm.edu/~toolson/human_cone_response.htm (visited on 23/01/2022).

[4] URL: https://commons.wikimedia.org/wiki/File:CIE_1931_XYZ_Color_Matching_Functions.svg (visited on 26/01/2022).

[5] URL: https://github.com/search?q=ray+tracer (visited on 25/10/2021).

[6] *AMD RDNA2*. URL: https://www.amd.com/en/technologies/rdna-2 (visited on 26/01/2022).

[7] *CIE 1931 color space*. URL: https://en.wikipedia.org/wiki/CIE_1931_color_space (visited on 14/01/2022).

[8] *ColorChecker RGB and spectra*. URL: https://babelcolor.com/colorchecker-2.htm (visited on 25/10/2021).

[9] Fabrizio Duroni. 'Spectral Clara Lux Tracer: physically based ray tracer with multiple shading models support'. In: (2016).

[10] *Hyperion*. URL: https://www.disneyanimation.com/technology/hyperion/ (visited on 21/02/2022).

[11] Rudolf Langkau, Wolfgang Scobel and Gunnar Lindström. 'Physik kompakt 2: Elektrodynamik und Elektromagnetische Wellen'. In: Springer Verlag, 2002, p. 247. ISBN: 978-3-642-56016-3. DOI: 10.1007/978-3-642-56016-3.

[12] S.K. Nayar and M. Oren. 'Generalization of the Lambertian Model and Implications for Machine Vision'. In: *International Journal on Computer Vision* 14.3 (Apr. 1995), pp. 227–251.

[13] *NVIDIA RTX Ray Tracing*. URL: https://developer.nvidia.com/rtx/raytracing (visited on 26/01/2022).

[14] *PBRT Version 3 Source Code*. URL: https://github.com/mmp/pbrt-v3 (visited on 25/10/2021).

[15] *PBRT Version 4 Source Code*. URL: https://github.com/mmp/pbrt-v4 (visited on 25/10/2021).

[16] Matt Pharr, Wenzel Jakob and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. URL: https://pbr-book.org/ (visited on 25/10/2021).

[17] *RenderMan*. URL: https://renderman.pixar.com/product (visited on 21/02/2022).

[18] *rust-gpu crate*. URL: https://github.com/EmbarkStudios/rust-gpu (visited on 26/01/2022).

[19] Umme Sara, Morium Akter and Mohammad Shorif Uddin. 'Image Quality Assessment through FSIM, SSIM, MSE and PSNR—A Comparative Study'. In: *Journal of Computer and Communications* 7 (2019), pp. 8–18.

[20] *SCLT Source Code*. URL: https://github.com/chicio/Spectral-Clara-Lux-Tracer (visited on 28/01/2022).

[21] Paul A. Tipler and Gene Mosca. 'Physik'. In: ed. by Peter Kersten and Jenny Wagner. Springer Spektrum, 2019, p. 1044. ISBN: 978-3-662-58281-7. DOI: 10.1007/978-3-662-58281-7.

[22] Ingo Wald and Vlastimil Havran. 'On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)'. In: (2006), pp. 61–69. DOI: 10.1109/RT.2006.280216.

[23] A. Wilkie et al. 'Hero Wavelength Spectral Sampling'. In: 33.4 (2014). Ed. by Wojciech Jarosz and Pieter Peers.

[24] Chris Wyman, Peter-Pike Sloan and Peter Shirley. 'Simple Analytic Approximations to the CIE XYZ Color Matching Functions'. In: *Journal of Computer Graphics Techniques (JCGT)* 2.2 (July 2013), pp. 1–11. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/02/01/.