Universidad Zaragoza

1542

Master's Thesis

# Técnicas para renderizado espectral en tiempo real
# Techniques for real time spectral rendering

Author

Pedro José Pérez García

Supervisors

Néstor Monzón González

Adolfo Muñoz Orbañanos

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2024

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. Pedro José Pérez García ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de Estudios de la titulación de Máster Universitario en Robótica, Gráficos y Visión por Computador (Título del Trabajo)

Techniques for real time spectral rendering
Técnicas para renderizado espectral en tiempo real

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 26 de junio de 2024

Firmado por PEREZ GARCIA PEDRO JOSE - ***8312** el día 26/06/2024 con un certificado emitido por AC FNMT Usuarios

Fdo: Pedro José Pérez García

# Contents

# Abstract

Traditional RGB rendering presents problems during image synthesization upon encountering wavelength-dependent phenomena, such as participating media scattering or iridiscence. Spectral rendering takes into account the entire visual spectrum, and thus, is able to correctly evaluate these types of phenomena. However, RGB rendering is still the dominant choice due to how expensive it is to produce spectral materials and assets based on real-world data, and how heavy in memory those can be both in memory and disk storage, making the entire process of moving and storing that data a cumbersome task that introduces overhead and causes bottlenecks in real-time rendering engines, where computation time is critical.

We present a physically-based, real-time spectral rendering pipeline that aims to mitigate these two factors by enabling the use of RGB assets in spectral contexts, introducing as little error as possible. For that end, we propose a real-time adaptation of a spectral upsampling technique, which allows for obtaining spectral responses for reflectances from the RGB coefficients in the original reflectance texture.

To exemplify this proposed pipeline, we offer an implementation that aims to be performant, extensible and follows the main principles of physically based rendering. For this last goal, we allow for simulating different observer response curves and we implement physically based materials. We implemented our pipeline in OpenGL form scratch, excluding some libraries that alleviated the work on the most basic tasks.

To prove the extendability of our approach, we combine it with previously existing techniques for real-time spectral rendering. We chose a technique to render underwater oceanic scenes in real time, allowing us to fully render those scenes in a spectral fashion.

Lastly, we validate our results against a path traced simulation, showing great accuracy for wavelength-dependent scenarios and outperforming RGB rendering. Our method proves to be the most reliable across all the test scenarios, offering great frame rates for a little price in terms of processing power.

# Acknowledgements

I would like to thank so many people for shaping who I am throughout my life, up until this point. Every little lesson, every moment, every word, every sad moment and every laugh have shaped me in a way nothing else could have.

Your names might not fit these pages, but my thankfulness towards all of you will always be within me, accompanying me wherever I go.

To mention some people, in a more or less structured order:

- Thank you, mom and dad, for sacrificing yourselves in body, mind and soul, to an extent I didn't even believe to be possible, without ever asking for anything in return. Now it's my turn.

- Thank you, Gonzalo and Diego, my inseparable brothers, for being such stupid idiots and making the bad days less bad. Thanks for putting up with my bad mood during this final few weeks. Not everybody is lucky enough to be a triplet but I'd recommend it to everyone. Stay silly.

- Thanks to Jorge and Héctor, my two best friends, for the never ending evenings where we talked about life, our fears, hopes and the uncertainty the future holds. I'm sure both of you will be a great novelist and artist, respectively.

- Of course, I can't forget my thesis supervisors, Néstor and Adolfo. Thanks for your patience and especially for your help and guidance during the last months. (Thanks Néstor for fixing my plots the day before the deadline)

- Lastly, thanks to everyone in the Graphics and Imaging Lab; everyone there is part of an incredible environment full of opportunities and talent, having such a group of amazing people and friends almost seems unreal, but it's not, thankfully.

# 1. Introduction

Physically based rendering (PBR) is one of the most common approaches in Computer Graphics. Its main goal is to synthesize images that are indistinguishable from those in the real world, and for this purpose, the physics of light transport and its interaction with matter need to be simulated.

Nowadays, Physically based rendering is highly demanded in several industries, like cinema production for rendering animated movies, or for adding visual effects in action movies that look convincing to the eye of spectators. Other possible uses can be found in interior design, cultural heritage activities, or videogames, where developers try to achieve a realistic look in order to increase the feeling of immersion that the players might experiment. From an artistic point of view, supporting physical simulations of light transport within a rendering engine allows artists to shift their focus into other (historically less tedious) tasks. Such a demand from the greatest entertainment industries in the world makes physically based rendering one of the most, if not the most popular field within Computer Graphics.

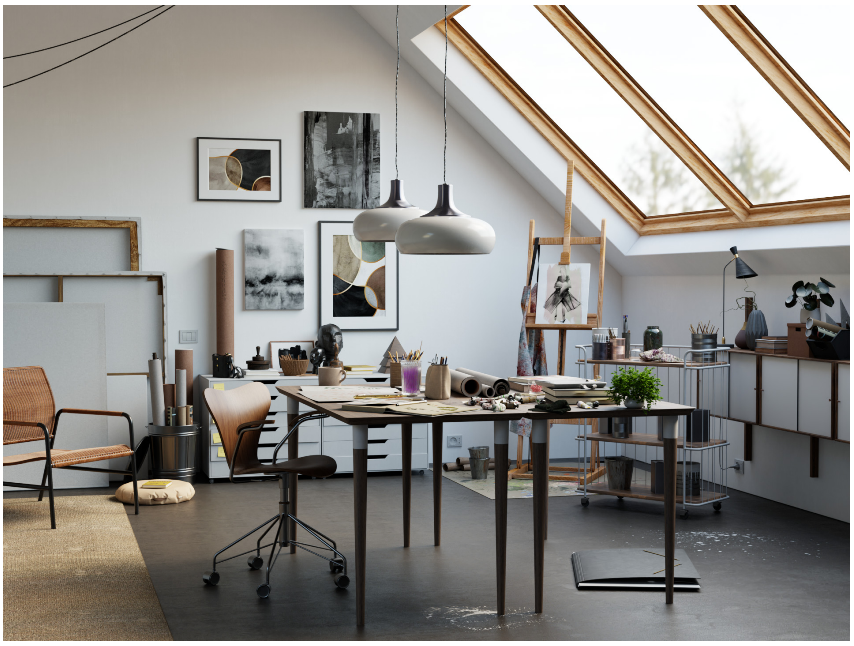


Figure 1.1: An example of what can be achieved with Physically Based Rendering. Credits to Matt Pharr, Wenzel Jakob, and Greg Humphreys. Reproduced under the Apache License.

However, despite the great advancements in PBR over the last 30 years it is still standard practice to synthesize images using only the red, green and blue colours as the image primaries. Using these three colours as *tristimulus* is widely known as RGB rendering. Its popularity is mainly due to the fact that the RGB representation model is designed to mimic our visual perception, since our visual system is composed of three types of cone cells that are specialized in receiving light at three different ranges of wavelengths, corresponding to short, medium and long wavelengths (SML). As figure 1.2 illustrates, the spectral radiance that each type of cell is specialized in sensing matches the ranges for the red, green and blue colours.

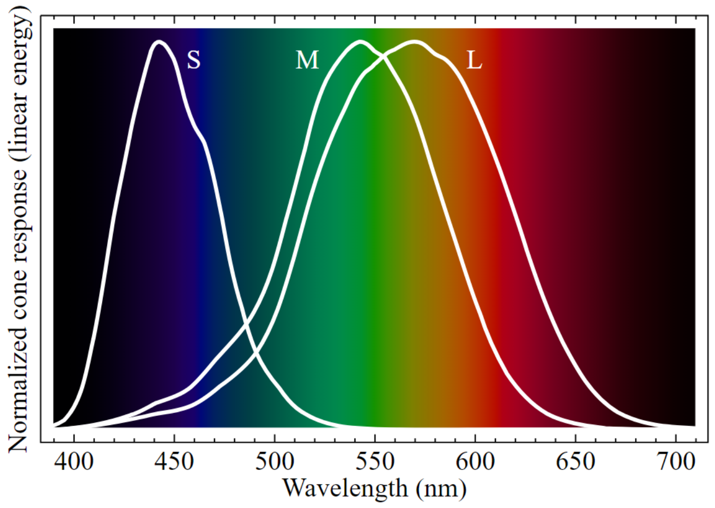


Figure 1.2: Graph illustrating the spectral responses of each type of human cone cell. Each one specializes in handling short (S), medium (M) and long (L) wavelengths.

On top of that, RGB rendering is generally faster and provides sufficiently good results when compared to its alternatives. In a greater context, the RGB colour model is a standard for encoding and displaying images in photography, video, television and digital displays. This is, again, because it allows to closely reproduce a wide variety of colors as just combinations of these three primaries, which were designed to mimic the human photoreceptors [SG31].

Therefore, this RGB colour model is a great model of the *interaction of light and our visual system*. However, it is not a great general model for modeling the *interactions of light and matter*, since light, as we know it, is the part of the continuous electromagnetic spectrum we can see with our eyes. Thus, the RGB colour model is a simplification of reality we're making for representation purposes. This discretization results in inaccurate final colours in synthesized images, introducing error in the final colours, as we show in Figure 1.4.

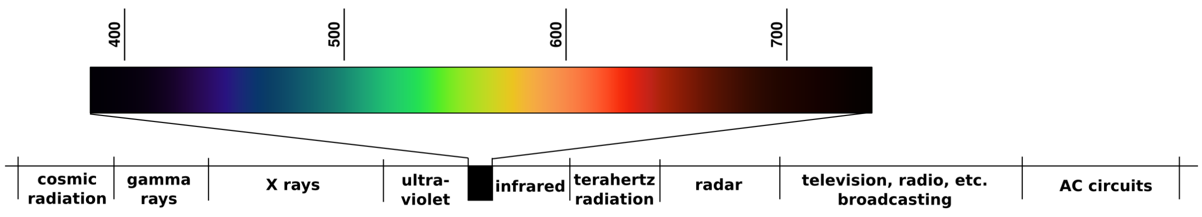


Figure 1.3: Visible spectrum to the human eye, wavelengths are expressed in nanometers. Modified from Johannes Ahlmann and licensed under the Creative Commons license.

Spectral rendering removes these limitations by simulating light transport in the spectral domain, making the lighting computations as a function of wavelength instead of the traditional RGB triplets.

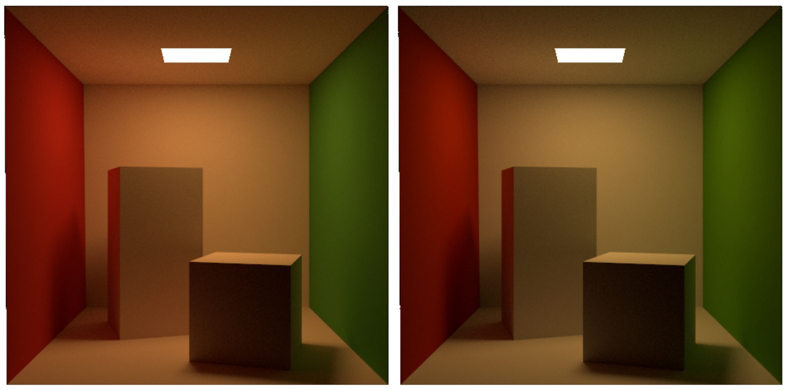


Figure 1.4: Comparison between two images synthesized using RGB rendering (left) and spectral rendering (right). There are visible colour differences in the left side of the image, where red colour bleeding is attenuated in the spectral image with respect to the RGB one. Image obtained from the Mitsuba documentation [JSR+22a].

As it can be seen in the images in Figure 1.4, using RGB rendering (left) presents some quite noticeable differences in the final result even in this simple scene, while spectral rendering (right) allows a more faithful simulation of light transport. Additionally, there are some wavelength-dependent phenomena that RGB rendering is not able to reproduce. For example, thin film interference occurs when light is reflected in both the inner and the outer layers of an extremely thin film, in the order of nanometers. These two reflected waves interfere with each other, both destructively and constructively, drastically changing the spectral distribution of the incident light and resulting in complex colour patterns like the example in Figure 1.5. As the interference patterns depend on the wavelength of light, spectral rendering is crucial to reproduce them.

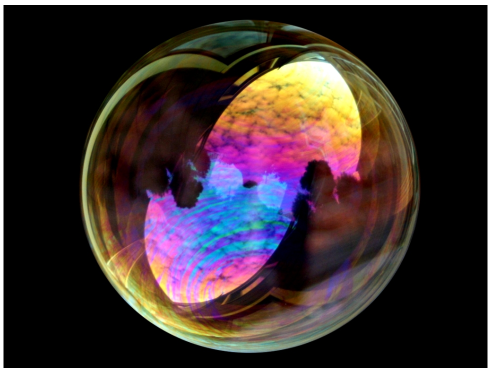


Figure 1.5: An example of thin film interference, a bubble of soap with water, where the thickness of the film is in the order of nanometers, creating several destructive and constructive wave interferences that depend on the wavelength of the light. RGB rendering cannot simulate this type of wavelength-dependent phenomena.

In industry environments, authoring 3D models requires creating materials and textures, and while spectral rendering is ideal for achieving realism, correctly gathering, interpreting and using spectral data for materials is extremely hard, while in an RGB pipeline it is as easy as just painting the colours for the materials. By combining these two factors, it can clearly be seen why RGB is still the dominant choice, despite being an inferior choice when it comes to accuracy in the final colours.

There's, however, an additional factor to account for. Most industry-level productions require a great amount of assets, up to hundreds of thousands. This is an extremely important consideration that makes disk space storage a critical factor to take into account. RGB materials and assets generally need only 3 colour channels to work properly, while a spectral renderer that is taking into consideration $n$ different wavelengths will need $n$ channels on its textures and assets.

This translates into extra disk storage, application memory consumption, and overheads upon fetching, loading and writing values on those assets, rendering this approach extremely prohibitive in practice.

This is what motivates our work, which bypasses these prohibitive memory requirements by combining the efficiency of *lightweight RGB textures*, and the accuracy of *spectral lighting computations*.

## 1.1 Real-time spectral rendering

This trend is much more noticeable in real-time applications, where computing time is critical, almost instantly ruling out the use of spectral rendering. For reference, real-time applications like videogames need to keep a stable framerate of at least 30 frames per second (FPS) to be perceived as smooth, which means that one image must be generated every 33.3 milliseconds. Lately, and depending on the type of application, the standard has been raised, with 60 FPS (one frame every 16.6 milliseconds) being the minimum threshold for some users.

The main goal of our work in this Master's Thesis is to present ways to enable the usage of spectral rendering in real-time applications, without compromising their performance or increasing the complexity associated to the creation of these assets. For that end we leverage spectral upsampling techniques, which allow using RGB assets in spectral rendering, introducing minimal round-trip error. We will cover the chosen techniques and lay out the reasons behind our choice.

To validate our approach, we have implemented a real-time spectral rendering engine where we adapt those techniques for real-time rendering. We also aim to validate our approach's compatibility with other existing methods for spectral rendering, ensuring that it can be extended and can work with other techniques, like oceanic rendering.

Finally, we provide some measurements to evaluate the results of our work, which can be found in Section 5.

## 1.2 Terminology: uplifting and upsampling

As a final note, it is worth mentioning that through all this thesis, the terms *spectral uplifting* and *spectral upsampling* are equally used in order to refer to the process of generating spectral

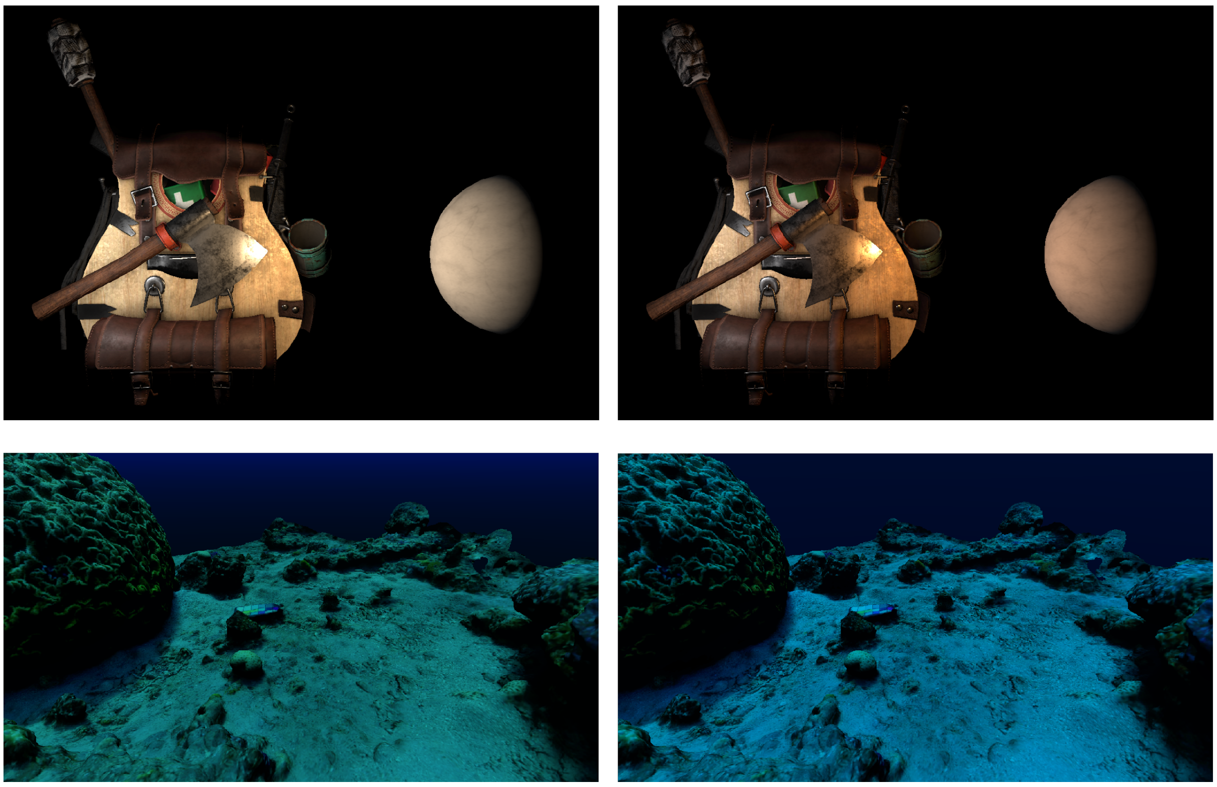


Figure 1.6: A preview of some of the results we have achieved. The images in the left column have been synthesized using traditional RGB rendering, while the ones in the right have been obtained via spectral rendering. We achieve better, more faithful colours with spectral rendering in real-time compatible framerates.

coefficients from RGB values, since both are used in literature (For example, Jakob and Hanika use *upsampling* [JH19], while Tódová prefers to use *uplifting* [TWF21]).

# 2. Theoretical background

So far we have offered a quick overview to the main problem we are trying to tackle: Spectral rendering, although being able to yield more accurate colours, is slower and more cumbersome than traditional tristimulus RGB rendering, due to its associated computational cost and the difficulty that creating, using and storing spectral assets implies. Those problems are accentuated in a real-time context, which is where our work comes in. We aim to present several techniques that can alleviate the downsides of real-time spectral rendering, in a way that allows combining them with other spectral rendering techniques.

In this section, we introduce some of the concepts and techniques that are relevant to our work, including those ones in which we base our approach.

## 2.1 Light Transport

Light transport is a well-studied problem that has drawn attention across several scientific disciplines for decades, leading to different approaches and methods to solve it.

The first formal formulation for it came with Chandrasekhar, on his treatise Radiative Transfer, where he first introduced the homonimous equation [Cha50] that tackles the behaviour of light when traversing participating media like fog, clouds, or water, in an integro-differential manner. Most derived works try to solve or approximate this equation.

### 2.1.1 The rendering equation

James Kajiya [Kaj86] was responsible for reformulating the Radiative Transfer Equation equation to a form more akin to the field of Computer Graphics (See Equation 2.1). This is a simplification of the previous equation which omits participating media and focuses on surfaces. It is still widely used, as the effect of media can be ignored in many scenes, especially in a small scale (recall, for example, the Cornell Box of Figure 1.4), and results in an easier integral equation to solve.

$$L_o(\mathbf{x}, \omega_\mathbf{o}, \lambda) = L_e(\mathbf{x}, \omega_o, \lambda) + \int_\Omega L_i(\mathbf{x}, \omega_\mathbf{i}, \lambda) f_r(\mathbf{x}, \omega_\mathbf{i}, \omega_\mathbf{o}, \lambda)(\omega_\mathbf{i} \cdot \mathbf{n}) d\omega_\mathbf{i} \tag{2.1}$$

This rendering equation describes in a formal manner how to compute the spectral radiance

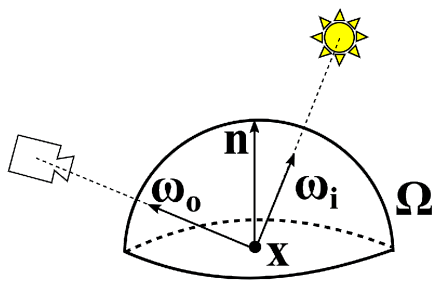


Figure 2.1: A visual simplification of the elements taken into account in the rendering equation: A camera or sensor, at least one light emitter, and a surface point $x$ that determines the incoming light direction $\omega_i$, the outgoing light direction $\omega_o$ and the surface normal $n$. $\Omega$ represents all the possible directions from which light might be arriving towards $x$.

that is leaving a point in space $\mathbf{x}$ in a given direction $\omega_\mathbf{o}$. With it, we can compute the incoming spectral radiance for every position in the image plane, which is equivalent to saying that we can compute the final colour for every pixel in the images we are synthesizing.

The following terms are considered:

- $L_o(\mathbf{x}, \omega_\mathbf{o}.\lambda)$: The spectral radiance at a given wavelength $\lambda$ outgoing from a point $\mathbf{x}$ in space towards a direction $\omega_\mathbf{o}$.
- $L_e(\mathbf{x}, \omega_\mathbf{o}, \lambda)$: The spectral emission of a given surface at point $\mathbf{x}$, into the direction $\omega_\mathbf{o}$. If the surface at point $\mathbf{x}$ is not a light emitter, this term will be nil.
- $\int_\Omega ...d\omega_\mathbf{i}$: This integral means that we have to take into account all the incoming directions $\omega_\mathbf{i}$ in the hemisphere $\Omega$ around the surface point $\mathbf{x}$, given that their outgoing direction is $\omega_\mathbf{o}$. At the same time, it means that this equation has a recursive nature where we have to compute the outgoing radiance at several points to get the incoming radiance at a different point.
- $L_i(\mathbf{x}, \omega_\mathbf{i}, \lambda)$: The incident spectral radiance that comes into $x$ from direction $\omega_\mathbf{i}$, with a particular wavelength $\lambda$.
- $f_r(\mathbf{x}, \omega_\mathbf{i}, \omega_\mathbf{o}, \lambda)$: This term corresponds to the bidirectional reflectance distribution function (BRDF) of the surface's material. It describes how light with a given wavelength $\lambda$ interacts with the material in the surface at point $\mathbf{x}$, coming from a direction $\omega_\mathbf{i}$. It determines the outgoing direction $\omega_\mathbf{o}$, and the amount of light that gets reflected, which is usually less than the amount of incident light due to surface absorption.
- $(\omega_\mathbf{i} \cdot \mathbf{n})$: Attenuation to take into account the angle between the surface's normal $n$ and the incoming direction of the light $\omega_\mathbf{i}$. If the light comes at a direction directly perpendicular to the surface, it covers a smaller area (a single point in space) and therefore its energy is more concentrated, while if it comes at a narrow angle, it covers a larger area, spreading its energy evenly and decreasing the amount that is focused at a differential point $\mathbf{x}$.

In Equation 2.1 we show a dependence on wavelength for most terms, but it can be omitted when doing RGB rendering by discretizing into those 3 colours.

### 2.1.2 Volumetric Rendering Equation

The aforementioned equation falls short to incorporate the contributions made by participating media like fog, water, smoke or clouds. We can take them into account by reintroducing some coefficients from Chandrasekhar's RTE:

- **Absorption coefficient** $\sigma_a$: Expressed in $m^{-1}$, it defines the probability of absorption per distance unit.
- **Scattering coefficient** $\sigma_s$: Expressed in $m^{-1}$, it defines the probability of an scattering event per distance unit. Scattering can be towards our light beam, gaining radiance (in-scattering), or out of it, losing radiance (out-scattering), as shown in Figure 2.2.
- **Emission coefficient** $\sigma_e$: Expressed in $m^{-1}$, it defines the emitted radiance per distance unit. Most participating media are non-emissive, and this term is often ignored.

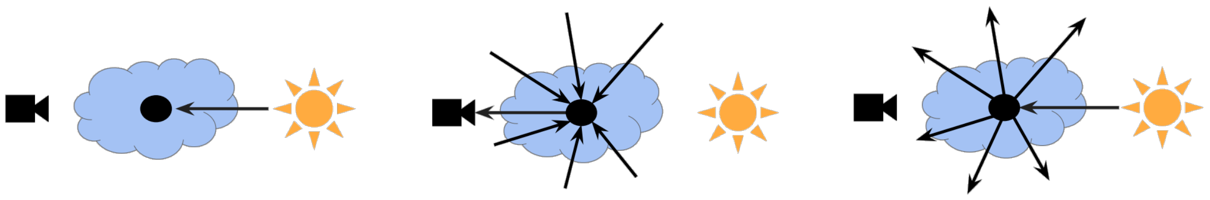

(a) Absorption (b) In-scattering (c) Out-scattering

Figure 2.2: Representation of the possible events in a non-emissive participating medium. As previously mentioned, in-scattering adds radiance, out-scattering substracts it and so does absorption.

It is common to sum together the absorption and scattering coefficients to form a new term called the extinction coefficient, $\sigma_t$. It is used to refer to the probability of radiance extinction, independently of its cause (out-scattering or absorption).

Additionally, the ratio between the scattering coefficient and the extinction coefficient is known as single scattering albedo, $\alpha$. We can think of it as the colour of our medium. It is also common to find the volume coefficients defined as space dependent. For example, instead of $\sigma_s$, $\sigma_s(x)$. Participating media where coefficients depend on the position in space $x$ are referred to as heterogeneous, and homogeneous media are those where scattering and absorption do not vary spatially. In fact, those coefficients are also wavelength-dependent, where spectral radiance gets attenuated differently depending on the wavelength $\lambda$.

The RTE expresses the radiance difference at a given point $\mathbf{x}$, through a direction $\omega$, for a small step through the ray of an infinitely small distance $dz$ as:

$$\frac{dL(\mathbf{x_z}, \omega)}{dz} = \sigma_s L_i(\mathbf{x_z}, \omega) - \sigma_t L(\mathbf{x_z}, \omega) \tag{2.2}$$

Note that equation 2.2 is ignoring emission since it is not a common phenomenon among participating media (besides fire and explosions).

If we put the RTE in integral form, we get the non-emissive Volume Rendering Equation:

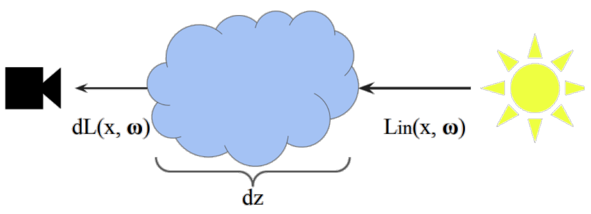


Figure 2.3: Radiance difference for a light ray stepping an infinitely small distance $dz$ through media.

$$L(\mathbf{x},\omega,\lambda) = T(\mathbf{x} \leftrightarrow \mathbf{x_z})L(\mathbf{x_z},\omega,\lambda)+\int_x^{\mathbf{x_z}} T(\mathbf{x} \leftrightarrow \mathbf{x_t})\sigma_s(\mathbf{x_t},\lambda)\int_\Omega L(\mathbf{x_t},\omega_\mathbf{i},\lambda)f_s(\mathbf{x_t},\omega_\mathbf{i},\omega,\lambda)d\omega_\mathbf{i}d\mathbf{x_t} \tag{2.3}$$

In Equation 2.3 we can see some new terms with respect to the standard rendering equation:

- $T(\mathbf{x} \leftrightarrow \mathbf{x_z})$: Transmittance between two points $\mathbf{x}$ and $\mathbf{x_z}$. This is the amount of radiance that reaches depth $z$ traveling through the ray. The rest gets extincted. It is given by the Beer-Lambert law, as in equation 2.4.
- $\int_x^{\mathbf{x_z}}$: Means that we have to account and integrate every contribution along the ray from $\mathbf{x}$ to $\mathbf{x_z}$, in infinitesimal intervals.
- $\int_\Omega$: For every infinitesimal point along the ray direction, we need to account for the inscattered radiance arriving from every possible direction $\Omega$ in space.
- $f_s(\mathbf{x},\omega_\mathbf{i},\omega,\lambda)$: The phase function. It models the distribution of possible scattering directions. If every direction has the same probability, the phase function is isotropic. If not, we call it anisotropic. It's also wavelength-dependent. It can be seen as a BRDF for participating media.

$$T(\mathbf{x} \leftrightarrow \mathbf{x_z}) = e^{-\int_{\mathbf{x_0}}^{\mathbf{x_z}} \sigma_t(\mathbf{x},\lambda)dz} \tag{2.4}$$

Computing analytical solutions to the rendering equation (2.1) and the volume rendering equation (2.3) is impossible in practice, which caused a shift in the approach that was taken towards them.

Over time, several approaches to solve these equations have been developed, with those based in Monte Carlo integration being the most popular. Path Tracing is the biggest exponent of this kind of algorithms [AK90], followed by ray tracing [Whi80] and photon mapping [JMLH01]. Monte Carlo methods are unbiased, which means that, if given an unlimited amount of computation time, they will converge into the correct solution to the light transport problem, presenting no noise on the final image. Lots of derived works have tried to speed up the convergence of this method, by either improving the sampling techniques in order reduce variance [PM93, MHD16]. Others, on the other hand, have focused on performing a better search of the illumination sources in the scene, like Bidirectional Path Tracing [LW96, JA18], Metropolis Light Transport [VG97, PKK00] or Path Guiding [HZE$^+$19, DWWH20].

## 2.2 Spectral Rendering

### 2.2.1 Theoretical basis

In general, all rendering systems approximate the rendering equation (2.1) (or the volume rendering equation (2.3) for participating media) for every pixel in the screen. However, while RGB renderers evaluate every term for RGB triplets and finally return the color $L_{RGB}$ which can be directly displayed, spectral renderers evaluate the full spectral power distribution $L(\lambda)$ at a given resolution. This then needs to be finally converted into RGB to match our perception, a process that would be carried out by our visual system and its cone cells in the real world.

This conversion should match our perception of colours. The International Commision on Illumination (CIE, *Commission internationale de l'éclairage*) defines the colour-matching functions [SG31], achieved by experimental testing. These functions define the proportion of intensity in each of the primaries (red, green and blue) that match a light with a given wavelength, also known as the *standard observer*. They are also one of the main pillars of Colorimetry, a sub-field that mixes different disciplines like Physics, Computer Graphics or Psychology, among others. It is heavily based on human perception and the human visual system.

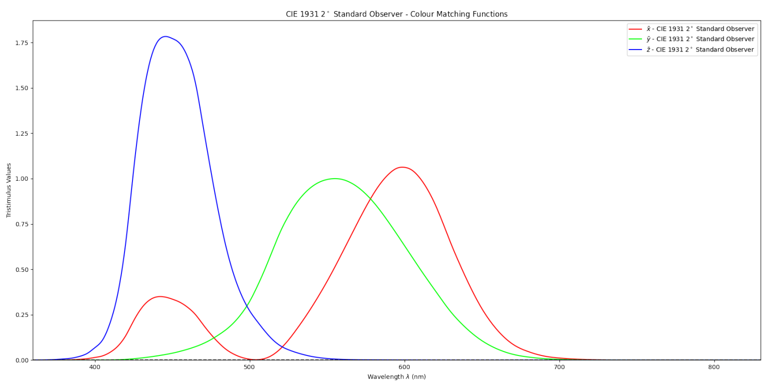


Figure 2.4: The CIE 1931 colour matching functions, in XYZ colour space.

If we compute the inner product of our spectral luminance $L(\lambda)$ and the colour matching functions for each channel ($\bar{r}$, $\bar{g}$ and $\bar{b}$), we can get the corresponding tristimulus response for the final image:

$$\begin{aligned} L_R &= \int_\Lambda L(\lambda)\bar{r}(\lambda)d\lambda \\ L_G &= \int_\Lambda L(\lambda)\bar{g}(\lambda)d\lambda \\ L_B &= \int_\Lambda L(\lambda)\bar{b}(\lambda)d\lambda \end{aligned} \tag{2.5}$$

It is common to integrate over the range of the visible spectrum $\Lambda$, which goes from around 400 nanometers to around 700 nanometers, but varies depending on the implementation.

In order to compute the integral described in (2.5), it is needed to perform a discertization of the spectral domain. Since the point of performing spectral rendering is to achieve better colour precision by avoiding the discretization that RGB rendering does, the more samples the better final result. We approximate the integral by using the rectangle rule, transforming it into a sum of $n$ equally weighted samples (although it could be possible to create unevenly sized intervals with different weights each):

$$\begin{aligned} L_R &= \sum_{\lambda} L(\lambda)\bar{r}(\lambda)\Delta\lambda \\ L_G &= \sum_{\lambda} L(\lambda)\bar{g}(\lambda)\Delta\lambda \\ L_B &= \sum_{\lambda} L(\lambda)\bar{b}(\lambda)\Delta\lambda \\ \Delta\lambda &= \frac{\lambda_{max} - \lambda_{min}}{n} \end{aligned} \tag{2.6}$$

Additionally, it is usually preferred to use the XYZ curves (with $\bar{x}$, $\bar{y}$ and $\bar{z}$ components, as in Figure 2.4), instead of the RGB ones because of the negative coefficients those have, and then convert from the XYZ color space to RGB by using the transformation in equation (2.7).

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \times \begin{bmatrix} 3.2406255 & -1.5372080 & -0.4986286 \\ -0.9689307 & 1.8757561 & 0.0415175 \\ 0.0557101 & -0.2040211 & 1.0569959 \end{bmatrix} \tag{2.7}$$

### 2.2.2 State of the art

As shown by Borges [Bor91], RGB rendering introduces error with respect to spectral rendering, which needs to take into account as many wavelengths of the spectral domain as possible, making it expensive. This is the main reason behind the popularity of RGB rendering in production and industry contexts, with the other being the ease of creating assets with RGB colours and parameters with respect to the difficulty of gathering spectral data and creating materials with it, in a way that users and artists can control and understand.

To alleviate these problems, several techniques have been proposed to enhance the performance of spectral rendering. Hero Wavelength Spectral Sampling [WND$^+$14] simplifies the problems of having to handle several wavelenghts at once when rendering using Path Tracing algorithms, especially the issues related to directional sampling. This technique allows for vectorization and can be easily combined with others, like van de Ruit and Eisemann's [vdRE21], which generates a coarse spectral radiance estimation in screen space, to perform importance wavelength sampling in the final rendering pass. Unfortunately for us, these techniques are unsuitable for real-time rendering.

## 2.3 Real-Time Spectral Rendering

In the last 40 years, advances in both hardware and software have made it possible to render images in real time, at an interactive frame rate. This means that a new image or frame is generated every few milliseconds. On the hardware side, graphics processing units (GPUs) have become more powerful, while in the software side, the expensive ray tracing was substituted by a pipeline based in rasterization [Pin88]. For a long time, most of the light transport calculations had to be heavily simplified, or completely ignored in pursuit of stylized graphics that did not satisfy the principles of light transport nor PBR [GGSC99, Dec96], a discipline known as stylized rendering or non-photorealistic rendering.

As time passed and Moore's Law allowed for it [Moo65], graphics processing hardware was able to put up with the demands of rendering algorithms, at high enough refresh rates to be considered in real time. Eventually, real-time spectral rendering was considered and several techniques were developed, in order to provide approximations to phenomena that can only be correctly simulated via spectral rendering, instead of the traditional RGB one.

For instance, Belcour and Barla propose a real-time approximation for iridiscence via a modified microfacet model [BB17], and another real-time (they don't test it for real-time rendering, even though they mention the method is fast enough to be used in a real-time setting) approximation for the Usambara effect [BBG23], which is a change in material colour depending on the light path length that is observed in some minerals and gemstones, like jade. Other wavelength-dependent phenomena that have been approximated for real-time rendering include diffraction and thin-film interference, consequence of several diffractions and refractions in thin layers over a material [TG17a, TG17b]. Lastly, Monzón et al. [MGAM24] propose a method to render oceanic underwater scenes by computing an analytic approximation of underwater scattering, based on real world data.

## 2.4 Spectral reflectance upsampling

To tackle the issues related to the creation of spectral assets, there are several works that aim to enable using RGB models and material textures in a spectral setting. MacAdam [Mac35] was able to produce spectra with box shapes by combining a small amount of predetermined spectra. The results were not very good, but his work set the first step in spectral reconstruction using other basis. From there, Smits [Smi99] proposed formulating the spectrum generation as an optimization problem on his pioneering work which was seminal to several others. It presented some issues, like not being compliant with energy conservation, or only being able to reproduce a small gamut of reflectances.

It is also worth mentioning that the problem presented by spectral upsampling is a highly ill-posed one, since a single RGB tristimulus value could correspond to an infinite subspace of reflectance spectra. This is what we know as metamerisms, the infinite possible combinations of reflectances that can produce the same final RGB colour. In order to tackle this problem, additional constraints have to be imposed for the optimization steps.

Posterior works improve on those issues, like the one by Meng et al. [MSHD15] where they improve the gamut of spectra that can be correctly reproduced by mapping reflectances to a bidimensional grid in the *xy* chromaticity plane, within the XYZ colour space. Their work still presented some limitations, since it introduced some round trip error in spectra reconstruction, and some reflectances did not generate correct results. Otsu et al. [OYH18] also reconstruct spectra with using basis functions, but incorporate Principal Component Analysis and greedy clustering to their method, in order to reduce dimensionality and round trip error. Additionally, their method only requires one matrix multiplication for the final reconstruction. Drew and Finlayson [DF03] propose a method "to multiply spectra without carrying out spectral multiplication" based on spectral sharpening, which translated the RGB values into sharp basis that allows modeling illuminant change with a diagonal matrix, instead of the common 3x3 one. Other works have achieved complete reconstruction of the sRGB reflectance gamut [MY19] (although Jakob and Hanika were first by a small margin of time, as we are about to mention).

All of the works mentioned so far focus on spectral reconstruction using basis functions, but concurrently, other types of approaches were developed. The work of Jakob and Hanika [JH19] was the first to completely map the entire sRGB gamut. They achieve so creating a function space parametrized by coefficients, which they generate by optimization. Those coefficients are stored in 3D look up tables that take into account the brightness of colours, unlike other previous works we mentioned. Those coefficients can be used to recover spectral response with an efficient sigmoidal mapping, which ensures that generated spectra will be smooth, a desirable quality.

Other works have stemmed from this one, trying to focus on metametic behaviour (where several spectra and illuminant combinations can produce the same colour appearance) [vdRE23] or on reducing the round trip error for user-specified reflectances [TWF21, TWF22].

Our work is also a continuation of the one by Jakob and Hanika. We take their resulting look up tables and use them in our real-time spectral rendering pipeline, aiming to make possible working with RGB assets. This way we also validate the usability of their work for a real-time environment, something the authors didn't contemplate in their original work. We also combine their approach with other techniques, to prove that out approach can be extended with preexisting techniques like those mentioned in 2.3, which will benefit from our solution.

Since it is extremely relevant for our work, we discuss more about Jakob and Hanika's work and why we choose it as our starting point in Section 3.2.1.

## 2.5 Spectral upsampling beyond reflectances

Even though most of the work we have presented so far is focused on generating spectral representations of the RGB responses for the reflectances in materials, there are other works that aim to upsample other types of RGB coefficients into the spectral domain. Guarnera et al. [GGDG22] propose a method to do so with RGB image-based illumination via both a genetic algorithm approach and a neural approach, while Jendersie [Jen21] proposes a simple yet effective method to upsample volume attenuation coefficients, by optimizing two threshold wavelengths that will determine which coefficient from the RGB tuple will be returned as the spectral response value.

We believe that this could be a good point to continue our work past the extension of this thesis. If we managed to get a general approach for correctly upsampling illuminations or volume coefficients, we could perform complete spectral rendering over RGB assets without introducing error, or introducing it minimally. We would also need to set the goal that those approaches should be performant in order to be used in real-time settings.

# 3. Our approach to real-time spectral rendering

Up until now we have introduced the main ideas behind spectral rendering, as well as some of the main shortcomings that make it less popular than its RGB counterpart.

To summarize, RGB rendering makes a discretization of the electromagnetic visible light spectrum into 3 bands that represent the red, green and blue components of our final images. When simulating certain phenomena that depend on wavelength (like the ones we can find in rainbows, clouds, soap bubbles and many others), relying in RGB rendering introduces error and produces incorrect results. On the other hand, spectral data has two main problems. First producing spectral assets for rendering is expensive and cumbersome in comparison with using RGB textures that artists can manipulate much more easily. Secondly, in the engine itself, spectral textures are problematic due to the memory bandwidth they require. Spectral upsampling is a technique that allows using RGB assets by converting the reflectance values into spectral responses for our rendering process, alleviating the main issues presented by spectral rendering.

This is the starting point of our work, where we have devised a real-time spectral rendering pipeline that allows for using RGB assets. As part of our contribution, we have developed a strategy to use a spectral upsampling technique that was concieved for offline rendering in real-time rendering, taking Jakob and Hanika's work as our starting point. We have created our own implementation of it in order to validate it, and we have combined it with other spectral rendering techniques (explained in Section 3.4) to prove that it can be extended and that we can build other techniques on top of it, further improving the results with respect to using those techniques separatedly.

First, we are going to explain the most relevant theoretical concepts behind each step of our implementation, and later (see Section 4) we will offer more low-level details on how everything works under the hood, like how the data is loaded and passed, which textures we need to create and use, as well as the tools we chose and the reasoning behind several design decisions.

## 3.1 Wavelength sampling

Spectral rendering implies integrating the spectral radiance contribution for each wavelength over the visible light spectrum, but in practice sampling infinite wavelengths is impossible. Instead, it is common practice to discretize the spectrum into several wavelengths that will be summed and weighted together to compute the final response. Logically, the more wavelengths we use in the process, the more accurate our results will be.

The spectral sampling process is an open area of research [vdRE21]. It would be ideal to select the wavelengths that will give the best possible final result, or minimize the required number of samples, without introducing error in the synthesized images. There are several techniques and strategies that aim to achieve this goal, as we have mentioned in Section 2.2. First, we need to determine the number $n_{wls}$ of wavelengths to use, and then we need to devise a strategy to carry out the actual wavelength selection (also referred to as sampling).

For our pipeline, we had in mind that we wanted the parameters to be modifiable, so we decided to allow for the number of wavelengths to be changed dynamically, to allow for better visualization of the effects of the number of samples over the final result.

For the actual wavelength sampling, we chose to go for the most obvious option and perform uniform sampling through the spectrum, which means that we select our wavelengths in a way that they are equidistant between them. After evaluating the spectral functions with those wavelengths, we integrate the obtained Spectral Power Distribution with a middle Riemann sum.

We also allowed users to change the wavelength interval at will by specifying minimum and maximum wavelengths, $\lambda_{min}$ and $\lambda_{max}$.

In equations 3.1 and 3.2 we show how to select the $i$-th wavelength sample from a total of $n_{wls}$ wavelengths by performing uniform sampling between $\lambda_{min}$ and $\lambda_{max}$:

$$\Delta = \frac{\lambda_{max} - \lambda_{min}}{n_{wls}} \tag{3.1}$$

$$\lambda_i = \lambda_{min} + \Delta i + \frac{\Delta}{2} \tag{3.2}$$

We will discuss more about the impact of the chosen number of wavelengths on performance in Section 5, Results. We also make some comments on extending our wavelength sampling strategies in Section 6.

## 3.2 Spectral upsampling

We aimed to create a real-time spectral rendering approach that mitigates some of the most common problems that spectral rendering presents. One of those problems is the difficulty presented by the creation of assets with materials ready to be used in spectral contexts. Usually, RGB materials and textures are much easier to use and create, given that an artist can just paint them, or use pictures as reference. This is not the case for spectral materials, where it is much harder to obtain tabulated measurements of the material's repsonse at different wavelengths, with the desired wavelength resolution (separation in nanometers between samples in the table), and so on.

Another option could be converting RGB textures from existing materials into spectral textures, where instead of getting three primary values for each pixel in the image, we would have one spectral response for each wavelength per pixel. Taking for reference a sampling resolution of 1 nanometer and a spectrum where our wavelengths $\lambda \in [300..700]$ nanometers, that's a x100 increase in storage costs for spectral textures. We would also have to take into account the fact that performing such a conversion would be a data-hungry process, and probably quite time demanding, especially if artists need to have some kind of control over the results.

We considered that this was a great entry point for our work from a real-time rendering point of view; and thus, we determined that spectral upsampling was the most appropiate technique to incorporate first into our pipeline.

Among the different techniques that we have presented (Section 2.4) and evaluated, we consider that Jakob and Hanika's work [JH19] can be a promising candidate to build upon. We dedicate the next section (Section 3.2.1) to introduce the theoretical basis behind their approach, to delve deeper into its particularities and to better understand the consequent challenges we faced to integrate the method in our work. We follow with our conclusions and some of the challenges that our implementation would have to solve in Section 3.2.2.

### 3.2.1 The optimized look up tables

As we explained, the spectral upsampling method we are going to use is based on Jakob and Hanika's work [JH19]. This method aims to generate smooth spectra for the entire sRGB gamut. Since spectral upsampling is an ill-posed problem where an infinite amount of reflectance combinations can produce the same final colour, some constraints were imposed for the optimization step, for a given RGB colour **b**:

- The composition of mappings from RGB to spectra and its inverse should be an identity:

$$rgb(spec(\mathbf{b})) = \mathbf{b} \tag{3.3}$$

- If not possible, the error should be minimal:

$$\| \; rgb(spec(\mathbf{b})) = \mathbf{b} \approx 0 \; \| \tag{3.4}$$

  Where $\| \cdot \|$ represents a perceptual error metric.

- Generated spectra should present a smooth appearance.

Additional considerations were made regarding the practicality of the approach, but we focus on the theoretical formulation of their approach for now.

With those constraints in mind, the optimization problem is posed as finding a smooth spectrum $\hat{f}(\lambda)$ that maps to an RGB colour $\mathbf{b} \in [0,1]^3$ in a colour space, which in our case is the sRGB one. Mathematically, it is described as follows in Equation 3.5:

$$\hat{f} = \arg\min_{f} \; \| \; \mathbf{b} - \mathbf{T} \int_{\Lambda} f(\lambda) W(\lambda) \mathbf{xyz}(\lambda) d\lambda \; \| \tag{3.5}$$

Where:

- **W** represents the SPD (spectral power distribution) of the white point for the desired colour space. Since we are using sRGB, this corresponds to the D65 illuminant.
- $\mathbf{xyz}(\lambda)$ denotes the CIE 1931 colour matching functions that we introduced in Section 2.2.1.
- **T** is the colour transformation matrix that allows mapping from the XYZ space to the RGB one, as introduced in Equation 2.7.
- The integration domain $\Lambda$ is defined to range from 360 nanometers up to 830, covering the entire visible spectrum.
- For the perceptual error metric $\|\cdot\|$, the CIE $\Delta$E 2000 perceptual error metric [SWD05] is used.
- $f(\lambda)$ is the spectral representation function used.

For representing the spectrum, a simple analytic model was used, requiring three coefficients $c_0$, $c_1$ and $c_2$ of a second-degree polynomial and a sigmoid function:

$$f(\lambda) = S(c_o\lambda^2 + c_1\lambda + c_2) \tag{3.6}$$

$$S(x) = \frac{1}{2} + \frac{x}{2\sqrt{1+x^2}} \tag{3.7}$$

This representation for spectra ensures that they will have a smooth appearance. Thus, the real target of the optimization problem is to find the $c_i$ coefficients for any given RGB colour in the sRGB gamut. Additionally, equation 3.7 has a direct translation into shader code, which can be obtained by using highly optimized instructions like `fma` (full multiplication and addition) or `inversesqrt`.

Due to problems with the error obtained in the optimization, an additional change is done: Discretizing the sRGB colour space into three, easier to optimize regions.

This disrcetization yields three 2D textures for every maximally bright colour in the sRGB colour space. For non-maximally bright colours, an initial coefficient scaling by $\frac{1}{\alpha}$ was proposed, but introduced too much error. Instead, an iterative optimization is proposed where first a colour $\mathbf{c}$ with a stable solution is optimized, and then brighter and darker colours $\alpha\mathbf{c}$ are solved in both directions, using the solution of the previous iteration as a starting guess.

This returns three 3D cubes of a $64^3$ resolution, instead of 2D textures. A last change was made to those 3D cubes. For the third dimension in the cube that encodes the brightness $\alpha$, it was observed that there was a lot of variability in the stored coefficients $c_i$ near the 0 and 1 values. Thus, a remapping was made, delinearizing that dimension in a way that gave more resolution to the regions near both 0 and 1. We show the extreme changes in the coefficients before the remapping and the smoother transition after it, in Figure 3.2.

### 3.2.2 Challenges of integrating this spectral upsampling

As we just explained, we take this work as the starting point to build upon in our search for a real-time performant spectral rendering pipeline, due to the following reasons:

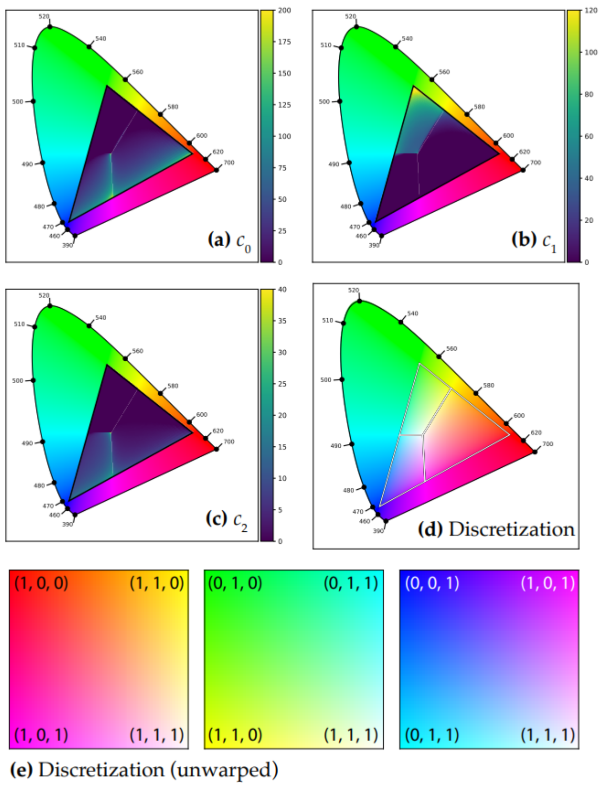


Figure 3.1: Obtained error after optimizing for all the sRGB gamut, shown for coefficients $c_0$ (a), $c_1$ (b), and $c_2$ (c). The proposed discretization is shown in (d). By unwarping the projection over the chromaticity plane, the final look of the originally proposed 2D textures is unveiled, along with the corresponding sRGB colour coordinates they map to (e). Image reproduced from the original work [JH19].

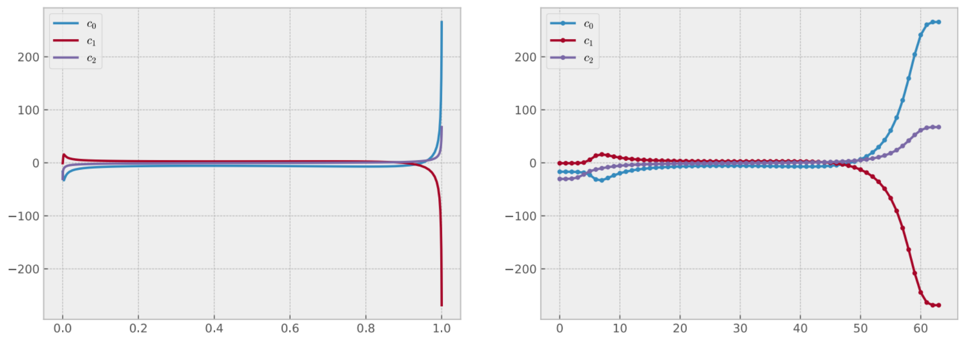


Figure 3.2: At the left we can see how fast the optimized $c_i$ coefficients in the 3D tables change as the $\alpha$ parameter gets too close to the 0 and 1 values. To mitigate this, Jakob and Hanika increase the resolution in those areas by delinearizing their mapping for this parameter. Image reproduced from the original work [JH19].

- Their work generates 3D coefficient tables that can be easily loaded in a shader inside a real-time rendering engine as look up textures.

- The generated tridimensional cubes are pretty lightweight, since they have $3 \cdot (64^3)$ elements each, which for 3 tables and 4 bytes per element, translates into $\approx 9.2$ megabytes of size. Modern GPUs and hardware can allocate space for textures in the order of gigabytes, so the memory footprint is kept small.

- The look-up cubes only need to be loaded once. After that, only a few look-ups are needed, along with the sigmoid that recovers the wavelegnth response from the coefficients.
- We can still use assets created with RGB rendering systems in mind, saving lots of time that would otherwise be employed in converting them into a spectral representation. By enabling the use of RGB assets and material we also enable saving the additional space that spectral assets require in comparison to the RGB ones.
- The final conversion step that returns a spectral response can easily be vectorized for leveraging SIMD (single instruction, multiple data) instruction sets, like those present in GPU devices.

The main challenge we would have to tackle for implementing this upsampling technique into our pipeline was that it has been designed for an offline rendering context (as en example, Mitsuba 3 [JSR+22b], a Monte Carlo-based offline renderer, uses this exact method for performing its spectral upsampling [JSR+22a].)

We had to devise a way to make it work in our real-time pipeline for the wavelengths we have selected, according to the strategy we just explained in Section 3.1 without compromising too much the final performance, making sure that the impact with respect to RGB rendering wasn't too excessive. Since we are targetting real-time performance, the first step would be to correctly load the coefficients in a texture that can be passed to a GPU shader.

After loading them, we retrieve the coefficients as follows in Algorithm 9:

**Algorithm 1** Look up table coefficient fetching algorithm

```
function FETCH($colour$)
    $i \leftarrow colour.argmax()$                          ▷ Find the largest component of the RGB colour
    $col\_r \leftarrow colour[(i+1) \mod 3]/colour.max()$
    $col\_g \leftarrow colour[(i+2) \mod 3]/colour.max()$
    $col\_b \leftarrow colour.max()$

    $colour\_norm \leftarrow [col_r, col_g, col_b]$
    return $table[i, colour\_norm]$                          ▷ Trilinearly interpolated table lookup
end function
```

The other challenge we faced while adapting this spectral upsampling technique is related to the non-linearity of the mapping for one of the dimensions in the 3D tables that we mentioned earlier. Since we are using tables with a resolution of $64^3$, some kind of interpolation is needed to get more precise coefficients and avoid introducing error in our upsampled spectra. But since we can't rely on automatic interpolation methods because of this remapping, we had to manually implement our own version for the interpolation in a way that the remapping was considered. This has several consequences on our implementation and its performance, which we will discuss later, in Section 4.4.

## 3.3 Material models

After implementing the wavelength sampling and the reflectance upsampling steps, we had to implement different material models, along with the underlying material system to support them. Since we are going for a physically based renderer, we tried to go for already validated, commonly used physically based materials, in order to have similar results when testing against offline renderers like Mitsuba [JSR$^+$22b], a physically based renderer designed and built for scientific and research purposes.

Please note that when we say Physically Based Materials, we aren't trying to say they are 100% physically accurate and follow the Laws of Physics, but instead, they are inspired by them. In real-time Computer Graphics it is common to label a material model as Physically Based if is based on microfacet theory, but doesn't have to necessarily follow some core principles, like energy conservation (for example, Disney's BSDF [MHH$^+$12, HMB$^+$15] fails to conserve energy but is considered a physically based model and is widely used in industry settings like Disney's Hyperion renderer [BAC$^+$18], Blender 3D [RF24], or Mitsuba itself [JSR$^+$22b]).

Most of the coefficients that are part of these material models are obtained from textures, which, after all, are RGB images with different sizes (they can also be sRGB, but everything we have stated so far still holds). Some of the coefficients might model properties like specularity to control which areas reflect light more than others, while others control reflectances, which is one of the most relevant parameters over the final colour in most material models.

Thanks to our implementation for real-time spectral upsampling, we are able to evaluate these material models in a wavelength-dependent context, while still using the widely extended and lightweight RGB textures. We manage to enable their use in spectral rendering contexts, without the need to perform any conversion over the original material textures, as we mentioned at the beginning of Section 3.2. Furthermore, we can combine spectrally uplifted materials with other techniques for wavelength-dependent phenomena, reducing even more the colour error in our scenes. We will show an example of this in Section 3.4.

### 3.3.1 Lambertian diffuse material model

The first model we wanted to implement was the Lambertian diffuse, since we wanted to be able to test some behaviours under a simple, validated model. Also, its only parameter is the diffuse reflectance, and since it can be passed as a texture, it matches perfectly our upsampling approach for reflectances, making this material ideal for testing purposes.

For mode complex models, like the Cook-Torrance, we might find some discrepancies with respect to a baseline renderer's results (i.e Mitsuba) that are caused by factors external to our spectral upsampling process (for example, the Fresnel-Schlick approximation doesn't take into account incoming light's spectral information, as we will explain in the section after this one), which is the main contribution we want to test in a real-time rendering environment.

Lambertian diffuses are characterized by their equal reflectancy of light among all directions, independently of the direction the light's coming from. Their mathematical formulation for this BRDF is the following, expressed in Equation 3.8 (we don't include the dot product $n \cdot \omega_i$ here):

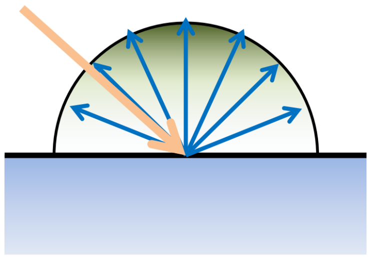


Figure 3.3: Diagram showing the reflectance distribution of lambertian materials, uniform among all directions.

$$f_r = \frac{albedo}{\pi} \tag{3.8}$$

### 3.3.2 Cook-Torrance material model

Inspired by the work of Brian Karis at Epic Games [Kar13], we decided to follow his implementation for a second, more complex physically based material. This PBR material is essentially an adaptation of the Cook-Torrance [CT82] material BRDF, choosing specific distribution functions to approximate some of its terms, which we explain through the current section. His adaptation of the material model has become widely used and popular, being the default PBR material for Unreal Engine 4 [Epia] and its latest iteration, Unreal Engine 5 [Epib].

The Cook-Torrance BRDF has two terms, a lambertian diffuse where light gets reflected evenly among all possible directions, and a specular one, where we take into account small surface imperfections, like microfacets and rugosities, that make light reflect in very different directions within a small patch of surface, depending on the tiny surface normals that appear as a consequence of these imperfections.

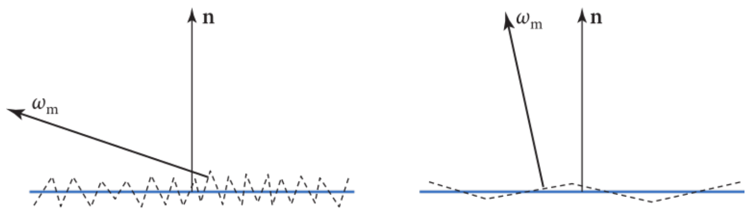


Figure 3.4: Two different surfaces where the geometric normal points in a different direction than the one of the microfacets in the surface. This affects the way light propagates, generating small interactions across the material's surface. In the Cook-Torrance shading model we can control the amount of displaced microfacets with the roughness parameter. Image courtesy of Matt Pharr, Wenzel Jakob, and Greg Humphreys under the Creative Commons License.

We can approximate that microgeometry with several models and functions that depend on few parameters and are easy to understand, making this material model ideal for commercial renderers aimed at the general public, like 3D artists.

Both terms in the BRDF (equation 3.9) fit our approach since we can easily upsample the RGB reflectance into spectral responses for the former, and the latter doesn't explicitly depend

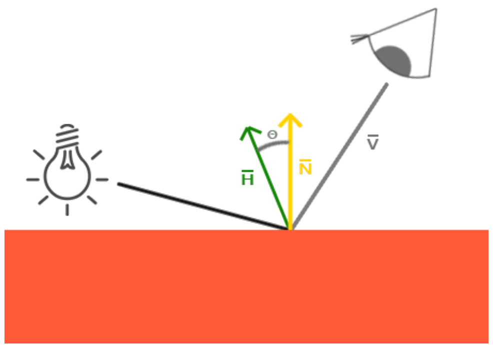


Figure 3.5: Graphical comparison of the halfway vector ($\overline{H}$) and the normal vector ($\overline{N}$). The halfway vector can be calculated with the average between the point-eye vector and the point-light source vector. Courtesy of Joey de Vries, under the Creative Commons License.

on wavelength, as it can be seen on equations 3.10 and 3.11.

The diffuse term is the same we explained in the previous section, we only need to divide the albedo reflectance by pi, since the dot product between surface normal and light direction gets accounted for later in the calculations (as seen in the Rendering Equation section, 2.1.1).

$$f_r = k_d f_{lambert} + k_s f_{Cook-Torrance} \tag{3.9}$$

$$f_{lambert} = \frac{albedo}{\pi} \tag{3.10}$$

$$f_{Cook-Torrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \tag{3.11}$$

For the specular term, we have three different coefficients in the enumerator and a normalization term in the denominator. These coefficients are approximations for different terms that depend on the view direction vector $v$ or $w_o$, the incoming light direction vector $w_i$, the halfway vector $h$, the normal vector $n$ and other parameters, which we are going to explain in the next paragraphs. The three terms correspond to:

- **$D$, the Normal Distribution Function:** Describes the distribution of surface microfacets that are aligned to the halfway vector $h$ (See Figure 3.5 for more details on the halfway vector). We use the Trowbridge-Reitz GGX [TR75] approximation for this term (See equation 3.12), where $\alpha$ represents the roughness of our surface. The more rough, the more chaotic the distribution of the microfacets will be, leading to diffuse-like results.

- **$F$, the Fresnel equation approximation:** In his work, Karis chose to use the Fresnel-Schlick approximation [Sch94] for the Fresnel equations, as shown in Equation 3.14. This approximation allows to save many computing cycles and instructions, and it is optimal for RGB rendering [Hof19], while spectral rendering would benefit from using the original equations, at the cost of a higher computing time. Nevertheless, we considered that following the original material model as it was proposed was the better option for us, since it would be more comparable to other implementations of this Cook-Torrance PBR material. This approximation relies on the halfway vector $h$, as well as a parameter called $F_0$, which represents the normal incidence at a 0 degree angle, which at the same time depends on the material's metalness, another parameter for the model. We chose to use a base value for $F_0$ of 0.04, which produces good results for most materials. In practice, $F_0$ acts as our $k_s$ coefficient for equation 3.9, and we should remove it from there in order to avoid taking it into account twice. We show it in the equations, since other Fresnel approximations might not take it into account and it could be left there. For completeness, $k_d$ is simply calculated as the complementary value of $k_s$.

- $G$**, the Geometry function:** This distribution function attempts to model the self-shadowing and occlusions caused by the surface's microfacets, reducing the total amount of reflected light. We are going to use the Smith's Schlick-GGX function to approximate this term [Sch94, Smi67], as seen on equation 3.13. Since we want to model two different phenomena, masking occlusions and shadowing (see left and middle diagrams in Figure 3.6), we need to compute this term twice, once for the view-point direction (masking) and once for the point-light direction (shadowing). This is known as Smith's method, which we illustrate in equation 3.15. Additionally, it can be noticed that there's an additional parameter $k$ we haven't explained yet. This parameter is related to the material's roughness parameter we introduced earlier, $\alpha$. It is some sort of remapping, which for our implementation (with no IBL [Deb02]) is the one in equation 3.16.

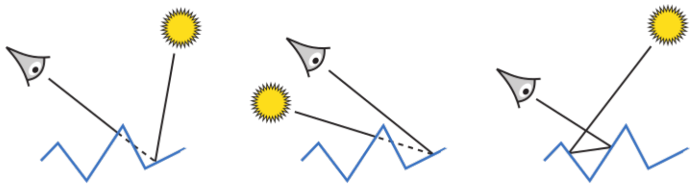


Figure 3.6: Three graphical examples of interactions of light with the microfacets in a surface (from left to right): Masking and shadowing, where a microfacet essentially blocks the view to the light our to our eyes, and in the right, interreflection, where light bounces among microfacets. The normal distribution function and the geometry function model these phenomena. Image courtesy of Matt Pharr, Wenzel Jakob, and Greg Humphreys under the Creative Commons License.

$$NDF_{GGX\ TR}(n,h,\alpha) = \frac{\alpha^2}{\pi((n\cdot h)^2(\alpha^2-1)+1)^2} \tag{3.12}$$

$$G_{SchlickGGX}(n,v,k) = \frac{n\cdot v}{(n\cdot v)(1-k)+k} \tag{3.13}$$

$$F_{Schlick}(h,v,F_0) = F_0 + (1-F_0)(1-(h\cdot v))^5 \tag{3.14}$$

$$G(n,v,l,k) = G_{SchlickGGX}(n,v,k)G_{SchlickGGX}(n,l,k) \tag{3.15}$$

$$k = \frac{(\alpha+1)^2}{8} \tag{3.16}$$

## 3.4 Participating media

Up until this point, we have applied and adapted spectral upsampling techniques for real-time rendering, and we implemented some material models that can benefit from this upsampling process, allowing us to render scenes spectrally by using only RGB assets in real time. Our approach can also be paired with other real-time spectral rendering techniques, benefiting the overall results thanks to being able to render most of the scene in a spectral manner, instead of combining spectral with RGB rendering. To show the potential that extending this technique by combining it with others might have, we implemented participating media in our spectral rendering pipeline, where scattering can be dependent on wavelength (elastic scattering), a phenomenon that traditional RGB rendering can't handle properly.

We devised a few ways to showcase this. We first took a look at conventional videogame fog rendering, based on a simple parameter-controlled mix between a predefined fog colour $c_f$ and the colour of that pixel $c_i$ after evaluating the corresponding material model on it, $c = fc_i + (1-f)c_f$. Usually, the parameter $f$ that controls the fog is equal to the linear depth of the object in that image pixel, even though it can also be obtained with an inverse exponential.

As it can be easily deduced, this is not a physically based approach, but an artistic-driven one. Hence, we determined that we should go in a different direction for our participating media.

After that, we took a look at the work of Monzón et al. [MGAM24] There, they propose a method for performing spectral rendering of oceanic underwater scenes in real time, which fits most of our desired purposes for our pipeline and could benefit from our spectral upsampling implementation. We take a deep dive into it in the next Section (3.4.1), and we later show the results of combining both approaches into our pipeline, in Section 5.

### 3.4.1 Oceanic underwater rendering

The work of Monzón et al. [MGAM24] devises a manner to use previously existing measurements done by oceanographers, to render oceanic water in real time while approximating single and multiple scattering in an analytical manner.

Their method is robust and highly performant, but it's only focused on the oceanic water volume and completely ignores any possible reflectances in the scene, like the ones of the oceanic floor models. Extending this method with our real-time upsampling allows to still use traditional lightweight RGB textures, while performing more accurate lighting computations, reducing the error with respect to using the RGB coefficients, or applying naive upsampling strategies on them.

This combination of both techniques also allows us to showcase the flexibility and usability of our work in all kinds of contexts, proving its robustness for several purposes, be it industry or research. We will later show some of the results we obtained, along with the appropiate error and perceptual error metrics.

In their work, two main assumptions are made: isotropic behaviour for the medium (this means that light gets scattered evenly across all possible directions at any point under the water) and homogeneous attenuation across the medium. For the sake of completeness, we need to briefly introduce some key concepts of their implementation that completes those we previously introduced about the VRE, in Section 2.1.2.

In oceanography, several properties of the water bodies are measured and evaluated. One of these properties is the diffuse downwelling attenuation coefficient $K_d$, a function that characterizes the decay in irradiance as a function of depth, and is wavelength-dependent [Mob94].

Oceanographer Nils Gunnar Jerlov measured this property for several water bodies and oceans around the world [JER51, JF60, Jer76, Jer77], giving birth to a classification of water types based on absorption, extinction and this $K_d$ coefficient. Those types are what we know as the Jerlov water types.

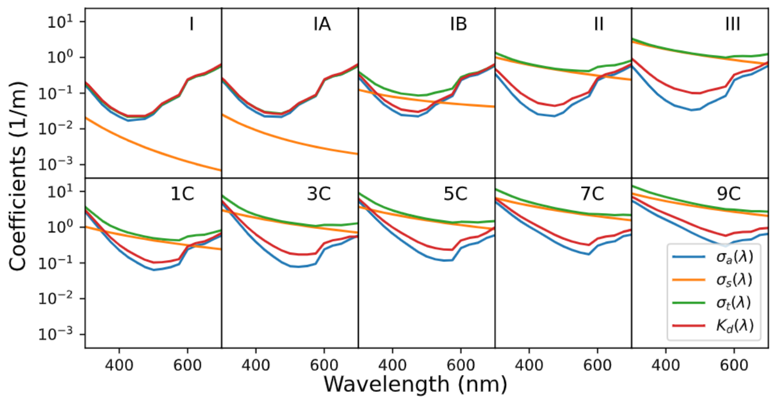


Figure 3.7: The 10 existing Jerlov water types, along with their measured coefficients and properties. In the top row, the open sea waters, and in the bottom row, the coastal types, ordered in decreasing scattering to absorption ratio (single scattering albedo).

Our version for their approximation accounts for attenuation and multiple scattering, both in the light ray that reaches the surface and in the ray that goes from our camera to the surface. It is worth mentioning that both in their original formulation and in our implementation, we are assuming that there is a single light emitter, which is perpendicular to the water surface (and thus, to the scene), pointing from above. This makes sense, since we are simulating oceanic underwater scenes. We also assume that the scene materials (which would represent the oceanic floor) are comlpetely diffuse, which is where our real-time spectral upsampling can benefit their method the most, generating spectral responses instead of having to work with rough translations from RGB values of the model's textures. The formula for our approximation of their method can be seen in equation 3.17:

$$L(O,\omega) = \frac{\sigma_s(E(0)e^{-K_d O_y})}{4\pi(K_d\omega_y - \sigma_t)}[e^{(K_d\omega_y-\sigma_t)S} - 1] + T(y_s)[E(y_s)\frac{\alpha}{\pi}(\cos\theta)] \tag{3.17}$$

There are several terms in this equation, which we proceed to explain in detail:

- $O$: The position of our sensor.
- $y$: Point in the oceanic surface where light gets reflected into our sensor or camera. Its depth with respect to the water surface is noted as $y_s$.
- $\omega$: The direction from the point in the surface to our camera sensor, normalized.
- $\sigma_s$: The per-wavelength scattering coefficient, in meters$^{-1}$. This is measured for every Jerlov water type.

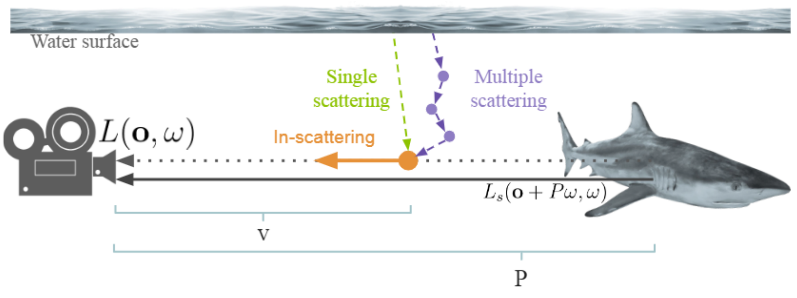


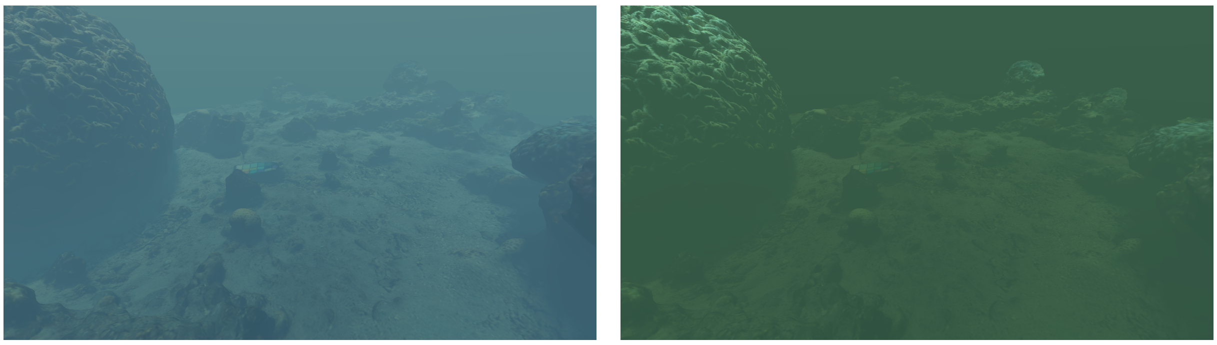
Figure 3.8: A quick overview of the in-scattering approximation in underwater scenes.

- $E(0)$: Incoming irradiance at depth 0, i.e the water's surface. Oftenly uses a standard emitter (the single light source in this type of scenes), like the D65 illuminant, meaning that its irradiance depends on wavelength.
- $K_d$: The aforementioned diffuse downwelling attenuation coefficient. Depends on the Jerlov water type we are trying to render, as well as on the wavelength.
- $O_y$: The depth our camera is currently submerged at. It can be seen as the $y$ component of the sensor position vector $O$.
- $\omega_y$: The depth (height) difference in the direction pointing from the surface point to the sensor. Can be seen as the $y$ component of the (before normalization) $\omega$ vector.
- $\sigma_t$: The per-wavelength extinction coefficient, in meters$^{-1}$. This is measured for every Jerlov water type.
- $S$: The distance along direction $\omega$ that separates our sensor's position $O$ from the surface point $y$.
- $T(point)$: Transmittance between the sensorand a point in the scene. We compute it using the Beer-Lambert law, as seen on Equation 2.4.
- $E(y_s)$: Incoming irradiance at depth $y_s$ after attenuation, which stands for the depth of our surface point. The formula for $E(\cdot)$ can be found in Equation 3.18.
- $\alpha$: The albedo of our surface at point $y$. We use our uplifted spectral values.
- $\cos\theta$: The cosine between our surface point $y$'s normal and light's incoming direction. Since we assume light coming always from the world's up direction, this term can be directly substituted by the $y$ component of the normalized normal $n$ vector.

$$E(y) = E(0)e^{-K_d\ y} \tag{3.18}$$

With this approximation for the VRE, we are able to render underwater scenes without almost impacting our application's framerate at all.

We will discuss further details in the performance and accuracy of our solution for underwater scenes rendering in Section 5, where we talk about the results of our rendering process and show



(a) Jerlov water type III. (b) Jerlov water type 9C.

Figure 3.9: Example of an oceanic scene rendered spectrally in real time with our pipeline under two different Jerlov water types. To the left, type III water, turbid, open sea. On the right, type 9C water, very turbid, coastal. Both are assuming an average human eye observer.

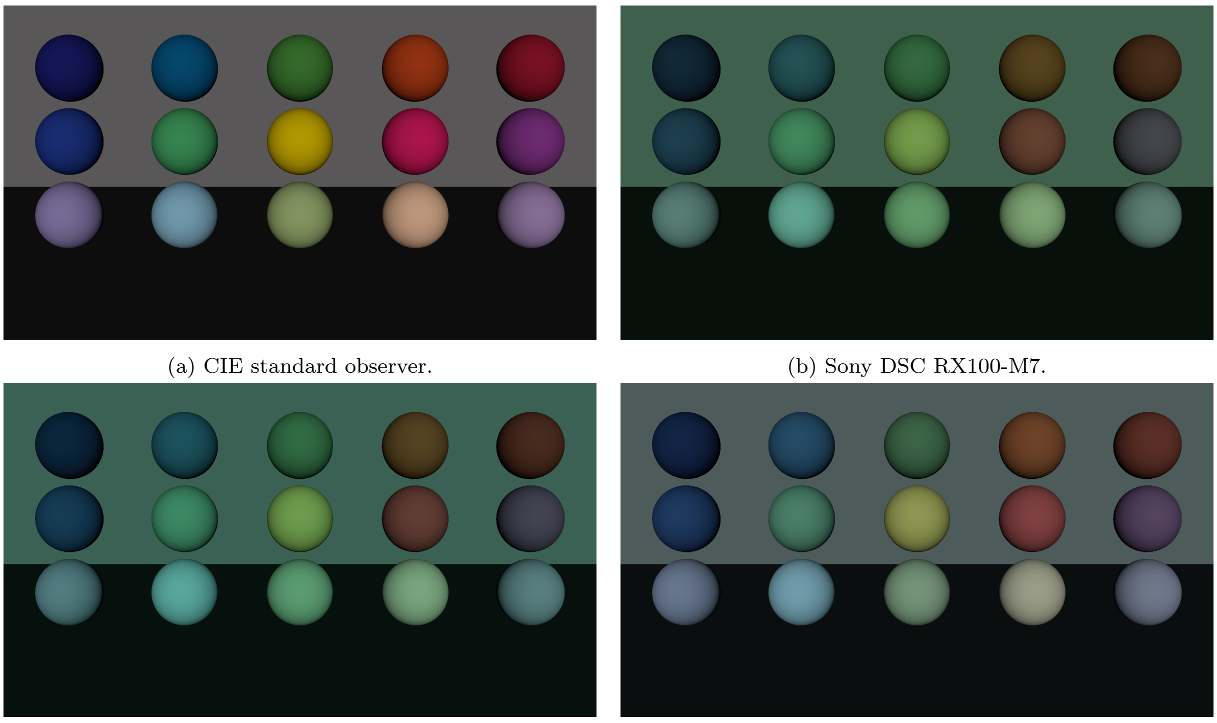
different comparisons in order to validate its functionalities and applicability to different industry contexts.

## 3.5 Sensitivity curves

As a last note on our renderer's implementation and details, we wanted to mention that we enabled the ability to simulate different spectral response curves for our cameras.

We take advantage of the fact that spectral rendering produces an SPD (Spectral Power Distribution) that we can multiply by any set of spectral response curves, like the CIE colour matching functions, as we explained in Section 2.2, to achieve this.

We preload a set of spectral response curves for different sensors, including the CIE 1931 colour matching functions and other commercial cameras, mostly those popular in oceanography, thanks to the work by Grigory Solomatov and Derya Akkaynak [SA23], who created a dataset for the estimated response curves of over a thousand cameras. By preloading them, we can swap them at runtime with no impact on performance, allowing for faster debugging times and better visualization for the final users, avoiding the need to rerun the renderer when needed.


(a) CIE standard observer.

(b) Sony DSC RX100-M7.

(c) Canon 40-D2.

(d) Canon 400-D2.

Figure 3.10: Comparison among different observers for the same underwater scene, rendered with our pipeline implementation in real time. The camera response curves have been obtained from the work of Solomatov and Akkaynak [SA23].

# 4. Real-time spectral rendering pipeline implementation

So far we have offered a look into the theoretical foundations behind our proposed real-time spectral rendering pipeline. In this chapter, we are going to dive deeper into the details of how we managed to implement our rendering pipeline based on spectral upsampling.

We have named our final implementation of the renderer "sRAT-RT", which stands for *"Spectral Renderer for All-Purpose Tasks, in Real Time"*. As the name suggests, we tried to go for an implementation that could be used in different contexts, with a wider scope than just this thesis. However, this is still an early version that would require more work to truly achieve its purpose.

We will offer details on the design, the most relevant parts of the implementation, and all the thought process and the decisions that we had to take during development, always trying to make the renderer scalable to prove that it could grow into a production-ready software.

## 4.1 Tools selection

We began the project with a decent amount of experience in rendering, although almost all of it was in offline rendering, mostly related to Monte Carlo path tracing algorithms. Real-time rendering was a completely new landscape for us at the time of starting this thesis a few months ago, which is also the reason we chose to dive into it. Fortunately, part of the knowledge is transferable, like some material shading models, while other things like rasterization and the different passes that are performed in the rendering process had to be learnt from zero.

Thus, the selection of the correct tools for developing the project was a critical task, since making a wrong choice would cost more time than if we already had some experience. We had two main options:

- Using a preexisting 3D engine, like Unity 3D [Tec05b], Godot [Fou07], Unreal Engine [Epia, Epib] or Panda3D [Dis02].
- Building a rendering engine from scratch, using modern rendering APIs like OpenGL [Gro92], Vulkan [Gro16] or DirectX [Mic95].

Both options presented many pros and cons. While using a preexisting engine can be seen as faster and more desirable, it is a double edge sword, since the codebase isn't necessarily designed to be modifiable to fit specific needs. Unity 3D has what it calls the Scriptable Rendering Pipeline (SRP) [Tec05a], and Godot is free and Open Source, but still young, slightly unstable and needs better documentation. Also, 3D engines have lots of features that aren't required for this project, like audio or physics engines, unnecessarily increasing complexity and potential error causes.

Figure 4.1: Some 3D engines, frameworks and APIs that we considered for our spectral rendering pipeline.

On the other hand, building from scratch a 3D engine is time consuming and it is necessary to have knowledge on the chosen API. Furthermore, some of them aren't compatible with every system. For example, DirectX is only available for Windows platforms. The same happens with Nvidia's Falcor [KCK$^+$22], a 3D framework designed for research, but unluckily only Windows computers that have ray-tracing capable GPUs can use it. Additionally, in order to reduce the workload in trivial tasks like 3D model loading or user interface and window management, we would need to find libraries compatible with our tools of choice, in the case of opting for building a 3D engine from scratch.

It was decided that we would go for an implementation in OpenGL, given that it's one of the oldest standards in the industry and could provide a good foundation for learning thanks to the immense documentation and resources available online [dV16]. We also decided to use C++ [Str79] as the language to operate the OpenGL API, given our experience on it.

### 4.1.1 OpenGL

OpenGL (Open Graphics Library) can be referred to as an API or a library, but it is a standard specification that defines a graphics API and its behaviour. It is up to the manufacturers and drivers developers to implement that behaviour in the way the find the most appropiate.

It is designed to behave as a great state machine that programmers can use or set as they want, in order to enforce one behaviour or another.

Shaders that are executed in the GPU must be written in GLSL (OpenGL Shading Language), a language designed for OpenGL shaders which has a similar syntax to basic C, adding

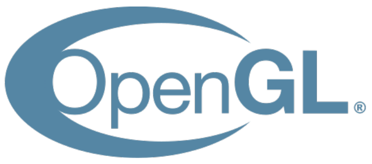


Figure 4.2: The logo of OpenGL.

some features like vectorial and matrix data types, and also enforcing many limitations related to GPU architecture and memory layout.

As we foreshadowed, additionally to OpenGL and C++ we incorporated into our project several libraries to reduce work on tasks that were orthogonal to the main point of the renderer, which is spectral rendering. These tasks include 3D model loading, OpenGL function wrapping, creating and managing windows, and creating a small user interface to control the test programs.

The list of libraries we used is the following:

- **mINI** for reading and writing `.ini` configuration files [Dur24].
- **Tinydir** for quickly traversing directories and reading their contents [tin24].
- **Assimp** for quickly loading 3D models in different file formats [ass24].
- **Glfw** was our choice library for window creation and management, specifically targetted at OpenGL [glf24].
- **GLAD** is a library that helps managing some of the low-level work behind OpenGL. Since OpenGL is technically a specification, several slightly different implementations exist for the same version, depending on the hardware and software manufacturers. GLAD helps making those differences transparent to us [gla24].
- **STB Image** is a library targetted at loading and saving images in different file formats. We used it to manage texture loading, as well as saving screenshots of our results [stb24].
- **GLM**, also known as OpenGL Mathematics, is a mathematical library that allows us to use different basic types like 2D, 3D and 4D vectors and matrices. It is specifically designed to be used with OpenGL code and GLSL shaders [glm24].
- **Mitsuba 3**, a research-oriented, multi-purpose offline renderer available for free that is physically based and oftenly used as ground truth for performing validation and comparisons [JSR+22b].
- **RenderDoc**, a debugging program that allows to profile the rendering process, as well as enabling visualization of some intermediate rendering stages and their partial results. With this program we performed most of the debugging tasks [Kar24].

## 4.2 Forward and deferred shading pipelines

When designing the rendering pipeline, we wanted to make it flexible and capable of offering the same range of features a modern one. In real time rendering, there are two main ways of rendering objects in a scene: Forward shading and deferred shading [DWS+88]. Each one has its own positives and negatives.

Forward shading has been for long the traditional way of shading objects, performing all the calculations in a single pass and directly offering a final pixel colour as a result. It also allows for easy implementations of algorithms like those focused on anti-aliasing [JESG12]. Its main downside is that it doesn't get any information about the scene's geometry, which results in wasting lots of time doing lighting calculations for fragments (potential final pixels in the image) that are not going to appear in the final image. Additionally, its cost is around $\mathcal{O}(f * l)$ for a scene with $f$ fragments and $l$ lights.

Deferred shading, on the other hand, handles lighting differently. As its name indicates, it defers those lighting calculations to a second pass. Its first pass collects information on the scene's geometry on a data structure called the G-Buffer, and does so on a per-fragment basis. This way, we can have access to the position, normals and other variables of the fragments that are going to be shaded on the second shading pass, having discarded the rest on the first, cheaper pass. Lighting cost is decreased to our rendering resolution times the amount of lights, $\mathcal{O}(px * l)$, with $px$ being the number of pixels in our screen. There are some downsides to this approach, like for example having only one shader for all the materials. We can have different materials by implementing an ID system in any buffer, such as the stencil buffer, a programmer-controllable buffer that can store arbitrary information. As we explain later, we implemented this ID system in the G-Buffer.

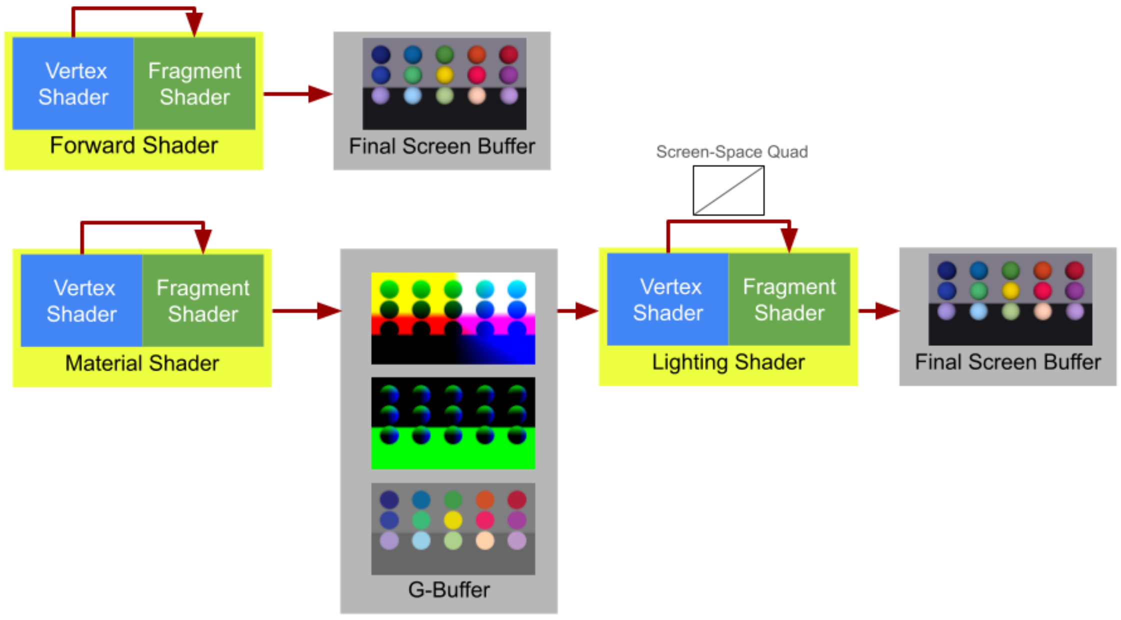


Figure 4.3: Comparison between forward (top) and deferred shading (bottom). Each shader (program that lives in the GPU composed of two stages called *vertex shader* and *fragment shader*) execution represents what is known as a *pass* or *draw call*. Deferred shading only requires rendering two more triangles than forward shading due to the second pass being in screen-space one, which is a negligible cost.

Given that our rendering process needs to upsample the reflectance for every fragment, it makes more sense to do it only for the actual fragments we need to render, instead of the ones we will have to discard for being occluded by other fragments. Since spectral upsampling comes with an associated temporal cost (see Section 4.4 for details on texture lookups), heavily reducing the number of fragments we have to compute lighting for is a great optimization by design that we can use.

With this information, we decided to go for a deferred shading approach, resulting in at least 2 passes over the scene's geometry. We detail in the following sections anything necessary to understand our implementation for the rendering pipeline itself, including the material models and the contents of our G-Buffer.

### 4.2.1 Our rendering pipeline: Render passes

We have already discussed that we would use the deferred shading technique in order to reduce the costs associated to our upsampling process. Now, we offer more insight into the implementation of our final rendering pipeline. Please, note that when we refer to a render pass, we are referring to the process of passing the scene through the entire rasterization pipeline, for a different purpose each time. Some passes are destined to collecting geometrical information, while others only perform colour corrections over the results of previous passes.

Most of this work could be done in a big, single rendering pass, but we elected against this approach since it would hurt modularity and extending some rendering techniques would be extremely difficult. Most industry renderers also perform several rendering passes with this in mind, effectively combining deferred and forward rendering depending on the intended appearance for each material.

**First pass: Filling the G-Buffer** As we have implemented it, we have the first two passes: In the first one, we process all the geometry in the scene to fill the contents of the G-Buffer with the necessary geometric and positional information that the next pass will need in order to carry out the actual rendering. The contents that we allocate in our G-Buffer are the following:

- World-space positions for each fragment.
- World-space normals for each fragment.
- Albedo reflectance (sampled from the corresponding textures) for each fragment.
- Other relevant per-material information (See Section 4.5 for details).

A visualization of the contents of our G-Buffer can be seen in Figure 4.5.

**Second pass: Shading proper** It is not until the second pass that we carry out any spectral rendering. Here we can read all the information we deposited into the G-Buffer during the previous pass and carry out the rendering as we know it. In this stage of rendering, we get other data passed into our shaders, like the look up tables we use for performing the upsampling step,

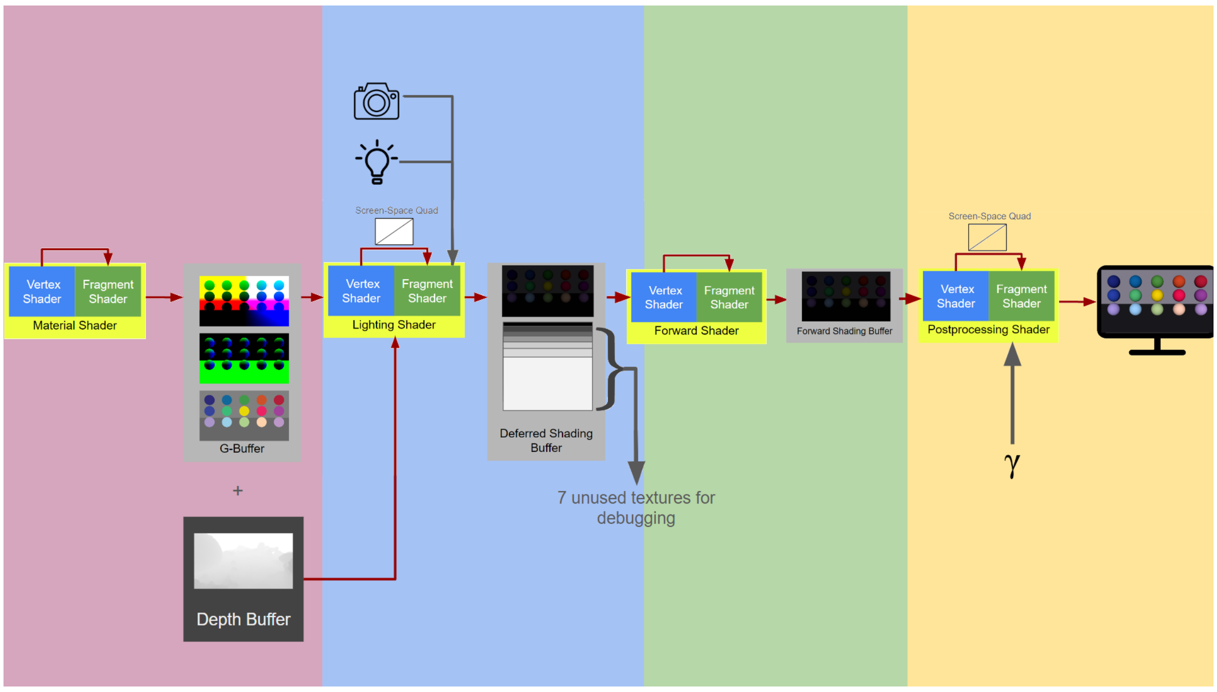


Figure 4.4: Diagram of our 4 render passes and the functions performed in each one. As we show, the actual rendering takes place in our second pass (in blue), which is where most of our explanations focus on. The first pass (in pink) is needed for geometrical information in deferred shading, while the fourth pass (yellow) exists for RGB to sRGB gamma ($\gamma$) conversion and further post-processing techniques.

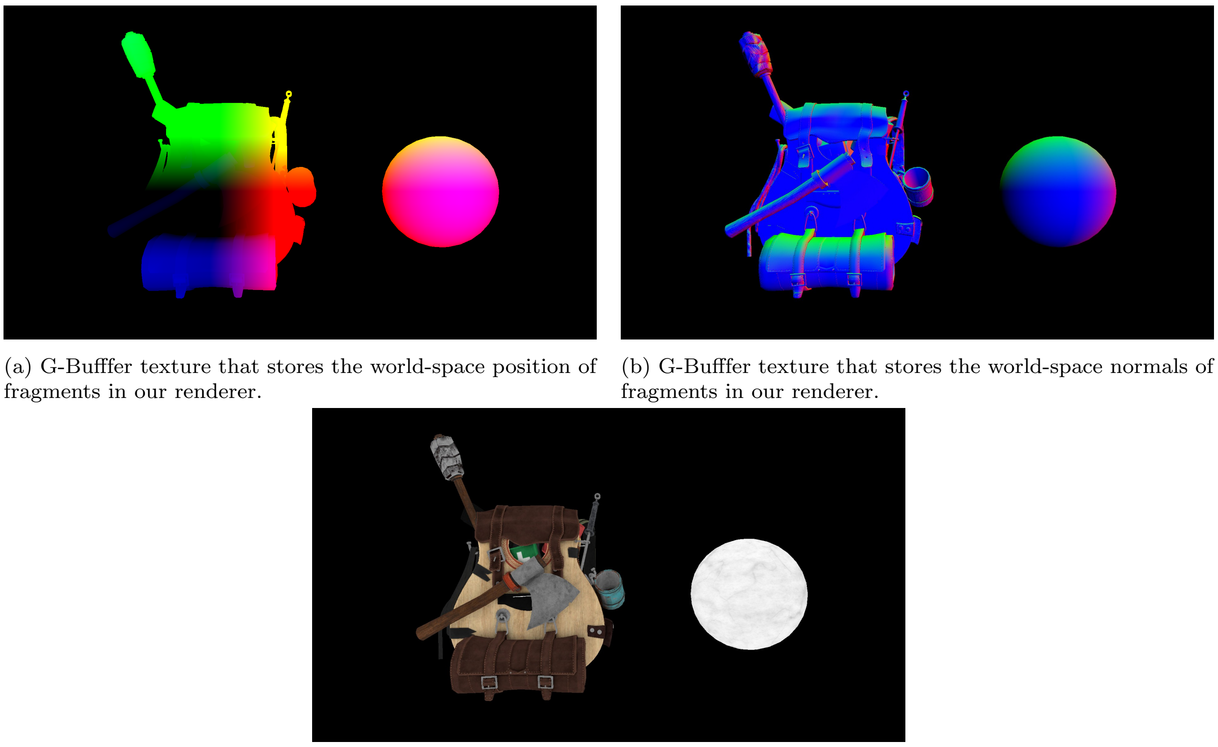
information about the response curve we want to apply, which wavelengths we want to select for the spectral rendering, and so on.

After getting all that data passed, we carry out the spectral rendering as explained through the last Chapter. The implementation details for the most relevant rendering steps (material models, underwater rendering, etc.) are detailed in the following sections.

It is worth mentioning that, since we have all the information we need, we can omit the geometry, saving lots of processing time and power. For that end, we render a screen-space quad composed of two triangles on which we project the final render. We don't even need to create a mesh with two triangles and store it somewhere, since we can create it on the fly in our vertex shaders.

**Third pass: Forward pass** We aimed for a renderer that was able to reproduce industry-level results. For that end, we also allow for an additional render pass that uses forward shading, in case that the need for implementing a technique (i.e antialiasing) that works better under forward rendering.

In order to correctly perform a forward pass after the deferred ones, we needed to copy the contents of the depth buffer from the first two passes into the buffer for this new pass, so that depth gets accounted for and we don't draw some objects on top of others, producing incorrect results, as we illustrate in Figure 4.7:

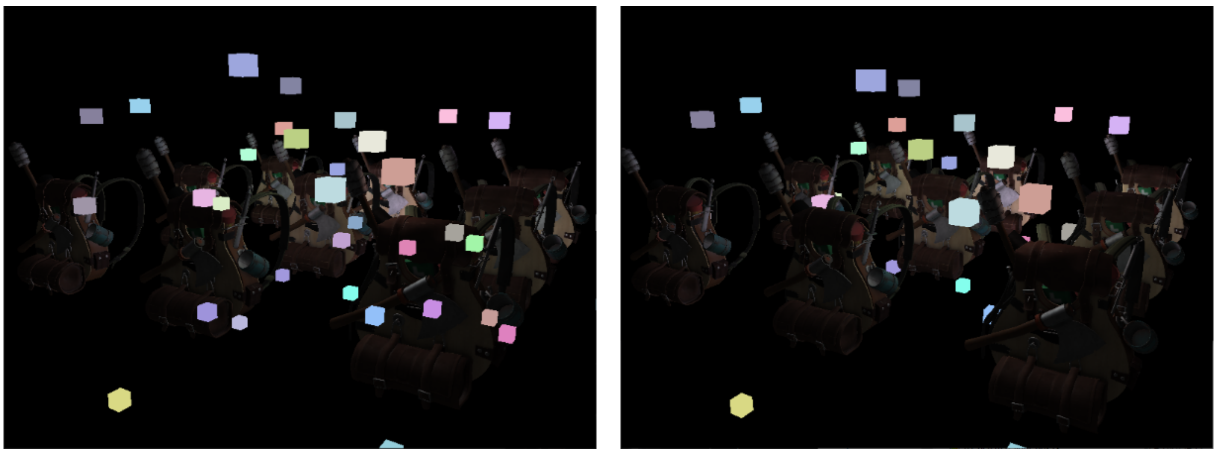


(a) G-Bufffer texture that stores the world-space position of fragments in our renderer.

(b) G-Bufffer texture that stores the world-space normals of fragments in our renderer.

(c) G-Bufffer texture that stores the albedo reflectances of fragments in our renderer.

Figure 4.5: Reference figure for our G-Buffer when rendering a scene in our implementation. We store in it per-fragment geometrical information of the models to render, as well as other information like the IDs of the material associated to the model appearing in each fragment.



(a) Incorrect depth information leads to objects rendering on top of each other when they shouldn't.

(b) Copying depth buffer information from the deferred lighting pass to the forward pass returns correct results.

Figure 4.6: Comparison between renders with incorrect depth and correct depth information. On the left, we don't consider the depth information of the deferred pass for the forward pass, while on the right we copy the contents of the depth buffer, resulting in a correct results where lights are rendered on top of the objects only when necessary.

However, we ended up not needing to perform any forward shading and deferred shading sufficed for all our needs in our final implementation. However, we chose to leave it for future extensions of the code, to make it easier supporting new features like the already mentioned antialiasing.

**Fourth pass: Postprocessing pass** After being done with the rendering, we obtain an image in RGB colour space. The main problem is that usually displays need the images to be encoded in sRGB colour space. The sRGB colour space is an alternate version for the RGB one, where colours are gamma corrected to account for nonlinearities in human perception of brightness. This is, for a given colour **c** with channels $r$, $g$, and $b$, we obtain the gamma-corrected colour $c'$ as $c' = \{c.r, c.g, c.b\}^{1/\gamma}$. Common values for $\gamma$ are 2.2 or 2.4.

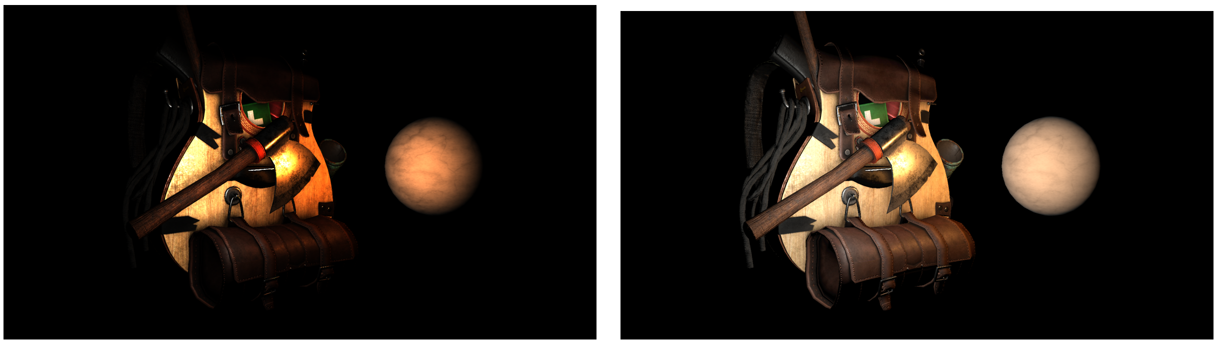

(a) Result of our rendering pass, in linear RGB colour space. (b) The same render in perceptually-linear sRGB space.

Figure 4.7: Comparison between the output of the spectral rendering before and after performing gamma correction in the postprocessing pass. We use a value of 2.2 for our correction parameter $\gamma$.

We perform this step in an additional render pass, that we can also use for other colour correction and postprocessing operations, like Reinhard tone-mapping [RSSF02] if we have HDR values in the image, posterization, dithering, and so on. Keeping it separate from the rest of our rendering process allows us to obtain a clean and modular code that will be more mantainable and extendable in the future. In our implementation for this thesis, we only need to apply the sRGB mapping and no other colour correction operands have been applied to our final images.

## 4.3 Wavelength sampling implementation

We discussed the main details of our uniform wavelength sampling in Section 3.1, and now we offer some details on how we made it work within our renderer.

Since we are aiming for real-time performance in our implementation while keeping the highest possible precision in our results without impacting the framerate, being able to experiment with some parameters is vital. The parameter with the biggest influence over both our final framerate and the quality of the final image is the number of wavelengths we want to use for rendering our scenes, $n_{wls}$, as we mentioned when explaining the use we gave to the rectangle rule in Section 2.2.1.

Thus, our implementation should be able to dynamically change the number of wavelengths we use for rendering if the users want to. It would also allow us to prototype and debug faster, and during the validation and data recollection stage of development. We control this $n_{wls}$ parameter from the user interface we added into the application.

The main challenge we faced when implementing this part of our pipeline is that GLSL shaders don't allow allocating a varying number of elements (array of $n_{wls}$ floating point numbers representing wavelengths in this case) as a uniform variable. We had two choices to circumvent this:

- Assign a fixed size to our wavelengths array, at the risk of overshooting and keeping that memory unusable for other textures and variables, or undershooting (even though this is unlikely if we want real time performance).

- Allocate the wavelengths in a texture, since they don't present any size limitations.

The second option made more sense, in order not to misuse memory space. This way, at the beginning of the execution and every time users change the number of wavelengths ($n_{wls}$ as we mentioned in Section 3.1) they want to render with, a new one-dimensional texture gets generated in the CPU side of the program with as many texels (texture pixels) as wavelengths we want to sample, storing the resulting wavelengths (in nanometers) in the red channel, `GL_RED`. We also need to disable texture interpolation in order for this workaround to work properly (otherwise it wouldn't behave as an array). This is not slow, as it could appear at first sight, since we are generating at most 200 texels and 8 to 24 in average, and we only do so every time users want to change the parameters, instead of doing it every frame.

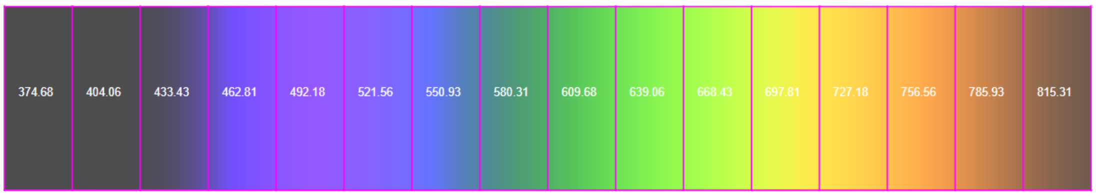


Figure 4.8: An example showing 16 wavelengths chosen uniformly in the range of 360 to 830 nanometers.

For now we only allow performing uniform sampling (as explained in Section 3.1), but we have prepared our program to allow for other strategies in case we extend our work in that direction in the future. We are conscious that performing uniform sampling in the CPU might be slower than letting the shaders in the GPU handle everything, but for other more complex strategies that might need to take into account temporal information it's the better choice. So, it is a design decision looking into the future of our implementation instead of its current state.

As a final note, we should mention that we locked the number of wavelengths to be always a multiple of 4, in order to leverage the SIMD (single instruction, multiple data) nature of GPU instructions that allow using vectorial types, like `vec4` in GLSL. In figure 4.9 we show an example on how the uniform wavelength sampling behaves over the visual spectrum (in this case the CIE 1931 colour matching functions).

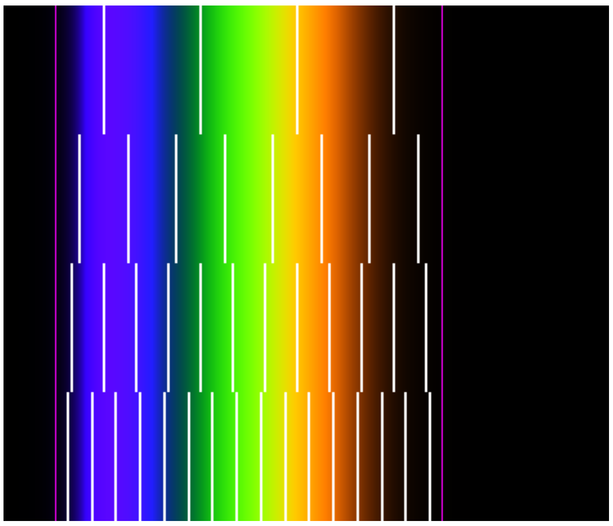


Figure 4.9: Example of how the uniform wavelength sampling behaves over the visible spectrum. The two lines in purple represent our thresholds for spectral rendering, $wl_{min}$ and $wl_{max}$, while the white lines represent our chosen wavelengths for varying values of $n_{wls}$. From top to bottom: 4, 8, 12 and 16 wavelengths.

## 4.4 Spectral upsampling implementation

In Section 3.2 we explained the basic concepts behind spectral upsampling. To sum it up, spectral upsampling allows us to go from RGB reflectance values in regular material textures to spectral coefficients that we can use in a spectral rendering environment.

We also mention that we take as a starting point previous work by Jakob and Hanika [JH19], where they present a method to optimize the upsampled spectra to minimize error, generating smooth spectra from 3 coefficients that encode the smooth spectra via a sigmoid function. Those 3 coefficients are the values they optimize, storing them in 3D look up tables.

Our main contribution consists in validating the usability of their optimization for real-time rendering, enabling the use of RGB assets in real-time spectral rendering. For that end, we load those coefficient cubes into 3D textures, a native type offered by OpenGL.

Since OpenGL supports 3D textures as a native type, we only have to care about reading the files that store the coefficient tables correctly for the texture creation step, something we only do once at the beginning of the renderer's execution, and sampling them appropriately in our fragment shader at each frame. Most of the GLSL shader code we use for our upsampling process is actually a direct translation from the original C code and the pseudocode in the paper.

There is, however, a small catch; a direct consequence of Jakob and Hanika's design, as we mentioned in Section 3.2. Since the brightness ($\alpha$) dimension in the 3D tables presents a nonlinear mapping, we can't rely on OpenGL's automatic texture interpolation. We had to disable it and manually perform a trilinear interpolation, which means that we need to perform 9 texture lookups in the fragment shader instead of relying in OpenGL's interpolation, which most likely is much more optimized than our manual method but can't know about this dimension's nonlinearity, introducing additional overhead.

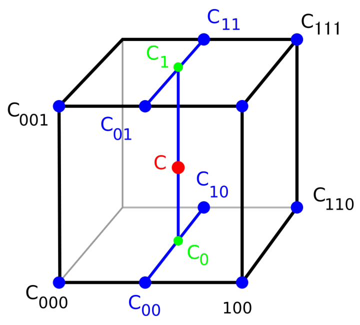


Figure 4.10: A visual depiction of trilinear interpolation, for which up to 8 texture evaluations are necessary, which in the figure would correspond to points $C_{000}$, $C_{100}$, $C_{110}$,$C_{010}$, $C_{001}$, $C_{101}$, $C_{111}$ and $C_{011}$.

This problem could be mitigated by modifying Jakob and Hanika's table generation implementation so that the colour brightness $\alpha$ dimension is linear, at the cost of losing precision in the extremes, which present a bigger variability (as we told in Section 3.2. Additionally, this could be mitigated again by increasing the resolution of their tables, currently set at 64x64x64 triplets of 3 floating point values (4 bytes each). As it can be easily seen, this option would increase in a cubic manner the memory usage within the fragment shader, and since we want the uplifting step to be as lightweight as possible, we chose against modifying the resolution. In the end, we would have to make sacrifices and reach a compromise.

It could be interesting to benchmark the impact on performance of varying resolutions, to try and find the best possible compromise between time lost in texture lookups and memory usage due to the texture's resolution, but for now we are not interested in getting a perfectly optimal implementation, instead we focus on getting a working one that shows that this uplifting technique is usable in real time and that it can be used along other spectral rendering techniques.

### 4.4.1 Rendering with the upsampled reflectances

In practice, we first fetch the 3D textures with the RGB albedo values we saved in our G-Buffer and save the obtained coefficients, and then we begin the spectral rendering. During it, we iterate through all our chosen wavelegnths that we stored in a 1D texture previously. For each wavelength we apply the sigmoid reconstruction to the coefficients we computed in order to get the spectral response to a specific wavelength. After that, we evaluate the corresponding material model with the current wavelength, we multiply the resulting radiance by its corresponding spectral response (we discuss the implementation of response curves in Section 4.7), and we accumulate that final value in a sum.

When we're done iterating over all the necessary wavelengths, we perform a last division on the accumulated value that completes our Riemann sum. Internally, we convert everything into the XYZ colour space, so the last step in this render pass is converting back to linear RGB. We detail this process later, after explaining the implementation of the process of applying sensitivity curves, since both are dependent on each other. We describe the process in Algorithm 2.

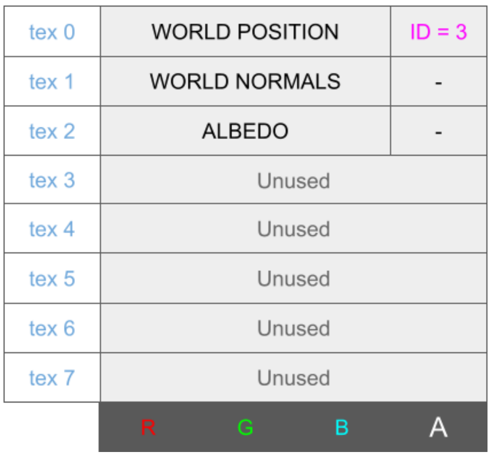
## 4.5 Material models implementation

In Section 3.3 we discussed the material models that we would use in our renderer. We also discussed that we found their use appropiate, since we could enable the evaluation of these material models under spectral contexts even though the materials themselves are authored using RGB resources.

Now, we offer more insights on the implementation details. More concretely, we are going to discuss the G-Buffer layout, and how we managed to support different materials with a screen-space deferred pass that is common to all of them through the use of material IDs.

### 4.5.1 Lambertian diffuse material model in our renderer

As we introduced in Section 3.3.2, this is a rather simple material model, and for implementing it we only require a single albedo texture along with the corresponding material ID (which we have decided to set as 3 for this material) for our lambertian BRDF's implementation. We store the albedo RGB texture and the material ID in the G-Buffer in our first pass, and later we do the lighting calculations in our screen-space shader, according to equation 3.8. The G-Buffer layout for this material looks like the representation in Figure 4.11.



| | R | G | B | A |
|---|---|---|---|---|
| tex 0 | WORLD POSITION | | | ID = 3 |
| tex 1 | WORLD NORMALS | | | - |
| tex 2 | ALBEDO | | | - |
| tex 3 | Unused | | | |
| tex 4 | Unused | | | |
| tex 5 | Unused | | | |
| tex 6 | Unused | | | |
| tex 7 | Unused | | | |

Figure 4.11: Data layout inside the G-Buffer upon evaluating a diffuse material model. Less parameters are used than for Cook-Torrance materials. We're also multiplexing some textures so that they hold different information depending on the colour channel. Parameters in black are common for all materials, and no specific parameters are required for this material. The material ID is fixed to 3. We leave space in the G-Buffer for possible future extensions that might need to use those textures.

### 4.5.2 Cook-Torrance material model in our renderer

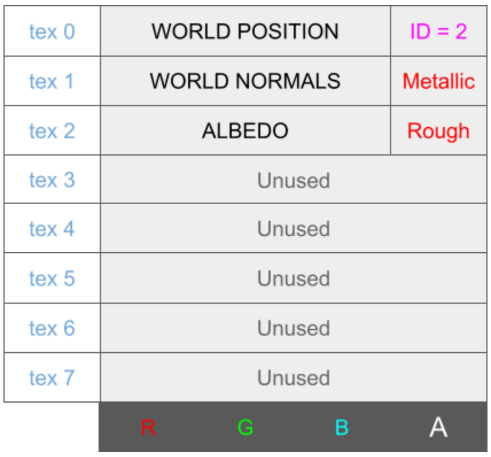
As we have shown in its theoretical section (Section 3.3.2), this material model is extremely useful and powerful, and can be used to model the appearance of several materials and surfaces, while depending on few parameters. As a reminder, these are:

- **Albedo**, its diffuse component colour, can be RGB or spectral after uplifting it.

- Metalness, which is oftenly referred to as **metallic**.
- **Roughness**, which determines the microfacet distribution.
- $F_0$, which in our case is always equal to 0.04, since it adjusts well to most cases.

These parameters (except $F_0$) can be passed to our shader as textures, so that they are positionally dependent inside the G-Buffer on the model we want to render. This is the standard for most PBR pipelines. As we mentioned in Section 4.2.1, we also include information on the screen space normals of the geometry, to save in detailing that otherwise should be included in the model's geometry.

We the necessary textures in our G-Buffer in the geometry pass, and render the final image in the second lighting pass. This means that we need to have two shaders for this material. One, corresponding to our `PBRMaterial` model class that sets the G-Buffer textures and values accordingly, and another shader to light the scene. As it commonly happens with deferred rendering, the second shader, the one responsible for doing material lighting, has to deal with not knowing which material it's lighting. Therefore, it has to be a single shader for every type of material, and not a per-material shader. In industry settings it is called a supershader or *übershader*. We handle this by storing a material ID in the alpha (transparency) channel of our world positions texture in the G-Buffer, which is normally unused.



| | | |
|---|---|---|
| tex 0 | WORLD POSITION | ID = 2 |
| tex 1 | WORLD NORMALS | Metallic |
| tex 2 | ALBEDO | Rough |
| tex 3 | Unused | |
| tex 4 | Unused | |
| tex 5 | Unused | |
| tex 6 | Unused | |
| tex 7 | Unused | |
| | R G B | A |

Figure 4.12: Data layout inside the G-Buffer upon evaluating a Cook-Torrance material model. More parameters are used than for Lambertian diffuses. We're also multiplexing some textures so that they hold different information depending on the colour channel. Parameters in black are common for all materials, while in red are specific for this one. The material ID is fixed to 2. We leave space in the G-Buffer for possible future extensions that might need to use those textures.

## 4.6 Oceanic Underwater Rendering Implementation

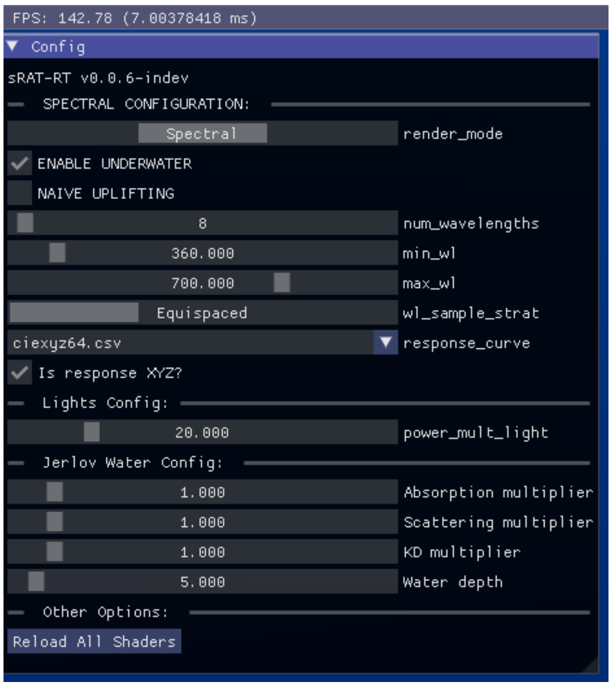
Earlier, in Section 3.4.1, we talked about how we built upon the previously existing work of Monzón et al. [MGAM24] for spectral rendering of underwater scenes in real time. In this case, by combining their technique with our real-time spectral upsampling we can achieve better results by being able to also render spectrally the models in the scene which are assumed to be diffuse when underwater. Since we assume them to be diffuse, their main property is the reflectance we are upsampling into the spectral domain.

For implementing their analytical approximation of volume rendering along with our spectral upsampling pipeline, we needed to load and pass some data into our shader:

- The absorption coefficient $\sigma_a$ corresponding to the desired Jerlov water type.
- The extinction coefficient $\sigma_t$ corresponding to the desired Jerlov water type.
- The diffuse downwelling attenuation coefficient $K_d$ corresponding to the desired Jerlov water type.

As a reminder, all this data is wavelength-dependent, so we encoded everything in textures for our OpenGL implementation. We also enabled rendering an RGB version of the scenes and the water types, so we also passed the same data as RGB triplets.

We also allow users to increase and decrease the values of these coefficients in the scene from the graphical interface, as well as allowing them to change the depth we are simulating, making the scene appear clearer or darker, depending on how close to the water surface we want to simulate our scene to be.



```
FPS: 142.78 (7.00378418 ms)
▼ Config
sRAT-RT v0.0.6-indev
— SPECTRAL CONFIGURATION:
Spectral            render_mode
✓ ENABLE UNDERWATER
  NAIVE UPLIFTING
8                   num_wavelengths
360.000             min_wl
700.000             max_wl
Equispaced          wl_sample_strat
ciexyz64.csv ▼      response_curve
✓ Is response XYZ?
— Lights Config:
20.000              power_mult_light
— Jerlov Water Config:
1.000               Absorption multiplier
1.000               Scattering multiplier
1.000               KD multiplier
5.000               Water depth
— Other Options:
Reload All Shaders
```

Figure 4.13: Our graphical user interface for controlling several parameters of our scenes, including multipliers for the water scattering, absorption and diffuse downwelling attenuation coefficients.

There was one main issue with our implementation for underwater rendering, however. Since the scenes in this renderer aren't the real world, we can be staring into the void, which with our deferred shading pipeline would mean that we can get [0,0,0,1] as our fragment position's *vec4* value in the respective texture in our G-Buffer. This means that we can't recover a position for those fragments in the infinity, when it should be our camera's far plane (the plane that sets the limit of how far our camera can see). We had to find a workaround for this specific case. Our solution was to use the depth buffer from the first deferred pass to recover the fragment's positions only when they were in the void (and with fog toggled on, since it's the only time we care about them). For this, we needed to recover the depth buffer from the deferred geometry rendering pass, and pass it as a new uniform variable to our shader. We also needed to pass the inverse view and the inverse projection matrices of our camera, which we can get by inverting the original ones that we compute at each frame.

## 4.7 Applying sensitivity curves

One of the last steps in our spectral rendering process is applying a sensitivity curve to our obtained spectral distributions, in order to get a final set of RGB values for our pixels, which we later convert to sRGB (See Section 4.2.1) in the post-processing pass, our actual last step. We have commented the theory behind this step in Section 3.5. This way we can simulate how our scenes would look like if they were being seen from different observers, like different cameras or the average human eye.

In order to allow users to swap observers at runtime for visualization purposes, we decided to load several `.csv` files containing the aforementioned response curves (See Section 3.5 for details on these curves), and let them decide which one to use from the application's user interface.

To make this approach work, we had to convert every response curve into a one-dimensional texture with 3 components that represent the response for red, green and blue at each wavelength (we also consider the case that the curves are in the XYZ colour space) that we can send into our spectral rendering pass. We also have to enable linear interpolation in these textures, otherwise we'd be introducing error in our results.

Once we have loaded all the data we need, the actual process looks like the following:

**Algorithm 2** Overview of our spectral rendering algorithm, including application of sensitivity curves, outputing a linear RGB image. Runs in the fragment shader of the deferred shading pass. See figure 4.4 for details on our rendering pipeline.

```
$lut\_coeffs \leftarrow fetch\_table(gbuffer\_stored\_albedo)$    ▷ Algorithm 9
for i = 0; i < $n_{wls}$; i++ do
    $\lambda \leftarrow fetch\_tex(wavelengths\_tex)$
    $upsampled\_spectral\_response \leftarrow Sigmoid(lut\_coeffs, \lambda)$    ▷ Equation 3.7
    $L_o \leftarrow material\_model\_eval(\lambda, upsampled\_spectral\_response)$
    $\{\bar{x}, \bar{y}, \bar{z}\} \leftarrow fetch\_tex(response\_curve, \lambda)$
    $L_o \leftarrow L_o \cdot \{\bar{x}, \bar{y}, \bar{z}\}$    ▷ Equation 2.6, part 1
end for
$colour_{XYZ} \leftarrow L_o \frac{\lambda_{max}-\lambda_{min}}{n_{wls}}$    ▷ Equation 2.6, part 2
$colour_{XYZ} \leftarrow normalize\_Y(colour_{XYZ})$    ▷ This way we get LDR colours
$colour_{RGB} \leftarrow XYZ\_to\_RGB(colour_{XYZ})$
return $colour_{RGB}$    ▷ Gamma correction to sRGB in postprocess pass
```

In summary, we perform our spectral rendering and right after computing each wavelength's spectral radiance, we multiply it by the curve's response at that same wavelength, taking advantage of our iterative implementation. After accumulating this responses for all the $n_{wls}$ wavelengths we're integrating over, we perform the final division for our rectangle rule integration. By applying the responses as we accumulate reflectances across the visible spectrum saves us from accumulating twice in two different loops.

# 5. Results and discussion

So far we have presented both the basis of our approach to real-time spectral rendering and our implementation for it, giving details and explaining some of the decisions we made during design and development.

Now, we validate our implemented pipeline. For this, we devised several test scenarios and that will help us in getting a grasp of the performance of our approach, as well as its strengths and weaknesses. Through this section we present several tests we made, to validate our pipeline's performance and results, by employing both quantitative and qualitative metrics. We will provide insights and discuss our findings through this chapter, as well as the methodology we followed and the concepts it's based on.

## 5.1 Quantitative validation

Our first step was performing a quantitative validation to confirm that our obtained synthesized images had the necessary quality to be used in real industry-level contexts. To perform such a quantitative validation, we needed to establish both a baseline to compare us to, as well as performance metrics that could give us an idea of how our real-time version behaves with respect to the baseline. We explain both of them before diving into our results and findings.

Mitsuba 3 [JSR$^+$22b] is a non-biased, Monte Carlo-based path tracer commonly used for scientific purposes, such as testing new methods and approaches, or for acting as a ground truth. That, added to the fact that it employs exactly the same spectral upsampling method as we decided to adapt to real-time rendering (Jakob and Hanika's approach [JH19]), it is the perfect baseline for our measurements. With it, we can obtain stochastic analytical solutions to the render equation (See Section 2.1.1 )that will act as our ground truth images.

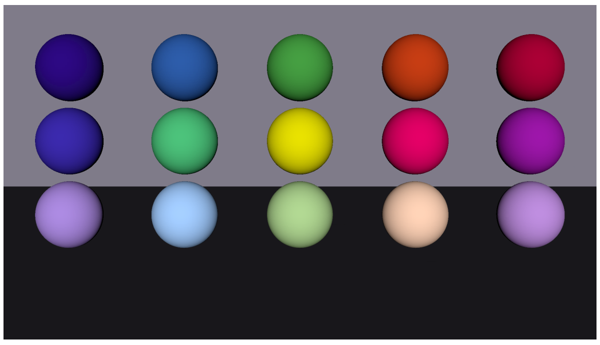
We are going to use Mitsuba's spectral variant results to compare against our real-time spectral implementation. To offer more perspective on the obtained measurements, we also implemented in our renderer the ability to perform RGB rendering. We will also compare those results against Mitsuba's baseline results.

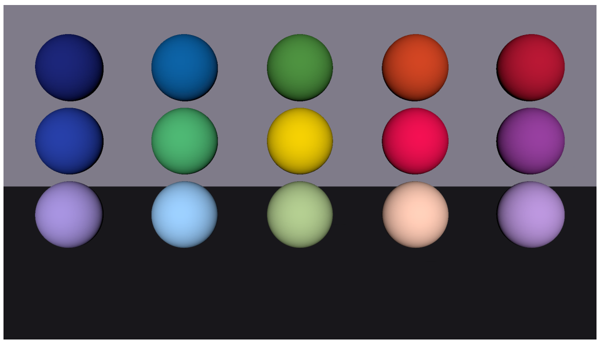
### 5.1.1 Naive upsampling strategy

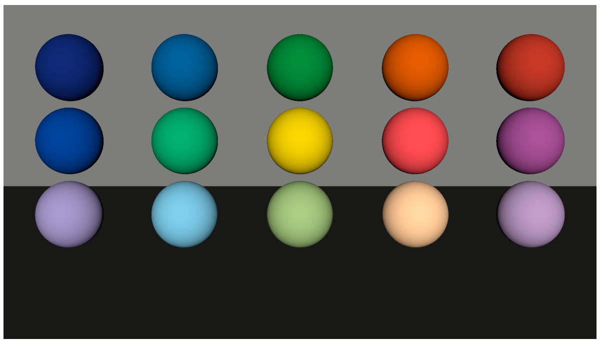
Additionally, to offer more ground for comparison with the results obtained by performing spectral upsampling, we also devised a straightforward but naive upsampling strategy, which we will compare against the other results. For a given reflectance $\mathbf{r} = \{\mathbf{r_{red}}, \mathbf{r_{green}}, \mathbf{r_{blue}}\}$ in RGB, the returned spectral reflectance $\hat{\mathbf{r}}(\lambda)$ for a given wavelength $\lambda$ is determined as in equation 5.1:

$$\hat{\mathbf{r}}(\lambda) = \begin{cases} r_{blue} & \text{if } \lambda \in (-\infty, \lambda_{th1}) \\ r_{green} & \text{if } \lambda \in [\lambda_{th1}, \lambda_{th2}] \\ r_{red} & \text{if } \lambda \in (\lambda_{th2}, \infty) \end{cases} \tag{5.1}$$

Where $\lambda_{th1}$ and $\lambda_{th2}$ are two thresholding values. Instead of performing some optimization to find the optimal values for them, like Jendersie did for volumetric coefficients [Jen21], we set them in an arbitrary manner to $\lambda_{th1} = 495nm$ and $\lambda_{th2} = 570nm$, values that map *approximately* to the threshold between our perception of blue and green, and green and red colours, respectively.


(a) Scene rendered using the naive upsampling technique taking 8 wavelength samples.


(b) Scene rendered using our regular upsampling method taking 8 wavelength samples.


(c) Scene rendered using the naive upsampling technique taking 140 wavelength samples.

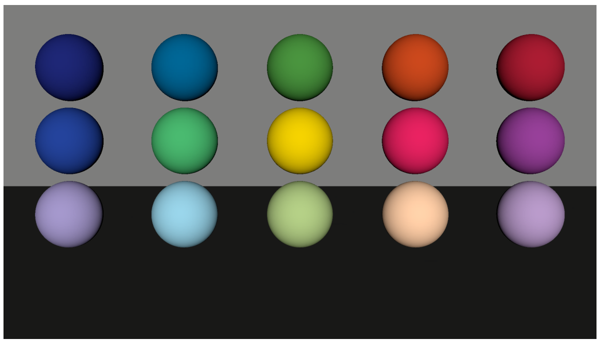

(d) Scene rendered using our regular upsampling technique taking 140 wavelength samples.

Figure 5.1: Comparison between spectral rendering with our naive upsampling and regular upsampling in our real-time renderer. Some discrepancies in coulours can be spotted if observed closely, especially by looking at the two topmost green spheres, the three blue spheres in the top left corner, or the four spheres in the top right corner. It can also be seen that, as we increase the number of wavelengths, energy isn't conserved with the naive approach, take for example the orange sphere in 5.1c.

The idea behind this is that this upsampling should perform decently in regular situations, when presented with smooth and uniform spectra, but when presented with peaked spectra its results should crumble. In Section 5.1.3 we delve deeper into this matter.

### 5.1.2 The CIE ΔE*2000 perceptual error metric

For this quantitative validation step we propose the use of several metrics, such as MAE (Mean Absolute Error), which are computed for each pair of pixels between the reference and the real-time techniques. However, these metrics are computed in the final colourspace, so a different more perceptual metric can help build a better understanding of the perceived difference by a human.

Hence, we introduce the usage of the CIE ΔE*2000 metric [SWD05]. It is a *perceptual* error metric built on top of its previous iterations over the years, indicating that its maturity, validated by the wide use it's given in colorimetry and colour sciences. This metric works on L*a*b* colour space and mainly aims to correct some inconsistencies on its predecessors (CIE ΔE*76 and CIE ΔE*94). Those inconsistencies include some brightness perceptual nonlinearities and adjustments for certain shades of blue by introducing new parameters and coefficients.

The CIE ΔE*2000 metric ranges from 0 to 100 depending on perceptual colour similarity, or how alike or different two given colours look to the human eye. More precisely, the scale is as follows:

- If the resulting value is less or equal than 1, the two compared colours are indistinguishable from one another. This is what it's called a just noticeable difference (JND).
- For values in between 1 and 2, the difference is barely perceptible through very close observation.
- For values ranging from 2 to 10, the colour difference is considered to be perceptible at a glance. The closer to 10, the faster it is to distinguish both colours from one another.
- For values from 10 to 49, the colours are more similar than opposite.
- Values greater than 50 and up to 100 indicate that those colours are almost opposites, or complete opposites.

### 5.1.3 Obtained results and findings

After explaining the basics of our testing environment and the error measurements we aim to compute, it is time to move on to the experiments themselves, along with discussion of the obtained results.

**Scene 1: Underwater coral reef** We compared our implementation against Mitsuba's path traced ground truth across different scenes, trying to simulate different relevant scenarios. We start our set of comparisons with an underwater oceanic scene, where we simulate different increasing depth values to check the effect of it on the reflectances of the scene models. A scene with high scattering dependency will help us understand the impact of this type of participating media over the reflectance values that lie within the volumes. In this particular case, the impact of the water scattering over the ocean floor's upsampled reflectances. We can see the results in Figure 5.2, along with two histograms for the average ΔE*2000 and MAE in Figure 5.3.

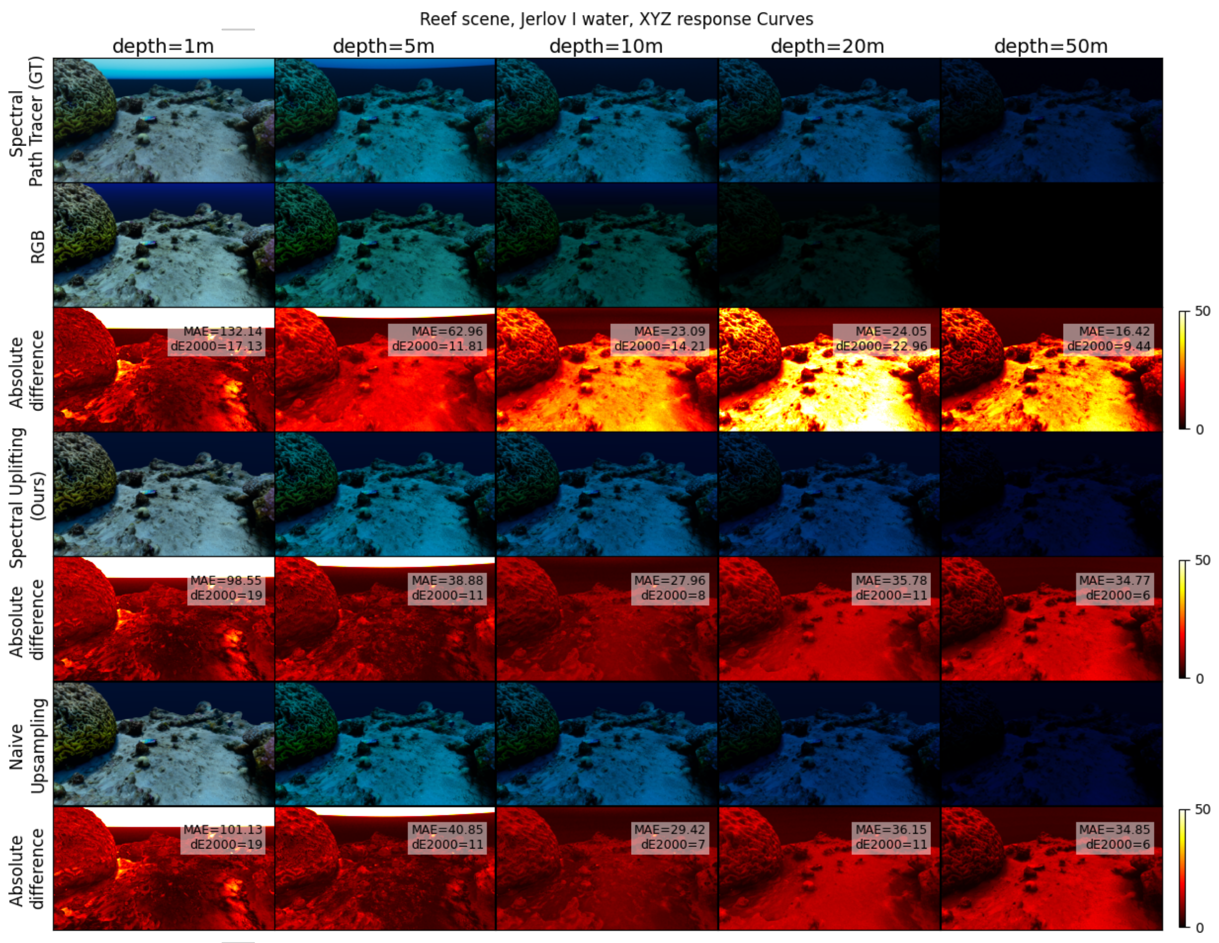
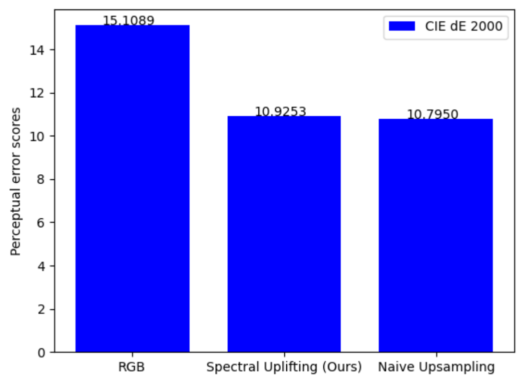


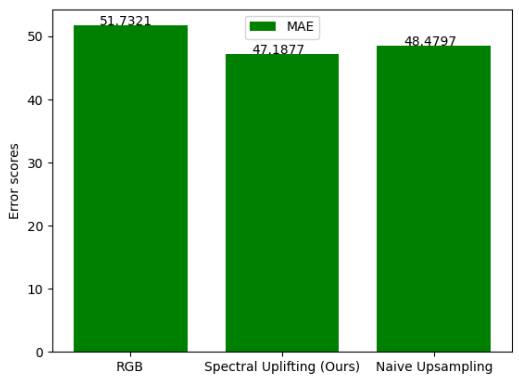
Figure 5.2: Our reef scene (CIE XYZ response curves simulating the human eye, D65 illuminant for the directional light, water is Jerlov type I, 200 wavelength samples used), compared against Mitsuba's path traced ground truth (top row) for varying simulated depths within the water body, increasing from left to right. We show our renderer's real-time computed RGB renders (second row), spectral renders obtained with our real-time spectral upsampling technique (fourth row), and finally spectral renders obtained with the naive upsampling technique (sixth row). Third, fifth and seventh rows show the image absolute difference values between the row on top of them and the corresponding ground truth in the first row, ranging from 0 to 255. We also offer the average MAE and CIE ΔE*2000 between image pairs.

As the third and fourth columns in 5.2 show, RGB rendering is unable to handle correctly the wavelength-dependent behavior of scattering and extinction that takes place within the water body. In the fifth column the effect is attenuated due to both images being darker as a consequence of depth and absorption, and not because the RGB rendering method regains accuracy.

Regarding spectral upsampling, our real-time approach beats the other two options in MAE, followed closely by the naive upsampling, and the opposite for the CIE $\Delta E * 2000$ metric, where there's almost a tie. Since our upsampled reflectances only apply to materials in the scene, we



(a) Average CIE $\Delta E * 2000$ for the image comparisons we performed in Figure 5.2.



(b) Average MAE values for the image comparisons we performed in Figure 5.2.

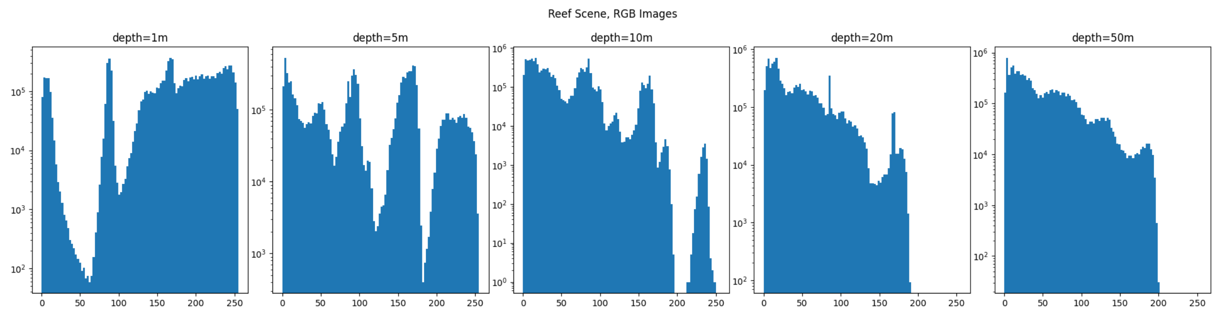
Figure 5.3: Averaged perceptual error (left) and MAE (right) for the images rendered with our pipeline, compared to the path traced ground truth. As a reminder, image differences range from 0 to 255.

have no control over the behaviour of the water body that partially occludes the space between the camera and the models to which we apply the materials. This means that the medium will hide the final colours in the scene, affecting the measurements. The more turbid the water, the more tinted with its colour the final scene will be, instead of those colours of the objects behind it, which in this case are the coral and the ocean floor. For example, a type 9C water according to the Jerlov classification would need much shallower depths to completely hide the oceanic floor in our scene.

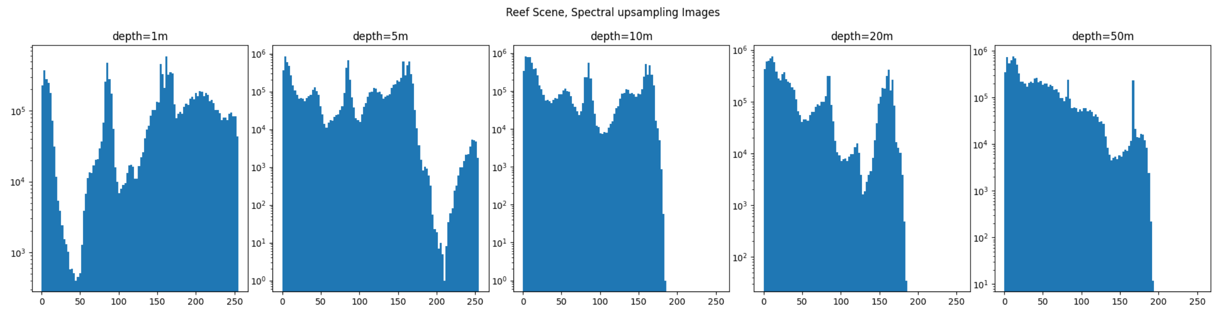
If we take a look into the histograms in Figure 5.4, we can see that for shallow depths the distribution of the errors looks similar among the 3 techniques. For 5, 10 and 20 meters deep, RGB distributions adopt a completely different shape, while the ones for both spectral rendering techniques present similar shapes. We also notice that, given a sufficiently high value of depth, such as 50 meters, error distributions begin to converge towards the same distributions. This is probably due to the scattering and extinction that take place in the water body, as we mentioned earlier.

**Scene 2: Lambertian spheres under the standard illuminant** Given that our first test scene was highly dominated by wavelength-dependent phenomena like scattering, RGB rendering was unable to correctly reproduce the colours in the medium, while both upsampling methods could match more faithfully the ground truth. For our second scene, we used a modified version of the spheres scene we've shown in previous figures (for example, Figure 5.1). Since our implementation doesn't support shadow mapping because it was not the focus of our work, we wanted to prevent the floor and back wall planes from showing any shadows in our ground truth render, knowing that path tracing can naturally account for occlusions and shadows.

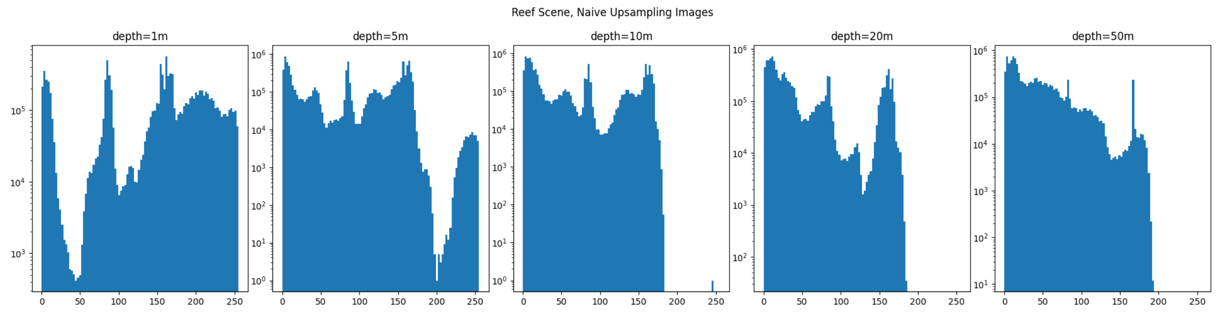
The target of this scene is to validate how far can our upsampling be from our ground truth under almost ideal conditions. With this in mind, we rendered the scene simulating an average human eye response (using the CIE XYZ response curves), under a standard D65 illuminant, which, as a reminder, acts as the white point for the RGB colour space. Figure 5.5 allows for



(a) Histogram for the Mean Absolute Error (MAE) values between our RGB renders and the baseline path traced render.



(b) Histogram for the Mean Absolute Error (MAE) values between our spectral renders that use our spectral upsampling and the baseline path traced render.



(c) Histogram for the Mean Absolute Error (MAE) values between our spectral renders that use the naive upsampling technique and the baseline path traced render.

Figure 5.4: Histograms showing the Mean Absolute Error distribution for the images we rendered for comparison in Figure 5.2. On the top row we show the RGB renders, while in the middle row we show the spectral renders that employ our spectral upsampling approach, and the bottom row presents those that leverage the naive upsampling approach. Depth increases from left to right.

visualizaiton of the image differences, while Figure 5.6 shows in a more data-oriented manner the average values for the errors. We have a look at the histogram distributions for mean absolute errors in Figure 5.7.

With the comparison in Figure 5.5 we see that in this case, our spectral rendering method handles the scene better than the other two approaches. This might be due to the fact that it is an ideal scenario with perfectly diffuse materials with only one RGB colour, and the illuminant for the scene is the white point for the RGB colour space. RGB rendering behaves in an erratic manner, since it returns the best values for CIE $\Delta E * 2000$ among the three images (even though it's pretty close), but its MAE metric is too high, indicating numerical imprecision in the method.

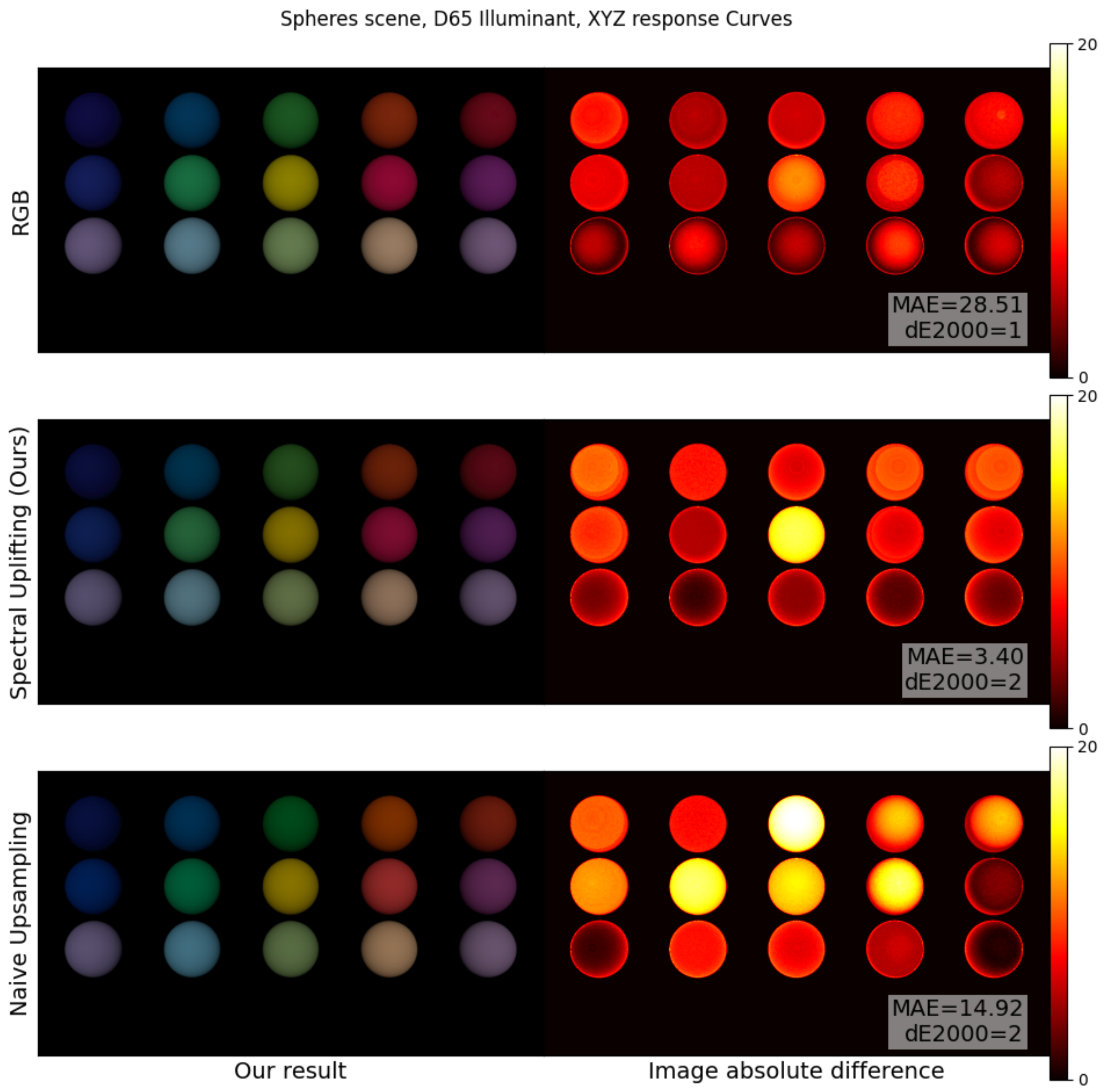
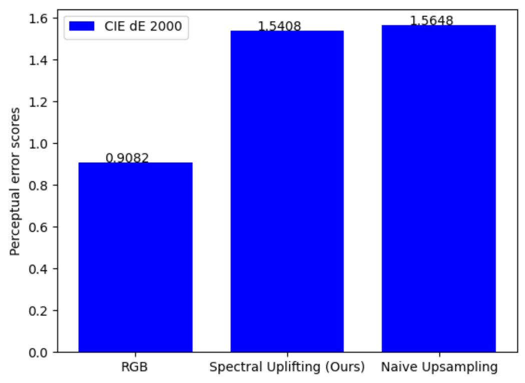
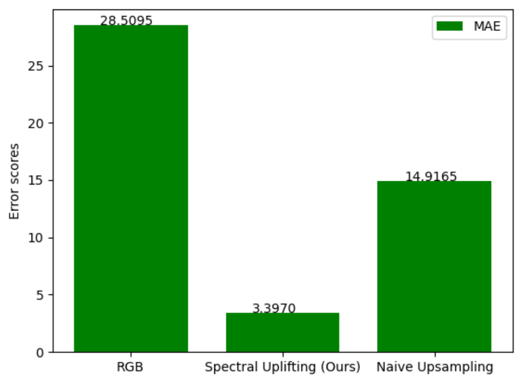

Spheres scene, D65 Illuminant, XYZ response Curves
RGB
MAE=28.51
dE2000=1
Spectral Uplifting (Ours)
MAE=3.40
dE2000=2
Naive Upsampling
MAE=14.92
dE2000=2
Our result
Image absolute difference
20
0

Figure 5.5: Our scene with 3 rows of 5 spheres each, rendered under the D65 illuminant and assuming a camera response equal to the average human eye (CIE XYZ response curves). We take 200 wavelength samples for our renders. Each row in the plot represents a comparison with our ground truth image, rendered with Mitsuba. In the left columns we can see the real-time renders we obtained for (from top to bottom) RGB rendering, our real-time spectral upsampling and the naive real-time upsampling method. On the right column, the image absolute differences between the images in the left column and Mitsuba's ground truth. We also show the per-image values for the MAE and CIE $\Delta E * 2000$ error metrics.

Looking at the other two upsampling strategies, we see that our approach behaves better in general, but the average values are brought down by a single colour, corresponding to the yellow sphere in the middle of the image. Regardless of this, our method can still outperform the naive upsampling approach both in CIE $\Delta E * 2000$ by a small margin, and in MAE by a highly representative one.

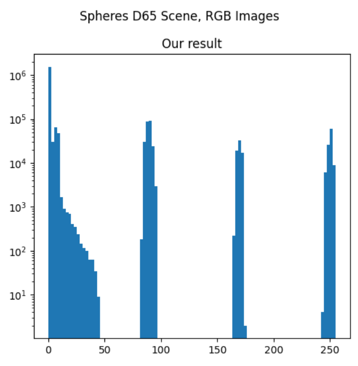


(a) CIE $\Delta E*2000$ values for the image comparisons we performed in Figure 5.5.

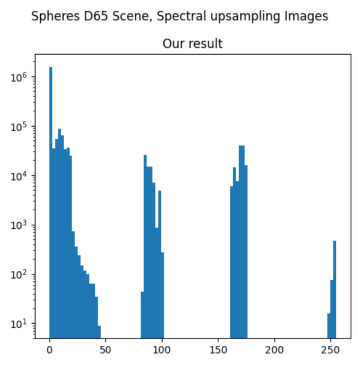


(b) MAE values for the image comparisons we performed in Figure 5.5.

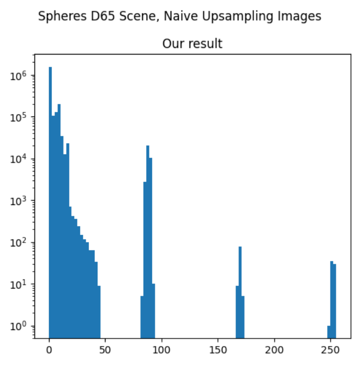
Figure 5.6: Perceptual error (left) and error (right) for the images rendered with our pipeline for Figure 5.5, compared to the path traced ground truth we obtained with Mitsuba. In this case, RGB beats the other two methods, mainly thanks to our choice of illuminant and the lack of wavelength-dependent phenomena in the scene.



(a) Histogram for the Mean Absolute Error (MAE) values between our RGB renders and the baseline path traced render we obtained with Mitsuba.



(b) Histogram for the Mean Absolute Error (MAE) values between our spectral renders that use our spectral upsampling and the baseline path traced render.



(c) Histogram for the Mean Absolute Error (MAE) values between our spectral renders that use the naive upsampling technique and the baseline path traced render.

Figure 5.7: Histograms showing the Mean Absolute Error distribution for the images we rendered for comparison in Figure 5.5. On the top row we show the MAE distribution corresponding to the RGB renders, while in the middle row we show it for the spectral renders that employ our spectral upsampling approach, and the bottom row presents it for those renders that leverage the naive upsampling approach, or RGB rendering.

Upon more detailed inspection of the mean absolute error distributions in the histograms in Figure 5.7, we see that MAE appears to be clustered around 4 groups in each of the approaches. This makes sense, since it's most likely correlated with the incidence angles of light in the surface of the spheres. We can also appreciate that in the RGB render there's more pixels that present either very low or very high values for MAE, suggesting a higher variance. Hence, it is advised to take the results with a grain of salt.

**Scene 3: Blue LED** As we commented, RGB outperforming both spectral upsampling techniques (even though it's by a small margin) on the second scene might be due to the fact that the D65 spectral power distribution (See Figure 5.8) presents a smooth (with no spikes), rather uniform appearance, which can also explain why the naive upsampling strategy hasn't behaved poorly on this experiment, since this approach thresholds the spectrum and a mostly uniform illuminant can help mitigating the negative effects of such thresholding.

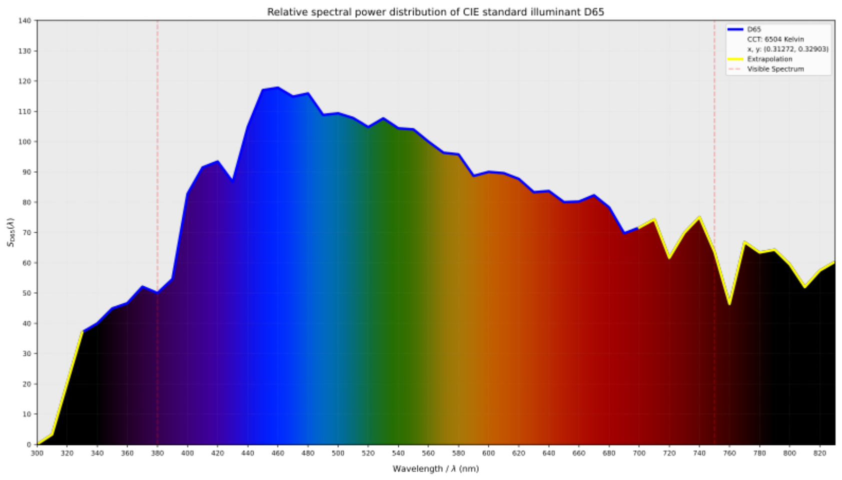


Figure 5.8: Spectral power distribution for the D65 illuminant. As it can be seen, it doesn't present big irregularities like spikes that could hurt the naive spectral upsampling approach.

Spiked spectra are a difficult case to handle within spectral rendering. Hence the design for this third scene, where we tried to go for any possible weaknesses in the rendering methods with an extreme setting. We modified our second scene so that the light emission pattern would correspond to a blue LED spectral power distribution, featuring a single thin spike. We chose the Globe-A19 LED for domestic use from LSPDD, a database for spectral distributions of different lamps and light emitters [RA24].

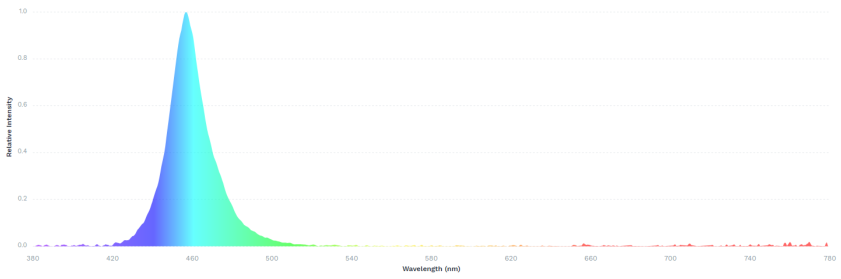
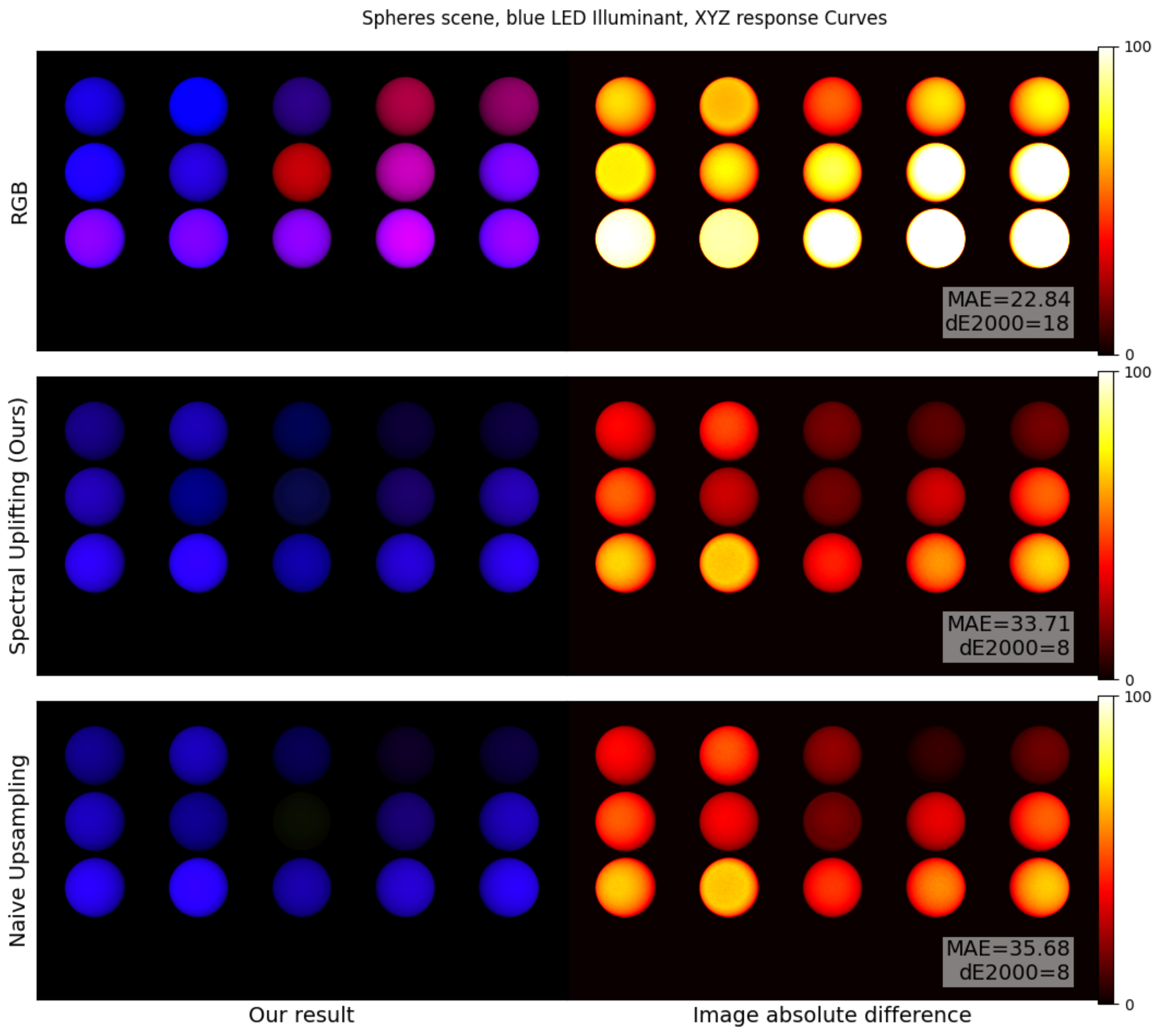


Figure 5.9: Spectral power distribution for our chosen blue light, corresponding to a Globe-A19 LED.

As we show in Figure 5.9, the LED spectral power distribution we chose features a prominent spike in the area surrounding the 450 nanometers mark, corresponding to blue wavelengths. We show the image comparison, as well as the relevant error metrics in Figures 5.10, 5.11 and 5.12.

As we foresaw, RGB lighting does a bad work when handling this type of illumination in comparison to the other two options. In this scenario it becomes evident why discretizing over the visible spectrum can be a bad idea, resulting in highly inaccurate results. It is also worth

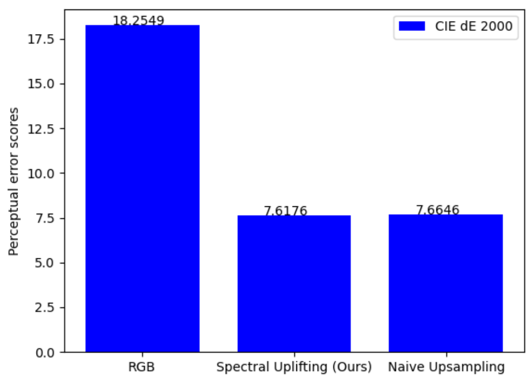


Spheres scene, blue LED Illuminant, XYZ response Curves
RGB
Spectral Uplifting (Ours)
Naive Upsampling
MAE=22.84
dE2000=18
MAE=33.71
dE2000=8
MAE=35.68
dE2000=8
100
0
Our result
Image absolute difference

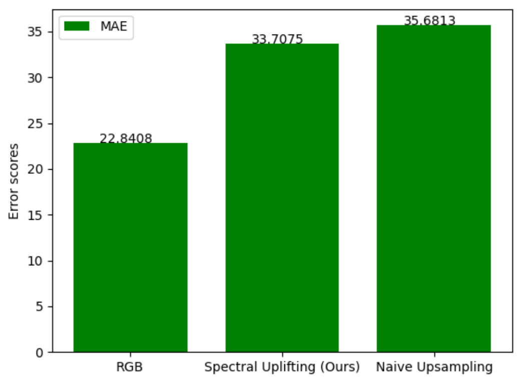
Figure 5.10: Our scene with 3 rows of 5 spheres each, rendered under a blue LED illuminant and assuming a camera response equal to the average human eye (CIE XYZ response curves). We take 200 wavelength samples for our renders. Each row in the plot represents a comparison with our ground truth image, rendered with Mitsuba. In the left columns we can see the real-time renders we obtained for (from top to bottom) RGB rendering, our real-time spectral upsampling and the naive real-time upsampling method. On the right column, the image absolute differences between the images in the left column and Mitsuba's ground truth. We also show the per-image values for the MAE and CIE $\Delta E * 2000$ error metrics.

noticing that, despite half of the colours being completely wrong, MAE still shows the best values among the three images. We believe this might be caused by an overflow in the RGB final values, and since MAE doesn't understand colour perception and only works with numerical distances, the obtained values are low even though the render is pretty much useless.

On the other hand, both of our spectral approaches behave appropiately, returning once again similar results. The fact that the naive upsampling approach holds relatively well and performs closely to our real-time upsampling is a bit of a surprise.



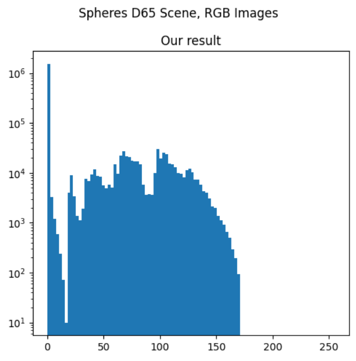
(a) CIE $\Delta E*2000$ values for the image comparisons we performed in Figure 5.10.



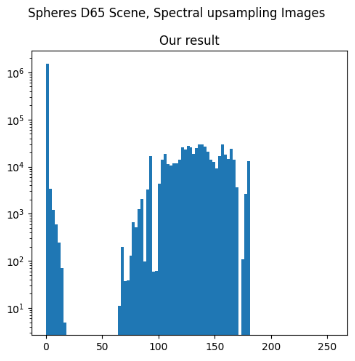
(b) MAE values for the image comparisons we performed in Figure 5.10.

Figure 5.11: Perceptual error (left) and error (right) for the images rendered with our pipeline for Figure 5.10 under a blue led illuminant, compared to the path traced ground truth we obtained with Mitsuba.

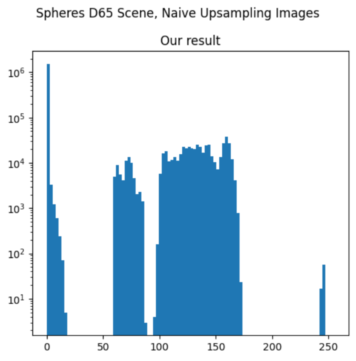
However, we can see that for certain combinations of illuminant and reflectances (mostly those that fall near or at both sides of the thresholds we commented in Section 5.1.1) this approach lacks precision. Upon closer examination, it can be seen that in Figure 5.10 the (originally) yellow sphere that is placed in the middle of the scene, along with the (originally) orange ones in the top right corner, almost disappear in the darkness. This behaviour is related to the arbitrary thresholding that is performed in this method.



(a) Histogram for the Mean Absolute Error (MAE) values between our RGB renders and the baseline path traced render we obtained with Mitsuba.



(b) Histogram for the Mean Absolute Error (MAE) values between our spectral renders that use our spectral upsampling and the baseline path traced render.



(c) Histogram for the Mean Absolute Error (MAE) values between our spectral renders that use the naive upsampling technique and the baseline path traced render.

Figure 5.12: Histograms showing the Mean Absolute Error distribution for the images we rendered for comparison in Figure 5.10. On the top row we show the MAE distribution corresponding to the RGB renders, while in the middle row we show it for the spectral renders that employ our spectral upsampling approach, and the bottom row presents it for those renders that leverage the naive upsampling approach, or RGB rendering.

Looking at the MAE distributions on the histograms we find that for RGB rendering, even though the overall error is huge, there are not many extreme values per pixel, and instead is a general accumulation of moderately high errrors. For both spectral approaches, we find that the MAE values for our upsampling mehods are slightly more uniform, with the naive approach falling apart in some specific pixels that report very high errors, most likely those pixels belong to the yellow sphere in the middle of the image, or the two rightmost ones in the top row.

With those 3 test scenes, we can extract some conclusions: First, RGB rendering is unreliable in most contexts. As a general rule, any dependence on wavelength can be considered a good enough reason to reject RGB rendering as a solution. Secondly, we expected the naive upsampling approach to perform worse than it did, but under most conditions its upsampled reflectances behaved decently enough. However, under heavy wavelength dependencies (like the LED case) where reflectancies and spectra power distributions seem to be complementary instead of similar, this method becomes unreliable too. (Recall the middle yellow sphere in the blue LED scene, in Figure 5.10.) Additionally, spectra obtained with this method will have a squared box-like appearance, while smooth shapes are appreciated for different purposes.

## 5.2 Qualitative validation

Lastly, we offer some insights on performance. Since one of the main points of our work is having an approach that works in real-time rates, it is vital to characterize and conduct a proper analysis on the rendering times, as well as the subsequent framerates. We conducted a study on average performance across a varying number of wavelength samples for our real-time spectral upsampling approach.

### 5.2.1 Testing environment

To put the results that will follow in the next section into perspective, we disclose the specifications of the machine we used to carry out the testing process:

- **CPU:** Intel Core i5-11400H @ 2.70GHz
- **Memory:** 16.0 GB DDR6 @ 3200MHz
- **GPU:** NVidia Geforcce RTX 3050 Laptop, 4.0 GB VRAM
- **Operating System:** Windows 10 Pro x64, Version 22H2

It is also noteworthy to say that, for this test, we set the compilation flags to Release mode.

### 5.2.2 Computational cost analysis

We measured the frame rendering times (and thus, the framerates, since both are the inverse of each other), taking into consideration several amounts of sampled wavelengths $n_{wls}$ each time. We took measurements for as low as $n_{wls} = 4$, and up to $n_{wls} = 1000$. As a reminder, we have mentioned earlier (Section 2.2) that we only allow for $n_{wls}$ to take values that are multiples of 4, in order to enable vectorization and hardware acceleration purposes, as well as encoding some spectral values (such as emission spectra) in four channel textures.

In practice, using such a high amount of wavelength samples isn't recommended since the benefits of increasing the amount of samples would soon get outnumbered by the computational costs, but we do so regardless of practical consideration, only for this theoretical characterization of render times in our system.

Theoretically, the time cost associated to performing spectral rendering to obtain a single frame in our renderer's implementation (Recall Algorithm 2) should be:

$$t_{frame}(n_{wls}) = t_{memory\_access} + t_{fetch\_LUTs} + n_{wls}(t_{sigmoid} + t_{lighting}) \tag{5.2}$$

Where $t_{memory\_access}$ can be seen as a constant that encompasses almost all the memory reads and writes during the process of rendering a frame, $t_{fetch\_LUTs}$ is the time we employ in fetching the look up tables that will return the coefficients we need for spectral upsampling, $t_{sigmoid}$ is the time employed in evaluating the reconstructed spectra via the sigmoid function we presented in equation 3.7, and finally, $t_{lighting}$ can be seen as the time employed in evaluating all necessary material models (and oceanic water models) for any lights in the scene.

Thus, we estimate a linear cost for our approach with an execution time of $\mathcal{O}(n_{wls})$. After taking the necessary measurements, we present the results in Figure 5.13.

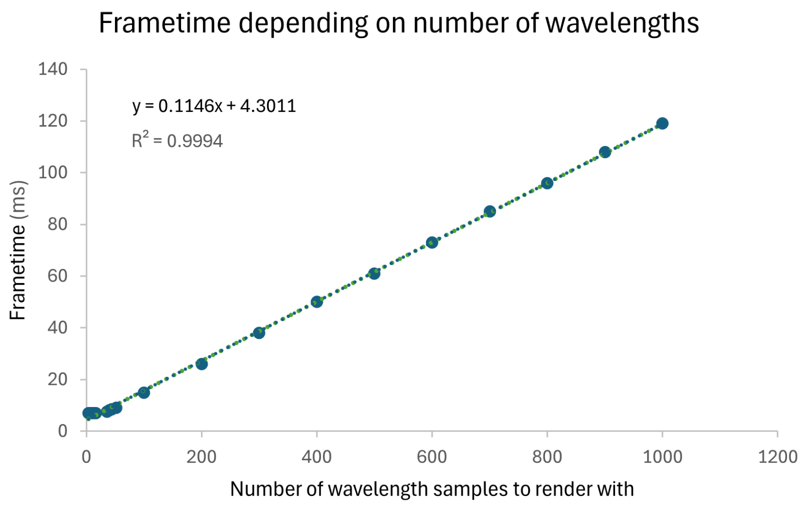


Figure 5.13: Plot showing variations in frametime depending on the number of wavelength samples used for spectral rendering. We also fitted the obtained data within a linear regression, and plot the obtained coefficients and correlation.

By just looking at the plot it can be seen that the relationship between the amount of wavelength samples and rendering time per frame is extremely linear, only presenting some noise for the first few values, where frametime was stable on 6.94 milliseconds (144 FPS, the maximum that our testing computer's display could handle). Thus, the frametime can be approximated by the following linear function:

$$t_{frame}(n_{wls}) = 0.11457 n_{wls} + 4.3010 \tag{5.3}$$

We have obtained highly interactive framerates (higher than 60 FPS) for reasonable amounts of samples $n_{wls}$, even when in the hundreds of wavelength samples our system stayed responsive at real-time framerates, which formally validates our approach to adapt this spectral upsampling technique into a real-time rendering context.

The time measurements and characterizations we have discussed here don't include a vectorized version of our shader code, since the only optimization we did for our development version of the program was to move the coefficient fetching process out of our main rendering loop, since it wasn't necessary and only introduced an enormous memory overhead, consequence of fetching the spectral upsampling coefficient look up tables, if left unoptimized. It could theoretically be possible to achieve a speed-up of nearly four times our actual rendering speed, but in practice it would be slightly less.

# 6. Final conclusions

Throughout this thesis, we have succesfully proposed a spectral real-time rendering pipeline in which we adapt a spectral upsampling technique for reflectances to make it work in real time, without producing a noticeable impact on performance. Being able to use this technique in real time helps tackling two of the main problems concerning spectral rendering:

- The cost in terms of time and effort of producing spectral textures, materials and assets by measuring spatially-varying reflectance data from the real world is exponentially higher than that of simply authoring those assets in RGB and sticking to that traditional type of rendering, which is one of the reasons behind the popularity of RGB rendering.
- Spectral assets can require copious amounts of memory with respect to their RGB counterparts depending on their resolution, occupying space both in disk and in memory during runtime, with the performance slowdowns that come associated to the memory overheads caused by moving such big amounts of data between memory, CPU and disk storage.

Our proposed technique now allows for using lightweight RGB assets in spectral rendering for real-time environments, like architectural visualizations, videogames or interactive learning applications, among others. All this comes at the single cost of one texture look-up per screen pixel, and six floating-point operations per wavelength sample.

We have created an implementation of our proposed pipeline in order to validate its correct functioning and exemplify how it could be implemented on an industry-level renderer. For that, we took into consideration the principles of Physically Based Rendering, and implemented one of the most common PBR material models, the Cook-Torrance model [CT82]. One issue with this material model is that it requires several coefficients, from which some encode a simplification of wavelength-dependent phenomena. It could be interesting to try and adapt this material model in a way that the Fresnel equations account aren't completely simplified, and account for the wavelength without excessively hurting performance [Hof19].

Another consequence of taking PBR into consideration for our pipeline is that we can simulate different spectral responses in our scene observers, effectively allowing to simulate several cameras with different spectral response curves. Final users can interactively control a wide array of parameters from a graphical interface, ranging from the integration range to the camera spectral response.

Our resulting implementation for this proposed pipeline is flexible, and leaves room for future expansions of the work we have done so far. In order to prove the extendability of our real-time approach with previously existing techniques for spectral rendering, we integrated a second spectral rendering method in our pipeline, an analytical approximation for real-time underwater rendering by Monzón et al. [MGAM24]. In their method, the authors offer a real-time performant approximation for performing spectral rendering of underwater scenes.

Among other assumptions, in their technique the authors assume the ocean floor to have a diffuse behaviour, yet they don't try to obtain spectral reflectance values for that diffuse material and limit themselves to performing spectral rendering on the water volume itself, ignoring the rest of elements. Thanks to our real-time spectral upsampling, techniques like this one can be combined to enable full spectral rendering of those types of scenes, enhancing the final results and avoiding RGB rendering, which in wavelength-dependent contexts can be highly inaccurate.

We have validated our real-time spectral rendering pipeline by comparing it against a path traced ground truth for several situations that modeled no wavelength dependency, and wavelength dependency in the iluminants of the scene. To put the obtained values into perspective, we compared our approach with traditional RGB rendering and a naive upsampling strategy. In most situations, our method performed the best and introduced the least amount of error with respect to the ground truth. In general, it showed the best behaviour whenever there was a wavelength dependency in the testing scenes. We have also conducted a study on rendering times and framerate where our implementation showed great performance, effectively validating the applicability of this approach for real-time environments.

This work can be considered the starting point for a new set of real-time works that focus on spectral rendering.

An interesting direction would be studying possible wavelength sampling strategies, in order to optimize the required number of them to achieve a good final image, which would ultimately increase performance. Such an optimization poses a challenge in itself, and a sensible starting point should be carefully considered. In this regard, screen-space multipass approaches could be explored [vdRE21].

Since our pipeline considers the possibility of simulating several spectral responses for the cameras in our scenes, that camera sensitivity information could be taken into account for sampling the spectral domain, establishing some importance sampling for areas where the camera response is higher.

As a closing thought, reflectance upsampling can be considered just the beginning of a series of works that are waiting to be put out there. There are already works that aim to upsample illuminances from the RGB colour space into the spectral domain [GGDG22], or volumetric coefficients [Jen21]. Eventually, we might reach the point where we can encode spectra for all types of materials, lights and participating media introducing minimal error, achieving full upsampling for RGB scenes into the spectral domain.

This work intends to be one small step towards that ambitious goal.

# Bibliography

[AK90] James Arvo and David B. Kirk. Particle transport and image synthesis. *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 1990.

[ass24] Assimp: The open asset importer library. `https://github.com/assimp/assimp`, 2024. Accessed: June 22nd 2024.

[BAC$^+$18] Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. The design and evolution of disney's hyperion renderer. *ACM Trans. Graph.*, 37(3), jul 2018.

[BB17] Laurent Belcour and Pascal Barla. A practical extension to microfacet theory for the modeling of varying iridescence. *ACM Trans. Graph.*, 36(4), jul 2017.

[BBG23] L. Belcour, P. Barla, and G. Guennebaud. One-to-many spectral upsampling of reflectances and transmittances. *Computer Graphics Forum*, 42(4), jul 2023.

[Bor91] Carlos F. Borges. Trichromatic approximation for computer graphics illumination models. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, page 101–104, New York, NY, USA, 1991. Association for Computing Machinery.

[Cha50] Subrahmanyan Chandrasekhar. *Radiative transfer.* 1950.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, jan 1982.

[Deb02] Paul Debevec. Image-based lighting. *Computer Graphics and Applications, IEEE*, 22:26–34, 04 2002.

[Dec96] Philippe Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June 1996.

[DF03] Mark Drew and Graham Finlayson. Multispectral processing without spectra. *Journal of the Optical Society of America. A, Optics, image science, and vision*, 20:1181–93, 08 2003.

[Dis02] Carnegie Mellon University Entertainment Technology Center Disney. Panda3d framework, 2002.

[Dur24] Danijel Durakovic. mini: .ini file reader and writer. `https://github.com/metayeti/mINI`, 2024. Accessed: June 22nd 2024.

[dV16] Joey de Vries. Learn opengl, 2016.

[DWS$^+$88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, page 21–30, New York, NY, USA, 1988. Association for Computing Machinery.

[DWWH20] Hong Deng, Beibei Wang, Rui Wang, and Nicolas Holzschuch. A practical path guiding method for participating media. *Computational Visual Media*, 6(1):37–51, 2020.

[Epia] Epic Games. Unreal engine 4.

[Epib] Epic Games. Unreal engine 5.

[Fou07] The Godot Foundation. Godot engine, 2007.

[GGDG22] Giuseppe Claudio Guarnera, Yuliya Gitlina, Valentin Deschaintre, and Abhijeet Ghosh. Spectral Upsampling Approaches for RGB Illumination. In Abhijeet Ghosh and Li-Yi Wei, editors, *Eurographics Symposium on Rendering.* The Eurographics Association, 2022.

[GGSC99] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 09 1999.

[gla24] Glad: Multi-language vulkan/gl/gles/egl/glx/wgl loader-generator based on the official specs. `https://github.com/Dav1dde/glad`, 2024. Accessed: June 22nd 2024.

[glf24] Glfw: A multi-platform library for opengl, opengl es, vulkan, window and input. `https://github.com/glfw/glfw`, 2024. Accessed: June 22nd 2024.

[glm24] Glm: Opengl mathematics library. `https://github.com/icaven/glm`, 2024. Accessed: June 22nd 2024.

[Gro92] Khronos Group. Opengl: The industry's foundation for high performance graphics, 1992.

[Gro16] Khronos Group. Vulkan home: Cross platform 3d graphics, 2016.

[HMB$^+$15] Stephen Hill, Stephen McAuley, Brent Burley, Danny Chan, Luca Fascione, Michał Iwanicki, Naty Hoffman, Wenzel Jakob, David Neubelt, Angelo Pesce, and Matt Pettineo. Physically based shading in theory and practice. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. Association for Computing Machinery.

[Hof19] Naty Hoffman. Fresnel Equations Considered Harmful. In Reinhard Klein and Holly Rushmeier, editors, *Workshop on Material Appearance Modeling.* The Eurographics Association, 2019.

[HZE+19] Sebastian Herholz, Yangyang Zhao, Oskar Elek, Derek Nowrouzezahrai, Hendrik P A Lensch, and Jaroslav Křivánek. Volume path guiding based on zero-variance random walk theory. *ACM Trans. Graph.*, 38(3):1–19, 6 2019.

[JA18] Adrian Jarabo and Victor Arellano. Bidirectional rendering of vector light transport. In *Computer Graphics Forum*, volume 37, pages 96–105. Wiley Online Library, 2018.

[Jen21] Johannes Jendersie. *Fast Spectral Upsampling of Volume Attenuation Coefficients*, pages 153–159. 08 2021.

[JER51] NG JERLOV. Optical studies of ocean waters. reports of the swedish deep-sea expedition, 3. *NO. I*, 1951.

[Jer76] Nils Gunnar Jerlov. *Marine optics.* Elsevier, 1976.

[Jer77] N. G. Jerlov. Classification of sea water in terms of quanta irradiance. *ICES Journal of Marine Science*, 37(3):281–287, 09 1977.

[JESG12] Jorge Jimenez, Jose I. Echevarria, Tiago Sousa, and Diego Gutierrez. Smaa: Enhanced morphological antialiasing. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, 31(2), 2012.

[JF60] NG Jerlov and M Fukuda. Radiance distribution in the upper layers of the sea. *Tellus*, 12(3):348–355, 1960.

[JH19] Wenzel Jakob and Johannes Hanika. A low-dimensional function space for efficient spectral upsampling. *Computer Graphics Forum (Proceedings of Eurographics)*, 38(2), March 2019.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 511–518, New York, NY, USA, 2001. Association for Computing Machinery.

[JSR+22a] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. Mitsuba 3 documentation. spectra: srgb spectrum. `https://mitsuba.readthedocs.io/en/latest/src/generated/plugins_spectra.html#srgb-spectrum-srgb`, 2022. Accessed: June 21st 2024.

[JSR+22b] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. Mitsuba 3 renderer, 2022. https://mitsuba-renderer.org.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[Kar13] Brian Karis. Real shading in unreal engine 4 by. 2013.

[Kar24] Baldur Karlsson. Renderdoc: stand-alone graphics debugger. `https://renderdoc.org/`, 2024. Accessed: June 22nd 2024.

[KCK+22] Simon Kallweit, Petrik Clarberg, Craig Kolb, Tom'aš Davidovič, Kai-Hwa Yao, Theresa Foley, Yong He, Lifan Wu, Lucy Chen, Tomas Akenine-Möller, Chris Wyman, Cyril Crassin, and Nir Benty. The Falcor rendering framework, 8 2022. `https://github.com/NVIDIAGameWorks/Falcor`.

[LW96] Eric P. Lafortune and Yves D. Willems. Rendering participating media with bidirectional path tracing. In *Rendering Techniques '96*, pages 91–100. Springer, 6 1996.

[Mac35] David L. Macadam. Maximum visual efficiency of colored materials. *Journal of the Optical Society of America*, 25:361–367, 1935.

[MGAM24] Nestor Monzon, Diego Gutierrez, Derya Akkaynak, and Adolfo Muñoz. Real-time underwater spectral rendering. *Computer Graphics Forum*, page e15009, 2024.

[MHD16] Johannes Meng, Johannes Hanika, and Carsten Dachsbacher. Improving the dwivedi sampling scheme. *Comput. Graph. Forum*, 35(4):37–44, 7 2016.

[MHH+12] Stephen McAuley, Stephen Hill, Naty Hoffman, Yoshiharu Gotanda, Brian Smits, Brent Burley, and Adam Martinez. Practical physically-based shading in film and game production. In *ACM SIGGRAPH 2012 Courses*, SIGGRAPH '12, New York, NY, USA, 2012. Association for Computing Machinery.

[Mic95] Microsoft. Microsoft download center: Directx, 1995.

[Mob94] Curtis Mobley. *Light and Water: Radiative Transfer in Natural Waters.* 01 1994.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 1965.

[MSHD15] Johannes Meng, Florian Simon, Johannes Hanika, and Carsten Dachsbacher. Physically Meaningful Rendering using Tristimulus Colours. *Computer Graphics Forum*, 2015.

[MY19] Ian Mallett and Cem Yuksel. Spectral Primary Decomposition for Rendering with sRGB Reflectance. In Tamy Boubekeur and Pradeep Sen, editors, *Eurographics Symposium on Rendering - DL-only and Industry Track.* The Eurographics Association, 2019.

[OYH18] Hisanari Otsu, Masafumi Yamamoto, and Toshiya Hachisuka. Reproducing spectral reflectances from tristimulus colours. *Computer Graphics Forum*, 37, 2018.

[Pin88] Juan Pineda. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20, jun 1988.

[PKK00] Mark Pauly, Thomas Kollig, and Alexander Keller. Metropolis light transport for participating media. In *Eurographics*, pages 11–22. Springer Vienna, Vienna, 2000.

[PM93] S N Pattanaik and S P Mudur. Computation of global illumination in a participating medium by monte carlo simulation. *J. Vis. Comput. Animat.*, 4(3):133–152, 7 1993.

[RA24] Johanne Roby and Martin Aubé. Lspdd: Lamp spectral power distribution database, 2024.

[RF24] Ton Roosendaal and Blender Foundation. Blender 4.1. `https://www.blender.org/`, 2024. Accessed: June 21st 2024.

[RSSF02] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. Graph.*, 21(3):267–276, jul 2002.

[SA23] Grigory Solomatov and Derya Akkaynak. Spectral sensitivity estimation without a camera. In *IEEE International Conference on Computational Photography (ICCP)*, July 2023.

[Sch94] Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[SG31] T Smith and J Guild. The c.i.e. colorimetric standards and their use. *Transactions of the Optical Society*, 33(3):73, jan 1931.

[Smi67] B. Smith. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, 1967.

[Smi99] Brian Smits. An rgb-to-spectrum conversion for reflectances. *Journal of Graphics Tools*, 4(4):11–22, 1999.

[stb24] Stb: single-file public domain libraries for c/c++. `https://github.com/nothings/stb`, 2024. Accessed: June 22nd 2024.

[Str79] "Bjarne Stroustrup". "the c++ programming language", 1979.

[SWD05] Gaurav Sharma, Wencheng Wu, and Edul N. Dalal. The ciede2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. *Color Research & Application*, 30(1):21–30, 2005.

[Tec05a] Unity Technologies. Unity scriptable rendering pipeline documentation, 2005.

[Tec05b] Unity Software Technologies. Unity 3d engine, 2005.

[TG17a] Antoine Toisoul and Abhijeet Ghosh. Practical acquisition and rendering of diffraction effects in surface reflectance. *ACM Trans. Graph.*, 36(5), jul 2017.

[TG17b] Antoine Toisoul and Abhijeet Ghosh. Real-time rendering of realistic surface diffraction with low rank factorisation. In *Proceedings of the 14th European Conference on Visual Media Production (CVMP 2017)*, CVMP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[tin24] Tinydir: Lightweight, portable and easy to integrate c directory and file reader. `https://github.com/cxong/tinydir`, 2024. Accessed: June 22nd 2024.

[TR75] T. S. Trowbridge and K. P. Reitz. Average irregularity representation of a rough surface for ray reflection. *J. Opt. Soc. Am.*, 65(5):531–536, May 1975.

[TWF21] Lucia Tódová, Alexander Wilkie, and Luca Fascione. Moment-based Constrained Spectral Uplifting. In Adrien Bousseau and Morgan McGuire, editors, *Eurographics Symposium on Rendering - DL-only Track*. The Eurographics Association, 2021.

[TWF22] L. Tódová, A. Wilkie, and L. Fascione. Wide gamut moment-based constrained spectral uplifting. *Computer Graphics Forum*, 41(6):258–272, 2022.

[vdRE21] M. van de Ruit and E. Eisemann. A multi-pass method for accelerated spectral sampling. *Computer Graphics Forum*, 40(7):141–148, 2021.

[vdRE23] Mark van de Ruit and Elmar Eisemann. Metameric: Spectral uplifting via controllable color constraints. In *ACM SIGGRAPH 2023 Conference Proceedings*, SIGGRAPH '23, New York, NY, USA, 2023. Association for Computing Machinery.

[VG97] Eric Veach and Leonidas J Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*, New York, New York, USA, 1997. ACM Press.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, jun 1980.

[WND$^+$14] A. Wilkie, S. Nawaz, M. Droske, A. Weidlich, and J. Hanika. Hero wavelength spectral sampling. In *Proceedings of the 25th Eurographics Symposium on Rendering*, EGSR '14, page 123–131, Goslar, DEU, 2014. Eurographics Association.