Efficient Spectral Rendering on the GPU for Predictive Rendering

David Murray, Alban Fichet, Romain Pacanowski

CHAPTER 42

EFFICIENT SPECTRAL RENDERING ON THE GPU FOR PREDICTIVE RENDERING

David Murray,[1] Alban Fichet,[1] and Romain Pacanowski[1,2]
[1]Institut d'Optique Graduate School, CNRS
[2]INRIA

ABSTRACT

Current graphic processing units (GPU) in conjunction with specialized APIs open the possibility of interactive path tracing. Spectral rendering is necessary for accurate and predictive light transport simulation, especially to render specific phenomena such as light dispersion. However, it requires larger assets than traditional RGB rendering pipelines. Thanks to the increase of available onboard memory on newer graphic cards, it becomes possible to load larger assets onto the GPU, making spectral rendering feasible. In this chapter, we describe the strengths of spectral rendering and present our approach for implementing a spectral path tracer on the GPU. We also propose solutions to limit the impact on memory when handling finely sampled spectra or large scenes.

42.1 MOTIVATION

Nowadays, computer-generated images are common due to the video game and movie industries' continuous growth. Although the images produced by the entertainment industry look more and more realistic, the process used to render them, named here *tristimulus rendering*,[1] is unsuitable for some domains. For example, the architecture and automotive domains require *predictive rendering* to assess the visual quality of their products before they are put on the production line.

Predictive rendering requires a spectral simulation of light transport. This permits the creation of complex wavelength-dependent effects, which can

[1]A tristimulus renderer computes light transport using only three color channels. The color space (e.g., RGB, HSV, etc.) defining the colors is arbitrarily chosen.

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_42

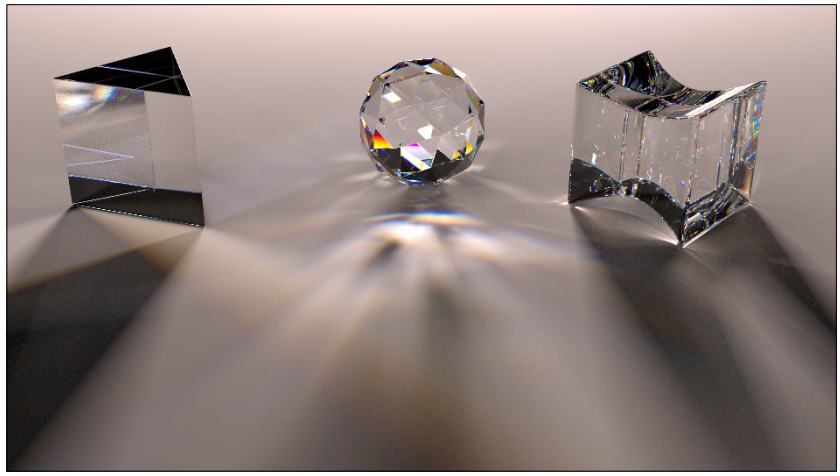


Figure 42-1. *Only a spectral renderer can simulate accurately the richness of color from wavelength-dependent effects, such as light dispersion.*

produce striking colors, such as light dispersion (see Figure 42-1), light polarization, iridescence, opalescence, and even fluorescence effects.

Compared to tristimulus rendering, the usual drawbacks for spectral rendering are its slowness and the memory footprint needed by scene assets. Indeed, assets must be described for each wavelength, which drastically increases the total memory footprint of a scene.

Central processing unit (CPU)–based spectral rendering engines are present in the academic sector (e.g., ART [25], PBRT v3 [23], and Mitsuba [11]) and also in the industry (e.g., Manuka [8]) but are not the norm. On the other hand, most renderers exploiting graphical processing unit (GPU) capabilities are RGB-based (e.g., Iray [14] and Cycles [3]). However, an increasing number of academic spectral renderers now exploit the GPU as well (e.g., Mitsuba 2 [18], PBRT v4 [22], and Malia [5]).

Historically, GPU compute capabilities have grown faster than those of CPUs but are more limited in terms of available memory. However, the recent release of specialized ray tracing hardware for the GPU (e.g., RTCore) and specific APIs (e.g., OptiX [20], DirectX 12 Raytracing, and Vulkan Ray Tracing [24]) provides an opportunity to achieve spectral rendering at interactive frame rates. This opens the possibility to build new pre-visualization tools (e.g., for designers) with spectral rendering capabilities.

In this chapter, we present, through our open source solution Malia [5], how to implement efficient spectral rendering on the GPU. Malia can be used to generate *spectral* images in offline mode (but still using the power of the GPU) or in pre-visualization mode by using progressive rendering and an OpenGL framebuffer to display interactively the current spectral image. These features permit us to showcase step-by-step guidelines for spectral rendering on the GPU while providing appropriate figures to illustrate both its usefulness and its performance.

In Section 42.2, we briefly summarize the theoretical limitations of tristimulus rendering and illustrate why it cannot be used to generate predictive and accurate images. In Sections 42.3 and 42.4, we explain and evaluate various technical solutions implemented in Malia for spectral rendering on the GPU. Then in Section 42.5, we present some results to illustrate what performance can be achieved with these solutions and how to scale the approach with larger assets as well as increased spatial and spectral resolutions of the simulated image sensor. Section 42.6 concludes this chapter with a discussion and outlines potential future work that could improve even further either the accuracy or the efficiency of the proposed solution.

42.2 INTRODUCTION TO SPECTRAL RENDERING

In a tristimulus renderer, all reflectance and illuminance color values are tristimulus. If these values derive from spectral data, then integrating spectra values into tristimulus values is performed prior to the rendering process, comprising its main limitation (see Section 42.2.1). In practice, these values are directly provided in XYZ or RGB color space.

A spectral renderer operates on a per-wavelength basis (see Section 42.2.2), covering the full visible spectrum. Reflectance and illuminance values are directly used as spectral data, thus requiring no transformation. If needed, the final spectrum can be converted to a specific color space using any relevant sensor model (see Section 42.2.3).

42.2.1 LIMITATION OF TRISTIMULUS RENDERING

The integration process, prior to light transport, is the core limitation of a tristimulus renderer. Equation 42.1 illustrates this approximation[2]:

[2]To simplify the notation, we represent only a single bounce. The reflectance spectra or tristimulus values have to be multiplied by each other for each further bounce.

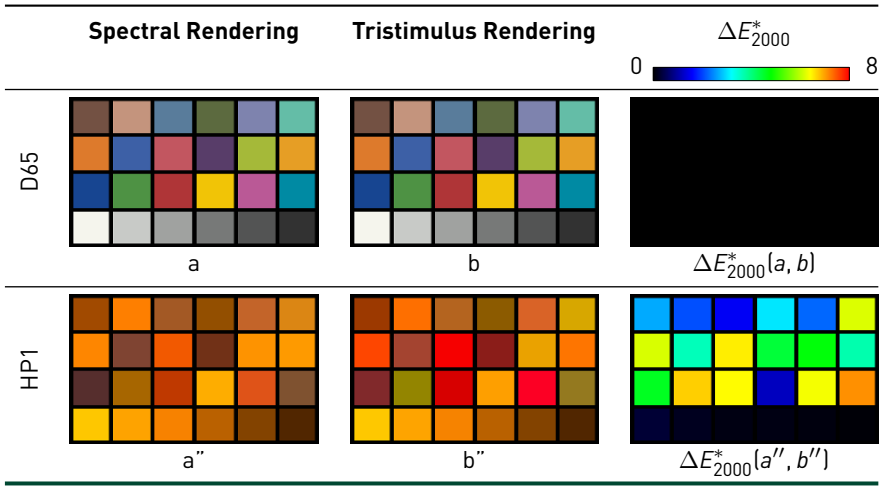


Figure 42-2. *Direct illumination comparisons between spectral and tristimulus renderings for two classical illuminants: D65 (top) and HP1 (bottom). When using spectral rendering, we transform the colors from spectra to RGB at the end. With tristimulus rendering, we transform both reflectance and illuminance values into their tristimulus counterparts prior to the evaluation. Then, we compute the final color by multiplying the illuminance and reflectance colors together. This approximation leads to observable differences as shown by the ΔE^*_{2000} metric.*

$$\underbrace{\int_\Lambda r(\lambda)L_e(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda}_{\text{spectral rendering}} \approx \underbrace{\overbrace{\left(\frac{1}{N_m}\int_\Lambda r(\lambda)L_e^m(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda\right)}^{\text{tristimulus reflectance}} \cdot \overbrace{\left(N_m \frac{\int_\Lambda L_e(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda}{\int_\Lambda L_e^m(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda}\right)}^{\text{tristimulus illuminance}}}_{\text{tristimulus rendering}}, \tag{42.1}$$

where $N_m = \int_\Lambda L_e^m(\lambda)\bar{y}(\lambda)\mathrm{d}\lambda$ is the illuminance normalization factor, $r(\lambda)$ is the spectral reflectance value, $L_e(\lambda)$ is the spectral power distribution (SPD) of the rendering illuminant, $L_e^m(\lambda)$ is the SPD of the illuminant used for measuring reflectance, and $\bar{c}(\lambda)$ is one of the color matching functions $(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda))$.

The presence of the normalization factor N_m in Equation 42.1 implies that the tristimulus reflectance value will only be correct if $L_e = L_e^m$, that is, only if the exact same illuminant is used for both measuring and rendering. Otherwise, discrepancies occur, as illustrated by Figure 42-2. This is very likely to happen if several different illuminants are used in a scene.

Global illumination increases even more the discrepancies (see [7]) presented earlier. As the tristimulus values are only approximated, sharp power distribution (e.g., the "HP1" illuminant SPD shown in Figure 42-3a) will be

(a) *Illuminant* **(b)** *Tristimulus rendering* **(c)** *Spectral rendering*

Figure 42-3. *Discrepancies induced by a tristimulus renderer are even more prevalent in a global illumination context. This leads to important color and intensity differences, especially with narrow spectra like the one used to light this Cornell box scene (HP1, high-pressure vapor lamp).*

either under- or overestimated. At each new reflection bounce event, a tristimulus renderer thus accumulates error by breaking energy conservation, yielding incorrect results (see Figure 42-3).

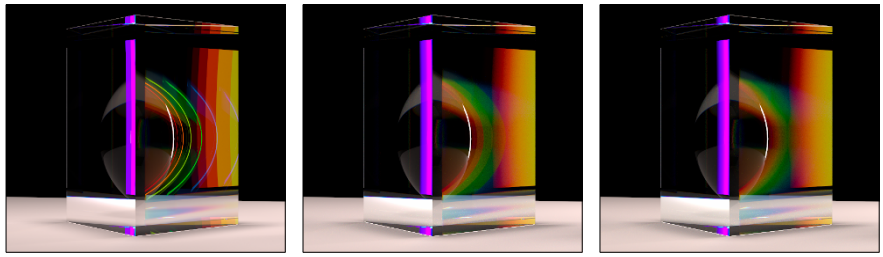
42.2.2 BASIS OF SPECTRAL RENDERING

To implement a spectral renderer, fixed wavelength sampling is the simplest scheme to use. A set of discrete wavelengths is determined at the beginning of the rendering process, and the path tracer will exclusively and exactly use these wavelengths. It offers simple asset management. However, this sampling scheme produces, most of the time, severe visual color artifacts (e.g., spectral aliasing). In particular when dealing with wavelength-dependent paths (i.e., refracted rays), only a discrete set of the possible paths is explored (see Figure 42-4a).

To avoid spectral aliasing, another dimension needs to be taken into account in the Monte Carlo integration process: the spectral domain (see Evans and McCool [6]). In this chapter, the spectral domain is fully covered by jittering the processed wavelength within an interval of two fixed wavelengths (see Figure 42-4b). We will refer to this interval as a *spectral bin*. To further reduce aliasing, the resulting value can be distributed to the adjacent spectral bins with any desired kernel for filtering (i.e., with a linear interpolation, see Figure 42-4c).

42.2.3 OUTPUT OF A SPECTRAL RENDERER

A spectral renderer can output a spectral image, a color image, or both. The choice mainly depends on the application. *Color images* require that the



(a) *Rendering with a discrete set of wavelengths*

(b) *Sampling wavelength at each new sample; accumulation in nearest bins*

(c) *Sampling wavelength at each new sample; accumulation distributed in neighboring bins*

Figure 42-4. *This scene shows a prism made of a dispersive glass: it has a wavelength-dependent index of refraction (later detailed in Section 42.4). (a) A discrete set of wavelengths are used. This produces significant spectral banding artifacts because only a subset of paths is explored. (b) We jitter the wavelength within each bin for each sample and then accumulate the resulting radiance in the nearest bins. There is still some banding because the transitions between sampled bins are visible in the sphere reflection. (c) We use a different kernel to accumulate results in neighboring bins, thus further decreasing the hard transitions between bins.*

sample accumulation is performed in a specific color space (XYZ, RGB, CMYK, etc.). Samples are thus converted prior to the accumulation. The main benefit of generating directly a color image is that the result will be the most accurate possible for this specific color space. However, the main downside of this format is that it is hardly transposable to another color space.

On the other hand, a *spectral image* remains "color space agnostic." As it contains the spectral power distribution, it can be converted to any color space (e.g., a display device or a printer). The downside of a spectral image is the additional storage cost as it requires one image layer per spectral bin. Note that an interactive renderer may generate, internally, a spectral image while displaying a color image by performing on-the-fly conversion. An important advantage of spectral images is that they store physical units, which is especially desirable for predictive rendering. For this reason, they also can be reused as the input to a spectral renderer.

One must be aware that there are two types of spectral images: *emissive* (storing energy $\in [0, \infty)$) and *reflective* (storing reflectance or attenuation values $\in [0, 1]$) images. When converting a spectral image to a color space, the process is not exactly the same depending on its type, as detailed in Figure 42-5.

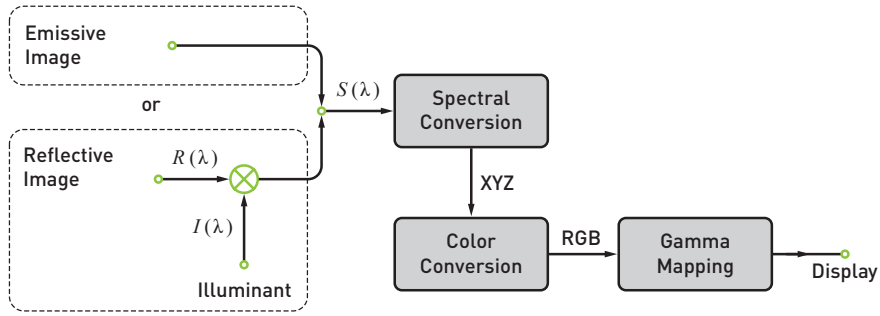


Figure 42-5. *Typical conversion pipeline for spectral images. Reflective spectral images have to be multiplied by an illuminant SPD prior to the spectral to tristimulus conversion. This ensures getting a preview of the reflectance given a specific illumination condition.*

In our renderer, we use spectral outputs: we want to retain the full radiometric quantities while having the ability to simulate the impact on different sensors capturing the same scene after the rendering process (e.g., a sensor from Ximea [4]).

In summary, spectral rendering is necessary when the following occur:

- Targeting color-accurate results regardless of the illuminant used in the scene.
- Using the resulting image on various and unknown display devices (i.e., need for "color space agnostic" images).
- Rendering specific visual effects implying wavelength considerations (e.g., dispersion, polarization, etc.).

42.3 SPECTRAL RENDERING ON THE GPU

Limited memory is the main challenge when using GPUs for spectral rendering: spectral assets are larger than their tristimulus counterparts. Often, all assets cannot fit at their full spectral resolution on video RAM (VRAM), and we must resort to frequent host-device asset transfers. A balance between VRAM usage, host-device transfers, and computation time is crucial for efficient rendering, all the more for predictive interactive renderers, as discussed later on in Section 42.5. Our implementation choices are guided by these current hardware limitations.

With these limitations in mind, we use a single wavelength approach (one wavelength per ray) to introduce the different strategies that can be

considered when implementing a spectral renderer. Then, using one of the described strategies, we show how *multiplexing* (multiple wavelengths per ray) increases the efficiency of a spectral renderer on a GPU. In particular, we present how we handle efficient wavelength sampling in this context, as this can be a pitfall when converting a tristimulus renderer to a spectral one.

42.3.1 SPECTRAL SAMPLING ON GPU FOR SINGLE WAVELENGTH RENDERING

The main difficulty when going from a CPU-based to a GPU-based spectral path tracer is sampling the spectral domain correctly. This process can no longer be efficiently performed on a per-ray basis due to the necessary dispatch on the GPU. It requires all assets at their full spectral resolution to be resident in VRAM if the scene can fit; otherwise, for each new sampled wavelength, all assets must be updated on GPU memory.

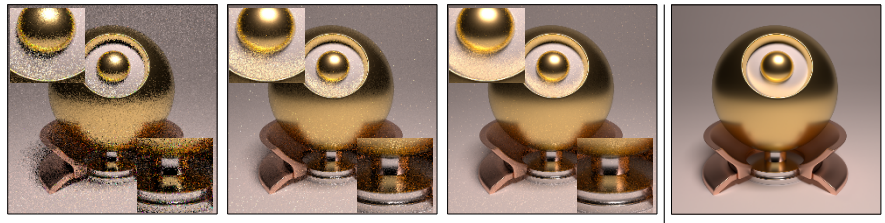
There are three main solutions for implementing single wavelength rendering on the GPU:

1. *Per-wavelength asset upload:* A random wavelength is sampled on the CPU, and all assets are uploaded to the GPU memory for this specific wavelength. This solution is memory-friendly but induces costly CPU-GPU exchanges (one per processed wavelength). Although being inefficient due to the important CPU-GPU transfers, this method provides the reference solution because it does not introduce any bias.

2. *Fixed wavelength asset upload:* A fixed set of spectral bins are processed sequentially. This requires only a subset of the asset spectral data to be uploaded in VRAM. Each bin is centered around one of the user-requested wavelengths. Only this central wavelength is used (fixed sampling). This solution may cause spectral aliasing, as illustrated in Section 42.2.2 and Figure 42-4a.

3. *Wavelength boundary asset upload:* A set of spectral bins is processed sequentially. Assets are uploaded on VRAM for the boundaries of each bin (see Figure 42-7). Sampling a new wavelength is done on the GPU by sampling an offset $\xi \in [0, 1]$. This offset is used to jitter the currently processed wavelength within the bin's boundaries. The assets' values are then *linearly* interpolated on the GPU using this offset.

   This method allows continuous wavelength sampling while limiting the number of CPU-GPU transfers. Note that there is a potential bias when using tabulated data with a resolution greater than the number of bins.



(a) *Single wavelength: 1 wavelength per ray 64 rays per pixel Time: 0.252 s*

(b) *Multiplexed: 32 wavelengths per ray 16 rays per pixel Time: 0.217 s*

(c) *Multiplexed: 32 wavelengths per ray 64 rays per pixel Time: 0.783 s*

(d) *Converged reference*

Figure 42-6. *Convergence comparisons between (a) single wavelength and (b,c) multiplexed approaches. The convergence is faster with multiplexing—(c) for the same number of rays per pixel and (b) for the same time—compared to the single wavelength rendering. The scene is composed of only metallic materials (i.e., conductors). All rendered images (*512×512 *pixels) are generated with an NVIDIA RTX 3070 with 32 wavelengths evenly distributed between 380 and 750 nm.*

In the following sections, we describe more extensively the boundary asset management for multiplexed rendering. The reasoning remains the same for single wavelength rendering, each bin being processed one at a time.

42.3.2 WAVELENGTH MULTIPLEXING

The efficiency of a spectral renderer can be greatly improved by using multiplexing: processing a single ray that carries multiple wavelengths (i.e., a spectrum) instead of a single wavelength. The benefits of this approach are twofold. First, it reduces the number of costly ray-intersection computations, and second, it also reduces the number of BRDF importance sampling and evaluation events.

Implementation-wise, adding simple multiplexing support to a single wavelength path tracer is straightforward. Instead of processing a single wavelength, a ray carries multiple wavelengths at once.[3] All computations are vectorized to handle all wavelengths. When propagation is wavelength-independent (e.g., no refractive materials), this approach improves the convergence significantly, as illustrated in Figure 42-6, and also reduces color noise.

This solution is GPU-friendly and can be interpreted as a binary version of the Hero Wavelength Spectral Sampling (HWSS) proposed by Wilkie et al. [26].

[3] RGB rendering may be, in a way, considered as a special case of multiplexing.

HWSS also computes and stores the probabilities that the current path is valid for the wavelengths it conveys. This allows efficient handling of media with wavelength-dependent scattering (e.g., participating media or fluorescent elements).

42.3.3 ENFORCING CONTINUOUS SPECTRAL SAMPLING WITH MULTIPLEXING

Given the GPU-oriented approach, we have additional constraints regarding asset management. To avoid costly CPU-GPU transfers, instead of uploading the assets for the sampled wavelength, we upload the assets for wavelengths at the upper and lower boundaries of each requested bin. Then, the random wavelength sampling is done on the GPU within each bin, and the assets are interpolated on the fly using the wavelength offset between the lower and upper bounds (see Figure 42-7).

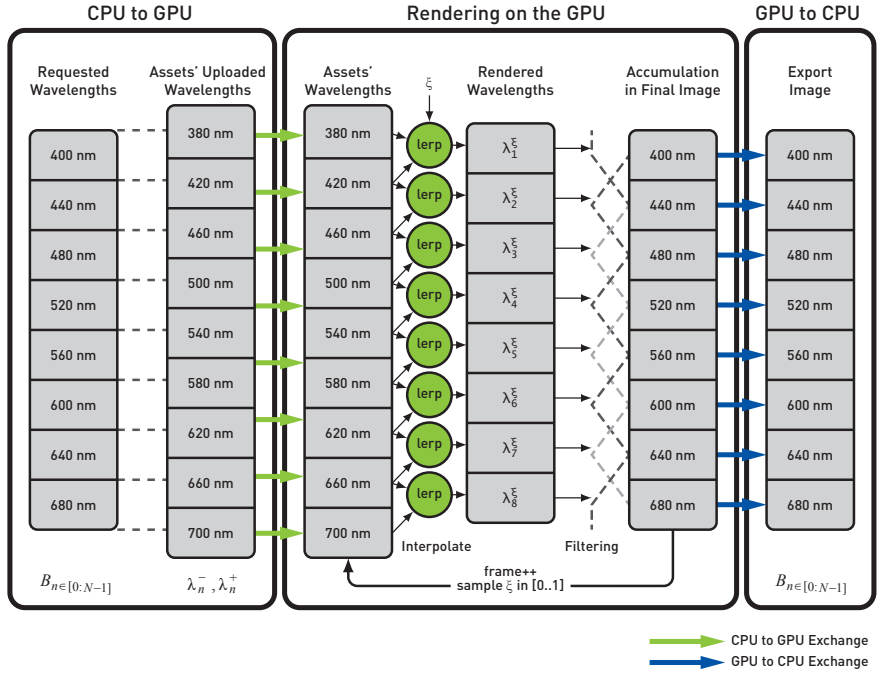


Figure 42-7. *An overview of the proposed pipeline. First, we determine the boundary of each spectral bin. Then, the assets are uploaded to the GPU for the bin boundaries. At each sample, a random offset ξ allows jittering the wavelength in the interval $[\lambda^-, \lambda^+]$. Each tabulated asset is evaluated using a linear interpolation between the two adjacent boundaries. Then, the result is accumulated to a spectral buffer, representing the requested wavelengths, and filtered to mitigate further spectral banding. Finally, the accumulation buffer is transferred to the CPU on request for saving the spectral image.*

This may introduce a small bias for tabulated assets but permits efficient rendering, by sparing a tremendous amount of data transfer. Note that such an approach remains completely unbiased with analytical spectra, or when the spectral resolution of the assets is lower than half the spectral resolution of the final image (Nyquist–Shannon sampling theorem).

UPLOADING ASSETS ON THE GPU

For a given set of N continuous bins $B_{n\in[0:N-1]}$ centered on the requested wavelengths λ_{B_n}, we upload the assets for the lower, λ_n^-, and upper, λ_n^+, boundaries of each bin (see Figure 42-7). The uploaded asset values at λ_n^- and λ_n^+ take the form of an average value, A_{λ_i}, to ensure energy conservation:

$$A_{\lambda_i} = \frac{1}{\Delta} \cdot \int_{\lambda_i - \frac{\Delta}{2}}^{\lambda_i + \frac{\Delta}{2}} A(\lambda)\, \mathrm{d}\lambda\,, \tag{42.2}$$

where λ_i is the wavelength at the boundary $\lambda_i = \lambda_n^-$ or $\lambda_i = \lambda_n^+$, $A(\lambda)$ is the value of the asset at λ, and Δ is the spectral bandwidth of the bin (i.e., $\Delta = \lambda_n^+ - \lambda_n^-$)

In the particular case of adjacent bins, we upload $N + 1$ spectral elements for each asset because the upper bound λ_{n-1}^+ of B_{n-1} matches the lower bound λ_n^- of B_n:

$$\lambda_n^+ = \lambda_{n+1}^- = \lambda_{B_n} - \frac{\lambda_{B_n+1} - \lambda_{B_n}}{2}\,. \tag{42.3}$$

At this point, we have access to $N + 1$ spectral values for each asset. However, we still propagate N wavelengths with each ray for the whole rendering process. For each of these wavelengths, we interpolate the asset value using two values from the $N + 1$ values stored on the GPU.

Note that this does not hold when using nonconsecutive bins, as in Section 42.5.3.

WAVELENGTH SELECTION

For each new ray, we sample an offset $\xi \in [0, 1]$ between the lower, λ_n^-, and upper, λ_n^+, boundaries of each spectral bin carried by this ray. Therefore, each new ray explores a different subset of wavelengths.

For a given random offset ξ, the corresponding wavelength λ_n^ξ for the bin B_n is bounded between λ_n^- and λ_n^+ and computed as follows:

$$\lambda_n^\xi = \lambda_n^- + \xi \cdot (\lambda_n^+ - \lambda_n^-)\,. \tag{42.4}$$

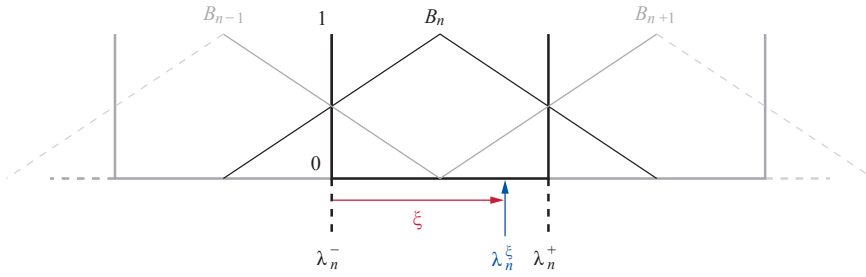


Figure 42-8. *After rendering a sample, the energy is accumulated in neighboring bins. We use triangular kernels to accumulate offset samples and attenuate potential spectral banding.*

ACCUMULATION IN SPECTRAL BINS

Once the radiance values L_e are computed for each wavelength λ_n^ξ carried by the ray, we accumulate the radiance into the final image. To mitigate potential banding artifacts, we use a triangular kernel to distribute the ray energy in neighboring bins (see Figure 42-8).

For the nth bin, the weights w are computed as follows:

$$w_{n-1} = \max(0, 0.5 - \xi), \quad w_n = 1.0 - \left|0.5 - \xi\right|, \quad w_{n+1} = \max(0, -0.5 + \xi). \tag{42.5}$$

Then, we can accumulate the resulting radiance $L_e(\lambda_n^\xi)$ in bins B_{n-1}, B_n, and B_{n+1}:

$$B_{n-1} = w_{n-1} \cdot L_e(\lambda_n^\xi), \quad B_n = w_n \cdot L_e(\lambda_n^\xi), \quad B_{n+1} = w_{n+1} \cdot L_e(\lambda_n^\xi). \tag{42.6}$$

42.3.4 SUMMARY

In this section, we first presented the different strategies for spectral sampling with their strengths and weaknesses with respect to the quality and GPU rendering constraints. We explained how to handle spectral multiplexing to improve a spectral renderer's performance by minimizing CPU to GPU transfers. Finally, we showed how to reduce spectral banding artifacts by using filtering during the accumulation step.

Our method can easily be extended to fully support HWSS with the addition of a path probability for each computed wavelength, increasing computation and memory usage. More complex filters can also be used to further reduce the spectral banding artifacts.

42.4 MULTIPLEXING WITH SEMITRANSPARENT MATERIALS

The main limitation of multiplexing appears when the scene contains semitransparent materials. In fact, this limitation arises with any material or medium where the geometrical path taken by the light depends on the considered wavelength. As discussed by Wilkie et al. [26], it forces the multiplexed renderer to operate in single wavelength mode.

In this section, we present a method to improve the efficiency of a multiplexed renderer when a ray hits a semitransparent material as such materials are the most common wavelength-dependent materials. First, we recall why semitransparent materials are not straightforward to handle and illustrate the limitation with multiplexing. Then, we present a solution to overcome partially this limitation and improve the efficiency of multiplexed renderers.

42.4.1 LIMITATION WITH SEMITRANSPARENT MATERIALS

At the wavelength scale, each medium's optical behavior can be characterized by its index of refraction (IOR). The IOR is a dimensionless number that represents, per wavelength, how fast the light speed is affected by the medium. The IOR is a complex number defined by

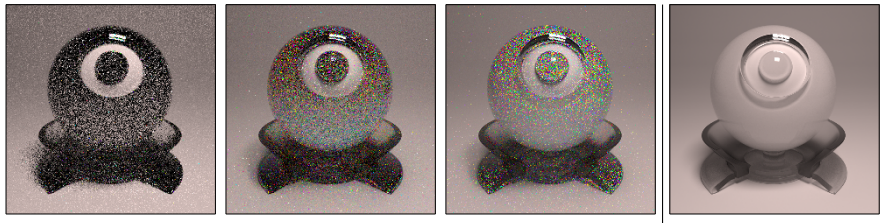
$$n(\lambda) = \eta(\lambda) + j\kappa(\lambda)\,, \tag{42.7}$$

where $\kappa(\lambda)$ represents how much a medium absorbs the light. When $\kappa \gg \eta$, the material is said to be a conductor (e.g., metals). Conductors mostly reflect or absorb light and barely transmit it, whereas dielectrics ($\eta \gg \kappa$, e.g., glass) reflect, absorb, and transmit light.[4] When a light wave changes medium, its geometrical path will be modified according to the Snell–Descartes law:

$$n_i(\lambda)\,\sin\theta_i = n_t(\lambda)\,\sin\theta_t \leftrightarrow \sin\theta_t = \frac{n_i(\lambda)}{n_t(\lambda)}\,\sin\theta_i\,, \tag{42.8}$$

where θ_i is the incident angle and θ_t is the transmitted angle, which defines the new direction of the wave. For a given θ_i, the resulting θ_t will depend indirectly on the wavelength (through the IOR). This dependence implies that only one couple (θ_i,θ_t) is valid for a specific wavelength. Because a multiplexed approach is valid as long as the light path is valid for all the wavelengths carried by the ray, *it cannot be used directly when handling such transmission events*.

[4] For more information on the IOR definition and its various implications in light behavior, we refer the interested reader to the book *Optics* by Eugene Hecht [10].



(a) *Single wavelength: 1 wavelength per ray 64 rays per pixel Time: 0.317 s*

(b) *Multiplexed: 32 wavelengths per ray 16 rays per pixel Time: 0.283 s*

(c) *Multiplexed: 32 wavelengths per ray 64 rays per pixel Time: 1.022 s*

(d) *Reference (single wavelength approach, precision of 1 nm)*

Figure 42-9. *Convergence comparisons of a scene with three semitransparent materials, the IORs of which are constant per wavelength. In this scene, the many decimation events necessary with (b,c) the multiplexed approach introduce color noise. However, multiplexing still offers a gain compared to (a) the single wavelength approach at equal rendering time. All images (*512×512 *pixels) are rendered on an NVIDIA RTX 3070 with 32 wavelengths evenly distributed between 380 and 750 nm.*

When crossing a transparent material, the rendering approach must fall back into a single wavelength propagation (i.e., the incoming spectrum is decimated to a single wavelength [6]). When the wavelength selection is done randomly, color noise reappears for this material type, as illustrated in Figure 42-9. It is important to note that this selection can be done on the fly, so that a light path without any transmission remains multiplexed. This still allows the multiplexed approach to perform well.

An option to overcome the remaining color noise problem could be to split the multiplexed ray into several single wavelength rays (each of them with its own geometrical direction [6]). However, this kind of branching is hardly recommended with CPU-based path tracers due to potential exponential workload. It may be even worse with a GPU-based engine where a compute unit is more easily overloaded than a CPU core, thus significantly increasing the rendering time.

42.4.2 IMPORTANCE SAMPLING FOR THE PROPAGATED WAVELENGTH

For (even slightly) colored dielectrics, a more GPU-friendly approach is to use importance sampling when choosing the wavelength that we want to keep, to make the best of this bad situation. Theoretically, the colored aspect of the medium comes from the IOR (both the real and imaginary parts) as stated by the Fresnel equations on the surface, but also by the type of particles (and their concentration) composing the medium. The most common way to

compute the spectral attenuation induced by the particles of the medium is to apply the Bouguer–Beer–Lambert law[5] $T_{\text{BBL}}(\lambda)$, which quantifies the transmittance rate:

$$T_{\text{BBL}}(\lambda, l) = e^{-\sigma(\lambda)l}, \tag{42.9}$$

where l is the distance (in meters) traveled by the light inside the medium and $\sigma(\lambda)$ is the absorption coefficient, related to the IOR by

$$\sigma(\lambda) = \frac{4\pi\kappa(\lambda)}{\lambda}. \tag{42.10}$$

To select the transmitted wavelength, we propose to build a 1D cumulative distribution function (CDF) for a fixed distance l, from the transmittance function $T(\lambda)$ defined as follows:

$$T(\lambda) = T_{\text{BBL}}(\lambda, 1) \cdot T_{\text{user}}(\lambda), \tag{42.11}$$

where $T_{\text{user}}(\lambda)$ is an additional and ad hoc attenuation rate allowing the user to choose to control which wavelengths are absorbed by the medium. The main benefit of building a CDF from Equation 42.11 is that it only depends on the wavelength and remains small in terms of memory footprint.

The complete importance sampling procedure is the following:

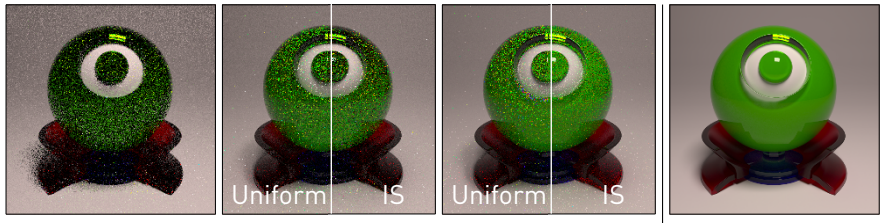
1. Build a CDF from $T(\lambda)$ for each material.[6] This is done on the CPU as a preprocess.
2. When a ray encounters a transparent material, sample a wavelength by inverting on the fly (on the GPU) the previously computed CDF: $\lambda = CDF^{-1}(x \in [0, 1])$.
3. Apply the corresponding probability density function, $pdf(\lambda) = T(\lambda)/\int T(\lambda)$.
4. Propagate the transmitted ray using only the sampled λ.

Note that for a GPU-based path tracer, only the first operation is performed on the CPU while loading the scene. The remaining operations are part of the path tracer execution on the GPU, hence they do not induce any extra CPU-GPU exchanges.

When the transmittance does not present any significant variation, one would still prefer to directly use uniform sampling because, in that case, importance sampling will be close to uniform sampling.

[5] We neglect the case of thin layers, which produces interference. In that case, the Bouguer–Beer–Lambert law is no longer valid (see Mayerhöfer et al. [16]).

[6] In our implementation, we use a 1 nm resolution for λ to compute the CDF and an arbitrary constant distance.



(a) *Single wavelength: 1 wavelength per ray 64 rays per pixel Time: 0.308 s*

(b) *Multiplexed: 32 wavelengths per ray 16 rays per pixel Time: 0.281/0.280 s*

(c) *Multiplexed: 32 wavelengths per ray 64 rays per pixel Time: 1.03/1.02 s*

(d) *Reference (single wavelength approach, precision of 1 nm).*

Figure 42-10. *Comparison of the different wavelength importance sampling (IS) procedures for a scene with different semitransparent colored materials. For this scene, the convergence rate is faster when using importance sampling (b,c, right side) rather than uniform sampling (b,c, left side) to select the propagated wavelength. Note that uniform sampling is still better than using (a) a single wavelength approach thanks to multiplexed reflections. All images (*512 $\times$ 512 *pixels) are rendered on an NVIDIA RTX 3070 with 32 wavelengths evenly distributed between 380 and 750 nm.*

Figure 42-10 illustrates how our importance sampling approach is useful with another difficult scene containing mostly colored dielectrics. At equal time, using importance sampling for the propagated wavelength greatly reduces the variance. Once again, the multiplexed approach is still better than the single wavelength approach.

As stated previously, the most accurate approach for wavelength importance sampling should also consider the incoming ray direction, the distance to the exit point, and the Fresnel transmission term. However, this aspect will be the subject of further study to assert if the gain in performance is worth the additional computational cost and memory usage.

42.4.3 SUMMARY

In this section, we have shown how multiplexed rendering can increase performance, even when dealing with strictly path-dependent materials such as dielectrics. We proposed a method to reduce the impact of the dielectric on multiplexed performance by using an importance sampling procedure to select the appropriate wavelength. This drastically reduces color noise and improves even more the efficiency. This technique is also well suited for GPU renderers with minimal CPU-GPU exchanges. Only a simple precomputation is performed, per transparent material, and only once at the scene loading stage.

This technique can also be used in conjunction with HWSS. Instead of selecting the Hero wavelength at the initial ray launching step, it may be done at the first scattering event. If this event is a dielectric intersection, this would improve HWSS efficiency where the renderer has to operate in single wavelength mode.

42.5 A STEP TOWARD REAL-TIME PERFORMANCE

Real-time rendering with Monte Carlo path tracing[7] is currently hardly possible on a home desktop computer with a tristimulus approach. If we focus on *predictive* and *accurate* rendering of complex materials, where spectral rendering is almost mandatory, it is even more difficult to achieve such frame rates, due to memory limitation and heavier workload. Furthermore, there is an additional, on-the-fly, spectral-to-RGB conversion to display the image, yet another post-process treatment.

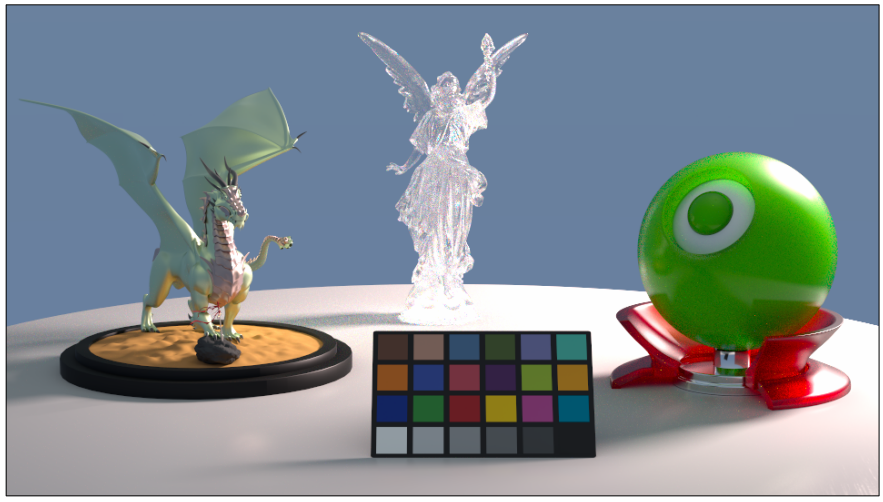
However, we demonstrate in this section that with a limited number of wavelengths and a fully multiplexed approach, one can hope to reach interactive frame rates with a spectrally accurate enough result. We also illustrate how some simple yet useful tools can be implemented to improve performance when one has limited compute capabilities or limited memory.

42.5.1 INTERACTIVE SPECTRAL RENDERING WITH MULTIPLEXING

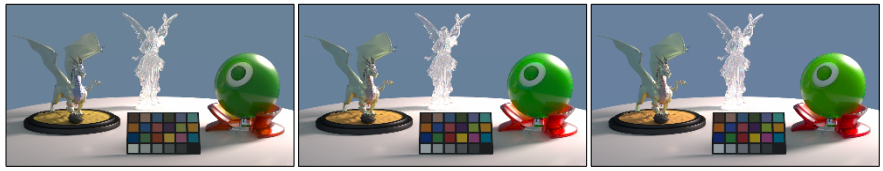
Multiplexing is a key feature when one wants efficient progressive rendering for an interactive purpose, or to aim at real-time spectral path tracing. To support this claim, we provide some results obtained with our solution Malia [5] in which all the aforementioned features are implemented. It is an academic-oriented path tracer (currently OptiX-based [20]), designed to offer predictive and accurate rendering with full support of measured and tabulated materials (e.g., coming from a BRDF acquisition process). As such, our solution is probably not optimized as well as a production one. We still wish to provide hints on the performance that can be achieved with our approach. All results are presented with respect to their RGB equivalent (using the same pipeline except for the asset management) to appreciate the relative performance.

Figure 42-11 illustrates that 16 wavelengths may be sufficient with many "everyday real-world" materials and even some complex ones (some spectra are highlighted in Figure 42-13). Sixteen wavelengths will be more than

[7] At 30 to 60 converged frames per second, with path length up to 10 bounces.



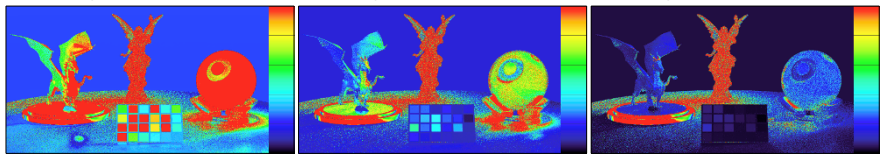
(a) *Reference image with 64 bins*



(b) *Rendering with 8 bins, 125.3 rays/pixel/second*

(c) *Rendering with 16 bins, 87.7 rays/pixel/second*

(d) *Rendering with 32 bins, 49.7 rays/pixel/second*



(e) $\Delta E^*_{2000}(a,b)$, *scale from 0 to 10*

(f) $\Delta E^*_{2000}(a,c)$, *scale from 0 to 10*

(g) $\Delta E^*_{2000}(a,d)$, *scale from 0 to 10*

Figure 42-11. *Comparisons (using CIE ΔE^*_{2000}) of the impact of the number of processed spectral bins, with converged rendering. Common materials (e.g., the MacBeth chart and the table) are depicted in a satisfying way with as few as 16 bins, and nearly perfectly depicted with 32. However, for complex materials (e.g., Lucy, see Figure 42-13 for a peak at the material), 16 bins may not be sufficient. In all cases, only eight wavelengths is not enough for such a scene. For reference, the same scene in RGB requires 6 ms to shoot one ray for each pixel (960 × 540 pixels), that is, 167.63 rays per pixel per second. We can see that the rendering time is nearly linear with the number of processed wavelengths. Performance data are given as the number of rays per pixel per second, the higher the better. All rendered images are generated with an NVIDIA RTX 3070 with wavelengths evenly distributed between 380 and 750 nm.*

enough for most applications that do not use specific and complex spectral materials (e.g., fluorescent materials and interferential materials, such as diffraction gratings or photonic crystals, see Joannopoulos et al. [13]). Note that in this claim, we are merely considering that the image is observed by a human eye. It may not stand in the case of a digital sensor with a fine spectral resolution.

With this in mind, Figure 42-11 also gives some rendering times as rays per pixel per second. We can see that, with our approach, simple scenes like the table (and the probe, see Figure 42-6) are fast to render with up to 32 rays per pixel, producing an image that is converged enough to be efficiently denoised. Targeting approximately ten true frames per second may be realistic as long as a denoising post-process is applied and is performant enough (see Section 42.6).

42.5.2 INTERACTIVITY: SPATIAL SUBDIVISION

If the aforementioned techniques are not sufficient to achieve interactivity, focusing the workload on a subset of pixels is a common approach for real-time RGB renderers. Spectral rendering is no different in this case.

Within Malia [5], as an example, this process is done in two steps. First, we define small square tiles inside the OptiX buffer (typically of four or nine pixels). Then, instead of processing one ray per pixel, we process one ray per tile. The ray spatial origin is jittered inside the tile, and its final value is stored in the corresponding pixel. Finally, when synchronizing the OptiX buffer with an OpenGL framebuffer for display, the framebuffer is filled by splatting the tiles with a simple weighted average (see Figure 42-12), where the weights are the radiance values integrated over the spectrum (that is, the Y channel of the XYZ intermediate value). Note that the content of the OptiX buffer (which may be saved) is still filled using its full original spatial resolution thanks to the jittering within the tiles.

Using square tiles with an adaptive size and splatting the intermediate result on the whole tile is an easy way to reduce the workload per frame. However, more advanced and efficient approaches can easily be used in a spectral context as long as they only operate on the spatial distribution of primary rays (e.g., path guiding by Guo et al. [9] reduces the workload by focusing on important pixels, whereas load balancing by Antwerpen et al. [1] operates in the context of multiple GPUs).

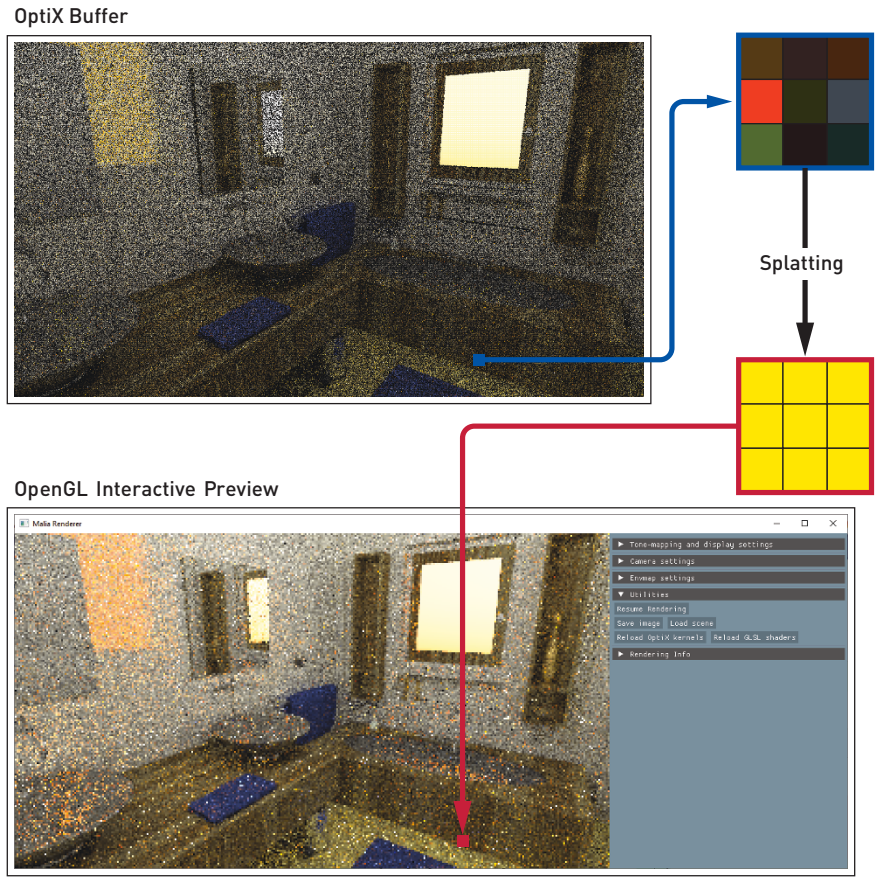


Figure 42-12. *Splatting process. To increase the interactivity, we degrade the spatial resolution by alternatively rendering one of the pixels within a given tile. The OptiX buffer remains at the full resolution to allow exporting the requested resolution.*

Although spatial subdivision is useful to reduce the per-frame workload, and therefore to increase the interactivity, the overall efficiency is significantly reduced due to the additional host calls and to the per-frame post-processing.

42.5.3 LIMITED MEMORY: MULTIPLE SPECTRAL PASSES

Memory can limit the amount of spectral bins that can be handled at once. Moreover, the size of a spectral bin impacts interactivity: a single frame takes longer to render when a higher number of spectral samples is requested. To leverage these issues, two options are possible:

1. *Continuous bin rendering:* The bins are rendered in order. We use multiple rendering passes to cover the whole spectral domain. This method has a minimal memory footprint: the upper bound of a bin and the lower bound of the next one share the same value. So, for N_{bins} rendered at once, we have to upload $N_{\text{bins}} + 1$ assets, as shown in Section 42.3. On the other hand, this method is not optimal for getting a fast preview: color previewing is only possible when all the passes are done.

2. *Interleaved progressive bin rendering:* To improve interactivity, one may want improved color rendering even with the first spectral passes. With an interleaved progressive bin rendering, the rendered bins are equally distributed over the whole spectral domain. So, color accuracy is increased at each pass. However, in this case, two bins do not share a common boundary anymore. So, for N_{bins} rendered at once, we have to upload $2 \times N_{\text{bins}}$ assets.

The first choice, continuous bin rendering, is the best candidate for maximum raw performance, so is the privileged method for offline rendering. The second choice is interesting for interactive rendering when color accuracy is preferred over raw performance to get a fast preview.

42.6 DISCUSSION

In this section we discuss open problems and aspects of spectral rendering and propose some solutions that could be tested and implemented in a production environment.

42.6.1 EFFICIENT SPECTRAL ASSET MANAGEMENT

As we discussed several times in this chapter, GPU memory may be the main constraint when considering spectral rendering. In particular, with scenes containing many textured materials, environment maps, or even tabulated spectral BSDF data, the memory management may be tricky. Uploading back and forth many textures (or buffers) has a significant impact (up to several seconds for big assets). It is always possible to reduce the number of processed wavelengths or to use a subsampling approach, as presented in Section 42.5, to circumvent memory constraint at the cost of rendering efficiency. Another straightforward solution, at least for reflective textures, is to use a better asset representation, namely an analytical one.

At the moment, in Malia, spectral textures are multi-layered images to store the values for all its wavelengths. This brute-force approach ensures that any material with sharp spectral distribution remains accurately depicted. However, many "everyday" materials do not exhibit sharp spectra. In those cases, projecting the spectrum into an appropriate basis (e.g., Jakob and Hanika [12] or Otsu et al. [19] using RGB uplifting, or Peters et al. [21] using bounded reflectance spectra) and reconstructing it on the fly may be sufficient to save memory at the cost of preprocessing the assets. Doing so permits sparing some GPU memory that can be used for assets with sharp distribution (for which uplifting is hardly possible) to their full spectral resolution. Note that for really smooth assets it may also be worth considering plain downsampling when one prefers performance over accuracy.

More extensive studies should be conducted to determine a good trade-off between spectral accuracy (to ensure that all sharp spectral features are depicted) and per-material memory management (full resolution, downsampled representation, RGB uplifting, etc.) with respect to the application considered.

Figure 42-13 illustrates part of this discussion: glasses have an absorption spectrum sharp enough to introduce significant differences when compared to plain RGB rendering, but they can be described analytically by a simple Gaussian, being more memory-friendly. However, the Lucy statue has a dielectric material with a handmade IOR (probably nonexistent but offering artistic features), which has no straightforward lightweight representation due to its sharp spectral distribution. The dragon also exhibits two handmade conductors with sharp spectral distribution. Note that, as illustrated by Figure 42-11, these spectral features are not well depicted in RGB.

42.6.2 DENOISING

Denoising is currently the most popular choice when it comes to real-time Monte Carlo path tracing. Even with an efficient RGB renderer and a high-end desktop computer, it is hard to provide at least 30 converged images per second. However, if a low-sample image (noisy) can be denoised efficiently, this goal can be attained.

The idea also applies to a spectral renderer. One may attempt to denoise a RGB image obtained from a spectral renderer with a real-time-capable RGB denoiser. Though such an approach is beyond the scope of this chapter, we must point out that one must be sure that a RGB algorithm can correctly

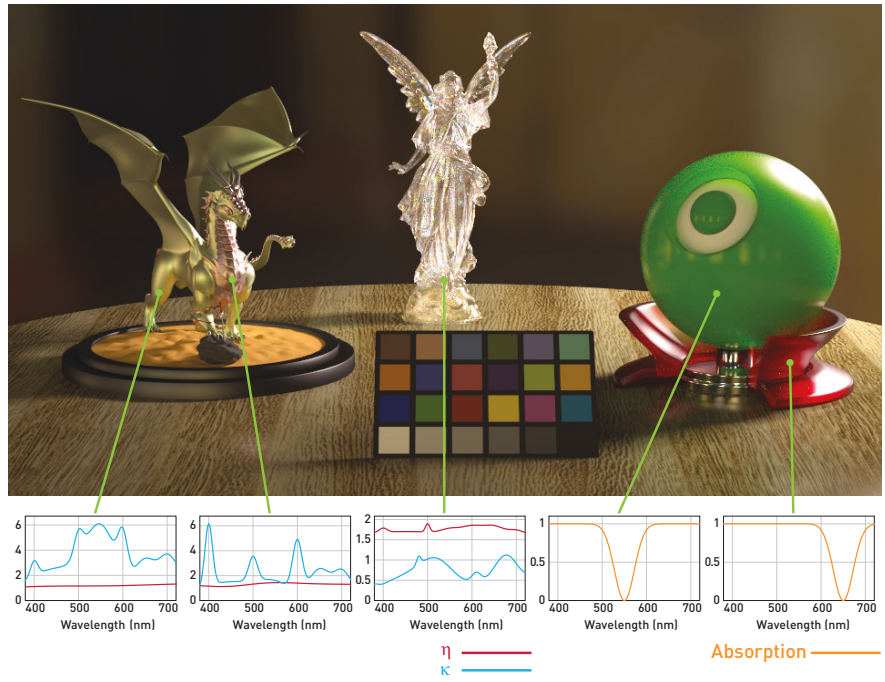


Figure 42-13. *The scene from Figure 42-11, illustrating examples of strong spectral dependency for the dragon, the Lucy statue, and the probe. These sharp distributions may require preprocessing to be stored more efficiently on the GPU, especially if they are set to vary spatially, such as in a texture.*

handle wavelength-dependent features and the potential color noise induced by the spectral rendering process. Further research may be necessary to assess this compatibility, to extend the RGB denoiser to the spectral domain, or to propose a fast and efficient spectral denoiser.

Note that our current concerns with the denoising of spectral images only apply in the context of rendering for predictive purposes. For entertainment purposes, if spectral features are required, one may look at some non-predictive-friendly denoising approaches. Some approaches (e.g., Liu et al. [15]) efficiently denoise a rendering with only one or two rays per pixel but introduce a potential mathematical bias by tweaking light transport integrals for built-in denoising. This kind of approach may be spectral-friendly, as it is incorporated in the light transport itself, but will not be predictive due to the potential bias.

42.7 CONCLUSION AND OUTLOOK

In this chapter, we have introduced spectral rendering and motivated its usage when color accuracy is desired or when specific physical effects need to be handled (e.g., dispersion, polarization, and fluorescence). We have illustrated the limitation of the tristimulus approach for light transport. Furthermore, we have introduced a pipeline for efficient spectral rendering on the GPU that limits memory usage and data exchanges between the device and the host. Finally, we have also introduced some optimizations to improve interactivity while retaining the predictive nature of the renderer.

Due to its still early adoption, spectral rendering is still a niche in the entertainment industry, but has become increasingly popular in recent years thanks to increased computational power and memory. We believe that spectral rendering will also generate more interest for GPU rendering despite the challenges it brings in terms of memory and bandwidth. GPU rendering offers an efficient way for providing both spectral and interactivity features in the context of predictive rendering. Recent academic-oriented spectral renderers have already opened this lead.

ACKNOWLEDGMENTS

This project was supported by the Agence Nationale de la Recherche Project VIDA (ANR-17-CE23-0017) as well as the Regional Nouvelle-Aquitaine Grant SIMOREVA-360. High dynamic range (HDR) environment maps were freely provided by HDRI Haven [27] and converted to spectral HDR images with our own tool. The original *bathroom* scene was modeled by McGuire [17]. Its materials were adapted and modified to transform the RGB scene into a spectral one. The dragon was freely downloaded from Bitterli's resources [2].

REFERENCES

[1] Antwerpen, D. v., Seibert, D., and Keller, A. A simple load-balancing scheme with high scaling efficiency. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 127–133. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2_10.

[2] Bitterli, B. Rendering resources. https://benedikt-bitterli.me/resources/, 2016.

[3] Blender Foundation. Cycles. https://www.cycles-renderer.org/, 2021.

[4] Caredda, C., Mahieu-Williame, L., Sablong, R., Sdika, M., Guyotat, J., and Montcel, B. Optimal spectral combination of a hyperspectral camera for intraoperative hemodynamic and metabolic brain mapping. *Applied Sciences*, 10(15):5158:1–5158:23, 2020. DOI: 10.3390/app10155158.

[5] Dufay, A., Murray, D., Pacanowski, R., et al. The Malia rendering framework. https://pacanows.gitlabpages.inria.fr/MRF, 2019.

[6] Evans, G. F. and McCool, M. D. Stratified wavelength clusters for efficient spectral Monte Carlo rendering. In *Proceedings of the Graphics Interface 1999 Conference, June 2-4, 1999, Kingston, Ontario, Canada*, pages 42–49, 1999. http://graphicsinterface.org/wp-content/uploads/gi1999-7.pdf.

[7] Fascione, L., Hanika, J., Fajardo, M., Christensen, P., Burley, B., and Green, B. Path tracing in production—Part 1: Production renderers. In *ACM SIGGRAPH 2017 Courses*, 13:1–13:39, 2017. DOI: 10.1145/3084873.3084904.

[8] Fascione, L., Hanika, J., Leone, M., Droske, M., Schwarzhaupt, J., Davidovic, T., Weidlich, A., and Meng, J. Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics*, 37(3):31:1–31:18, 2018. DOI: 10.1145/3182161.

[9] Guo, J. J., Bauszat, P., Bikker, J., and Eisemann, E. Primary sample space path guiding. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas and Implementations*, pages 73–82, 2018. DOI: 10.2312/sre.20181174.

[10] Hecht, E. *Optics*. Pearson, 5th edition, 2016.

[11] Jakob, W. Mitsuba 2: Physically based renderer. http://www.mitsuba-renderer.org, 2010.

[12] Jakob, W. and Hanika, J. A low-dimensional function space for efficient spectral upsampling. *Computer Graphics Forum (Proceedings of Eurographics)*, 38(2):147–155, Mar. 2019. DOI: 10.1111/cgf.13626.

[13] Joannopoulos, J. D., Johnson, S. G., Winn, J. N., and Meade, R. D. *Photonic Crystals: Molding the Flow of Light*. Princeton University Press, 2nd edition, 2008.

[14] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The iray light transport simulation and rendering system. In *ACM SIGGRAPH 2017 Talks*, 34:1–34:2, 2017. DOI: 10.1145/3084363.3085050.

[15] Liu, E., Llamas, I., Cañada, J., and Kelly, P. Cinematic rendering in UE4 with real-time ray tracing and denoising. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 289–319. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2_19.

[16] Mayerhöfer, T. G., Pahlow, S., and Popp, J. The Bouguer–Beer–Lambert law: Shining light on the obscure. *ChemPhysChem*, 21(18):2029–2046, 2020. DOI: https://doi.org/10.1002/cphc.202000464.

[17] McGuire, M. Computer graphics archive. https://casual-effects.com/data, 2017.

[18] Nimier-David, M., Vicini, D., Zeltner, T., and Jakob, W. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6):203:1–203:17, Dec. 2019. DOI: 10.1145/3355089.3356498.

[19] Otsu, H., Yamamoto, M., and Hachisuka, T. Reproducing spectral reflectances from tristimulus colours. *Computer Graphics Forum*, 37(6):370–381, 2018. DOI: https://doi.org/10.1111/cgf.13332.

[20] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, July 2010. DOI: 10.1145/1778765.1778803.

[21] Peters, C., Merzbach, S., Hanika, J., and Dachsbacher, C. Using moments to represent bounded signals for spectral rendering. *ACM Transactions on Graphics*, 38(4):136:1–136:14, July 2019. DOI: 10.1145/3306346.3322964.

[22] Pharr, M. PBRT version 4. https://github.com/mmp/pbrt-v4, 2020.

[23] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[24] Subtil, N. and Werness, E. NVIDIA RTX: Enabling Ray Tracing in Vulkan. Presentation at GPU Technology Conference, https://on-demand.gputechconf.com/gtc/2018/presentation/s8521-advanced-graphics-extensions-for-vulkan.pdf, March 27, 2018.

[25] The ART development team. The Advanced Rendering Toolkit. https://cgg.mff.cuni.cz/ART, 2018.

[26] Wilkie, A., Nawaz, S., Droske, M., Weidlich, A., and Hanika, J. Hero wavelength spectral sampling. In *Proceedings of the 25th Eurographics Symposium on Rendering*, pages 123–131, 2014. DOI: 10.1111/cgf.12419.

[27] Zaal, G., Majboroda, S., and Mischok, A. HDRI Haven. https://hdrihaven.com/.