BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

HIGH DYNAMIC RANGE RENDERING OF VIRTUAL 3D SCENES

ZOBRAZOVÁNÍ VIRTUÁLNÍCH 3D SCÉN S VYSOKÝM DYNAMICKÝM ROZSAHEM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	**VADIM GONCEARENCO**
SUPERVISOR VEDOUCÍ PRÁCE	**Ing. JAN PEČIVA, Ph.D.**

BRNO 2024

Bachelor's Thesis Assignment

153645

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Goncearenco Vadim**
Programme:	Information Technology
Title:	**High dynamic range rendering of virtual 3D scenes**
Category:	Computer Graphics
Academic year:	2023/24

Assignment:

1. Get familiar with high dynamic range (HDR) rendering of 3D scenes and with associated technologies.
2. Design Vulkan-based demonstration application utilizing selected methods from the area of HDR rendering.
3. Implement the application and demonstrate its capabilities on appropriate graphic scenes.
4. Discuss your results and possible future development. Compare visual quality when using HDR and when not using it.
5. Publish you work on internet. Consider making your code available under one of open-source licenses.

Literature:

- follow the instructions of the supervisor

Requirements for the semestral defence:
Application prototype.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Pečiva Jan, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	10.11.2023

Abstract

This bachelor's thesis covers the topic of High Dynamic Range (HDR), specifically various HDR techniques that are widely used in the field of computer graphics. Additionally, it covers the Vulkan API and its application for HDR rendering of virtual 3D scenes. The practical part of this thesis is a 3D rendering application, which purpose is to demonstrate the practical implementation of the described HDR techniques, such as global tone mapping operators (photographic, filmic and ACES-based), dynamic eye adaptation, spatially-variant tone mapping based on bilateral filter, exposure fusion, and an advanced bloom effect.

Abstrakt

Tato bakalářská práce je zaměřena na téma Vysokého Dynamického Rozsahu (HDR), konkrétně různých HDR technik, které jsou široce používány v oblasti počítačové grafiky. Navíc se zabývá Vulkan API a jeho aplikací pro HDR vykreslováni virtuálních 3D scén. Praktická část této práce je aplikace pro renderování 3D scén, jejímž účelem je demonstrovat praktickou implementaci popsaných HDR technik, jako jsou globální operátory mapování tónu (fotografické, filmové a založené na ACES), lokální operátor mapování tónu založený na bilaterálním filtru, dynamická adaptace oka, exposure fusion a pokročilý bloom efekt.

Keywords

High dynamic range, HDR, tone mapping, exposure, eye adaptation, bilateral filter, exposure fusion, bloom, image processing, image filtering, 3D rendering, 3D scene, C++, Vulkan API.

Klíčová slova

Vysoký dynamický rozsah, HDR, mapování tónů, expozice, adaptace oka, bilaterální filtr, exposure fusion, bloom, zpracování obrazu, filtrování obrazu, 3D renderování, 3D scéna, C++, Vulkan API.

Reference

GONCEARENCO, Vadim. *High dynamic range rendering of virtual 3D scenes.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pečiva, Ph.D.

Rozšířený abstrakt

Tato bakalářská práce se zabývá tématem High Dynamic Range (HDR), což je klíčová technologie v oblasti počítačové grafiky, která pomáhá překonávat omezení tradičních metod renderování tím, že věrně zachycuje a reprodukuje složitosti světla a barev.

Práce zkoumá různé metody běžně používané v této oblasti a ukazuje, jak mohou být využity k HDR renderování virtuálních 3D scén. Praktickým cílem je vyvinout aplikaci pro 3D renderování, která demonstruje implementaci popsaných HDR technik, konkrétně: *globální operátory mapování tónu (fotografické, filmové a založené na ACES), dynamickou adaptaci oka, prostorově variabilní mapování tónu využivajicí bilaterální filtr, exposure fusion*, a pokročilý *bloom efekt.* Každá jednotlivá technika je pak zhodnocena z hlediska *vizuální kvality* a *efektivnosti.*

Protože HDR monitory nejsou stále příliš běžné v dnešní době, začal jsem svým výzkumem zkoumáním metod mapování HDR obrázků na Standardní Dynamický Rozsah, který lze zobrazit prakticky na jakémkoli zařízení. To lze dosáhnout pomocí *operátorů mapování tónu.* Jednodušší operátory jsou *globální (nebo uniformní)*, což jsou v podstatě pouze matematické funkce, které jsou aplikovány stejným způsobem pro barvu každého pixelu obrázku. Takové operátory mohou být efektivní a snadně implementovatelné, ale nemusí stačit k dosažení lepší a realističtější vizuální kvality obrázku.

Mým dalším krokem ve výzkumu bylo najít více sofistikované řešení problému, a proto jsem začal zkoumat oblast *lokálních (nebo prostorově variabilních)* operátorů mapování tónu. Tyto jsou obvykle náročnější na výkon a určitě obtížnější k integrování do renderovacího pipeline, protože jejich implementace často zahrnuje několik post-processing kroků k dosažení požadovaného výsledku. Nicméně takové operátory se v posledních $20-25$ letech staly stále populárnějšími, protože výkon osobních počítačů se významně zvýšil. V této práci jsem se rozhodl prozkoumat jeden z nich, který jsem zjistil, že je relativně efektivní a dostatečně jednoduchý k integrování do ukázkové aplikace. Tento operátor byl vynalezen v roce 2002 a je založen na *bilaterálním filtru*, který je také popsán v této práci.

Kromě toho jsem se v této práci zabýval i tématem *dynamické adaptace oka.* Proces adaptace oka lze simulovat výpočtem *průměrné luminance* scény (*pomocí histogramu luminance*) a následným upravením luminance následujícího snímku na základě vypočítaného průměru a uplynulého času pomocí funkce *inverzního exponentu.*

Při dalším zkoumání oblasti HDR jsem narazil na novější a složitější techniku nazvanou *exposure fusion.* Tato technika dokáže produkovat působivé vizuální výsledky, které jsou jední z nejlepšich, z pohledu zachování zobrazitelného rozsahu jasu. Funguje tak, že smísí různé úrovně expozice stejného obrázku na základě několika vah, které jsou spočítány pro každý pixel. Obrázky jsou míchány na různých prostorových frekvencích, což umožňuje plynulé přechody mezi různě osvětlenými částmi obrázku a zároveň se vyhýbá *"halo"* artifaktům.

Výzkumná část této práce je zakončena implementací bloom efektu. Jsou zde zkontrolovány a porovnány dva přístupy k tomuto efektu, přičemž ten pozdější (z roku 2014) je realističtější a vizuálně přitažlivější. Realističtější přístup je poté implementován v ukázkové aplikaci.

Praktickým výsledkem této práce je ukázková aplikace pro 3D renderování vytvořená pomocí grafického API Vulkan a programovacího jazyka C++. Umožňuje uživateli načítat různé 3D scény a HDR skyboxy k posouzení různých HDR post-processing efektů. Efekty lze libovolně zapínat/vypínat a kombinovat.

Při posuzování jednotlivých implementovaných HDR technik jsem dospěl k závěru, že *exposure fusion* poskytuje nejlepší vizuální výsledky ve srovnání s ostatními metodami

mapování tónu. Avšak vyžaduje větší výpočetní výkon a správné nastavení individuálních úrovní expozice pro každou scénu. Efekt adaptace oka obecně funguje podle očekávání, avšak obvykle vyžaduje nějaké předchozí nastavení (například nastavení *min* a *max* luminance pro histogram, mezí histogramu atd.). Efekt květu produkuje velmi dobré a relativně realistické vizuální výsledky, ale má drobnou vadu, a to *"stair-step" artefakty*, které mohou být viditelné při nižším rozlišení. Řešením tohoto problému může být změna nebo vylepšení metody, vzorkování textur během procesu upsampling nebo downsampling.

High dynamic range rendering of virtual 3D scenes

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Jan Pečiva Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Vadim Goncearenco
May 3, 2024

Acknowledgements

I would like to express thanks to my supervisor Mr. Ing. Jan Pečiva Ph.D. especially for his willingness to help, patience, and for his substantive professional advice, which greatly assisted me in the preparation of this thesis.

Contents

List of Figures

Chapter 1

Introduction

In the domain of computer graphics, there has always been the urge to achieve better and more realistic visual experience. High Dynamic Range (HDR) is a revolutionary technology that helps to overcome the limitations of traditional rendering methods by faithfully capturing and reproducing the complexities of light and color. It is achieved thanks to a broader spectrum of luminance levels and color gamuts, which provides unparalleled realism and depth to the final image.

This thesis is focused on deeper understanding of HDR concept and describe the main HDR techniques that are used for rendering of virtual 3D scenes, as well as describing the ways to implement them.

Chapter 2 serves as an in-depth overview of different concepts from the domain of High Dynamic Range. It is a theoretical summary that aims to provide a comprehensive understanding of the subject and explore relevant computer graphics techniques. The chapter begins with an overview of the HDR concept. It then reviews several approaches to global tone mapping operators, followed by a discussion of exposure and the process of automatic eye adaptation. Subsequently, the study examines two local tone mapping techniques: one utilizing a bilateral filter and the other employing exposure fusion. Finally, the chapter concludes with a section on the bloom effect, where two approaches to its implementation are reviewed.

In Chapter 3 the implementation of demo application is described. It starts with an overview of Vulkan API and continues with the application structure. However, the main focus of this chapter is the implementation of HDR techniques that were described earlier. This chapter also contains the assessment of visual results of the implemented techniques, which includes visual comparison and subjective opinion.

Chapter 4 contains a description of conducted testing of the application. It contains the description of characteristics of testing devices and compares performance of individual techniques on each device.

Chapter 2

State of the art

2.1 About High Dynamic Range

The term dynamic range is mainly used in the domain of signal processing in relation to image, video, or audio. In the context of images, dynamic range refers to the difference of brightness between the darkest and the lightest color that is present in an image.

Logically, in the real world, the dynamic range is essentially unlimited, but our eyes can perceive only a limited range of real-world brightness. The overall range of light intensity that our eye can perceive was estimated to be roughly between 10^{-6} to $10^{8} cd/m^{2}$. However, the range we can perceive at any moment in time measures about 5 orders of magnitude (10^5 e.g. between 10^2 and 10^7). This is due to the fact that our eyes need time to adapt to certain lighting conditions, which takes some time [4].

Speaking about other ways to capture the world image, contemporary digital cameras for example have a dynamic range of just about 3 orders of magnitude. Conventional LCDs have a dynamic range of about 2 to 3 orders of magnitude while prototype HDR displays measure 4 to 5 orders [4].

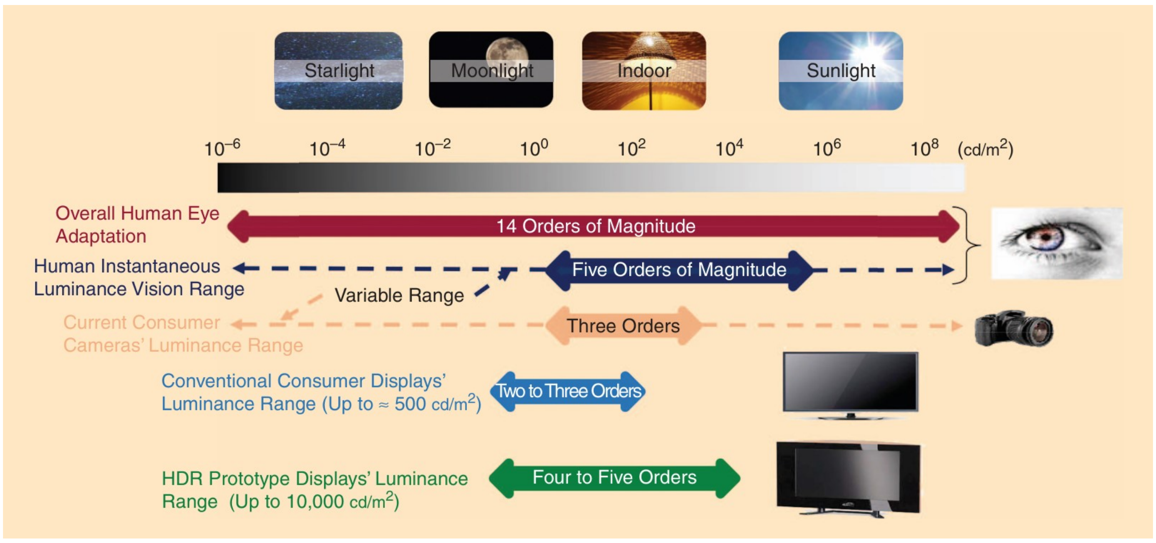

Starlight
Moonlight
Indoor
Sunlight
10^{-6}
10^{-4}
10^{-2}
10^{0}
10^{2}
10^{4}
10^{6}
10^{8}
(cd/m^2)
Overall Human Eye Adaptation
14 Orders of Magnitude
Human Instantaneous Luminance Vision Range
Five Orders of Magnitude
Variable Range
Current Consumer Cameras' Luminance Range
Three Orders
Conventional Consumer Displays' Luminance Range (Up to ≈ 500 cd/m^2)
Two to Three Orders
HDR Prototype Displays' Luminance Range (Up to 10,000 cd/m^2)
Four to Five Orders

Figure 2.1: Illustration of dynamic range of Human Visual System and different output devices.

Image taken from [4].

2.1.1 Tone mapping

Considering the limited availability of HDR displays on the market which still have a fairly limited dynamic range, there is a clear need for a way to compress the high dynamic range of an HDR photograph or a rendered HDR scene to a lower dynamic range for display purposes. This can be achieved through the use of tone mapping operators, which are essentially mathematical functions that are used to map a wider range of values to a narrower one.

There are several categories of tone mapping operators that were developed, these are: global operators, local operators, frequency operators, and gradient operators. Global operators apply compression on all image pixels uniformly, while in the case of local operators range compression for a given pixel depends on the surrounding pixels. Frequency operators compress the spatial frequency of the image and gradient operators compress the gradient image [30].

The two major categories of global and local operators will be further discussed in the following chapters.

2.2 Global tone mapping operators

2.2.1 Photographic tone mapping

Arguably the most commonly cited tone mapping operator is the one introduced in 2002 by *Reinhard et al.* in their paper [29]. This operator is noted for its computational efficiency, widespread application across various contexts, and especially its suitability for real-time rendering. While the aforementioned paper presents both global and local operators, our focus will be on the global variant.

This tone reproduction operator draws from proven techniques in the field of photography and aims to achieve realistic image representation. It does not strive for perfect imitation of the actual photographic process but rather builds upon the so-called zone system described by Ansel Adams in the 1980s [2].

Initially in the paper an operator is presented which enables the user to set (or rather *shift*) the dynamic range of an image based on its *key value* [1]. Illustration of key value variation in an image as presented in the paper by Reinhard et al. can be seen on Figure 2.2.

The operator consists of two steps.

First, the term $\bar{L}_w$ is computed, which represents an approximation to the key of the image.

$$\bar{L}_w = \frac{1}{N} \exp\left(\sum_{x,y} \log\left(\delta + L_w(x,y)\right) \right) \tag{2.1}$$

Where $L_w(x,y)$ is the luminance of the pixel (x,y), N is the total number of pixels in the image and δ is a small value to avoid the singularity of $log(0)$ in case completely black pixels are present in the image[2].

[1]The key of a scene represents an average measure of its subjective brightness.

[2]The logarithmic average is used instead of arithmetic average to account for a nonlinear response to a linear increase in luminance.

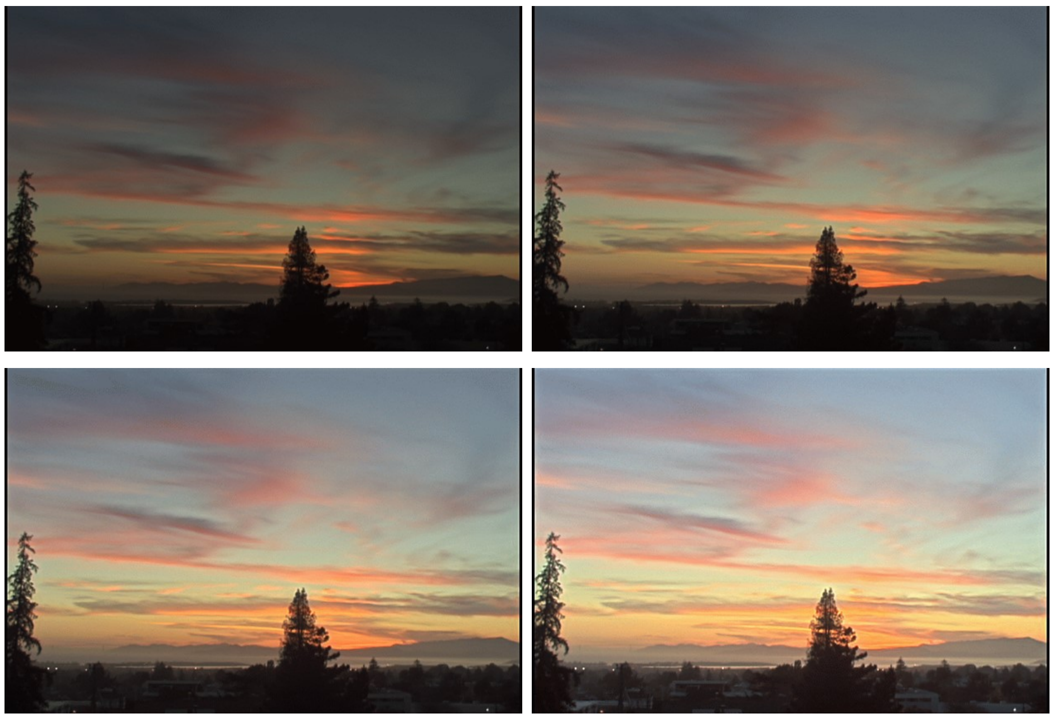


Figure 2.2: Same image with different *key value*. Top left to bottom right: 0.09, 0.18, 0.36, 0.72.

Image taken from [29]

Then, using the estimated key value, every pixel of the image is mapped to middle-grey[3] (0.18 on a scale from zero to one). This is done through the equation:

$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y) \tag{2.2}$$

where $L(x,y)$ is a scaled luminance and $a = 0.18$. The user can map the image to different values of a, depending on whether the image is supposed to have a low or high key value. Typically, a would vary from 0.18 up to 0.36 and 0.72 (high key) or vary down to 0.09, and 0.045 (low key).

The main problem with this operator is that many scenes have predominantly a normal dynamic range, with a few high-luminance regions near highlights or in the sky, so it is not desirable to shift all the luminance values by an equal amount. It is mentioned that this can be mitigated with an "s"-shaped curve operator like the ones traditionally used in photography or film. Such an operator would compress both the high and low luminance values, but instead further in the paper a different approach is chosen, which instead aims to only compress the high luminances:

$$L_d(x,y) = \frac{L(x,y)}{1 + L(x,y)} \tag{2.3}$$

This formula is guaranteed to bring all luminances within displayable range. For the case when it is not desirable, an extended operator is presented, that allows high luminances to burn out in a controllable fashion:

[3]Middle gray is a tone that is perceptually about halfway between black and white.

$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L_{\text{white}}^2}\right)}{1 + L(x,y)} \tag{2.4}$$

where L_{white} is the smallest luminance that will be mapped to pure white. This function acts as a blend between Equation 2.2 and a linear function. If L_{white} value is set to the maximum luminance in the scene or higher, no burn-out will occur. If it is set to infinity, then the function reverts to Equation 2.2. By default, L_{white} should be equal to the maximum luminance in the scene. Result of applying this equation can be seen on Figure 2.3.

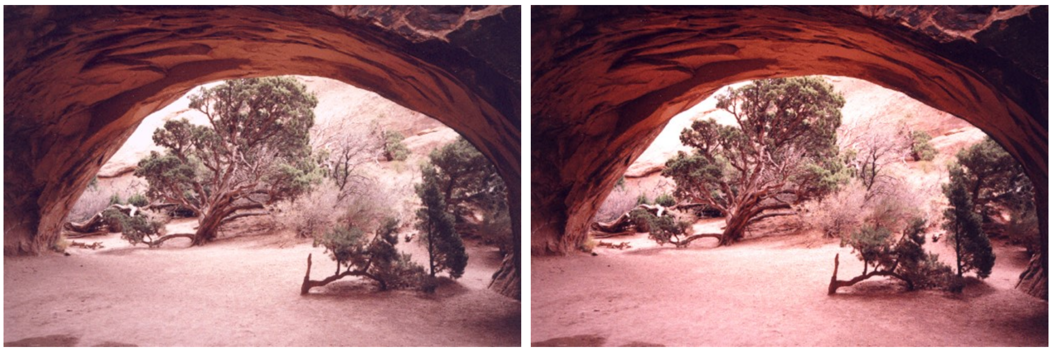


Figure 2.3: Result of applying Reinhard's global operator (Equation 2.4). Input (left) vs. output (right).

Image taken from [29]

Authors claim that compression provided by this operator is sufficient for most HDR scenes, but in case of very high dynamic range, important detail may still be lost. Further more sophisticated approaches are proposed in the same paper, including a local(spatially-varying) tone mapping operator that imitates *dodging-and-burning* [4] technique from photography, but those are out of scope of this chapter.

To visualize individual equations I have plotted them on Figure 2.4 and Figure 2.5.

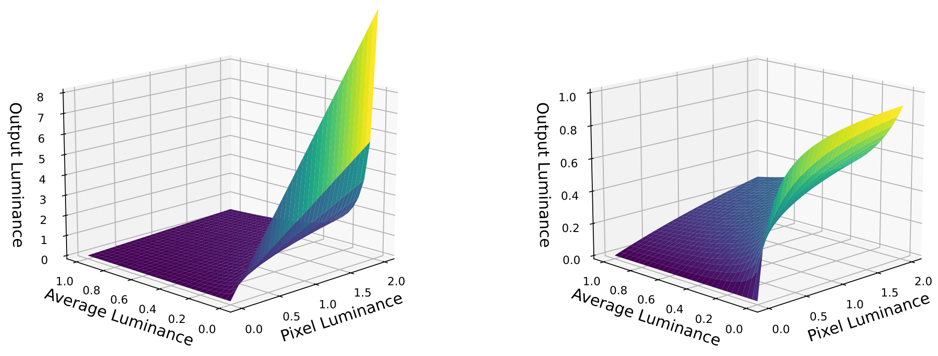


Figure 2.4: Result of equation 2.2 (left) vs. applying both equation 2.2 then 2.3 (right).

[4]A technique used to alter the exposure of certain areas of a photograph.

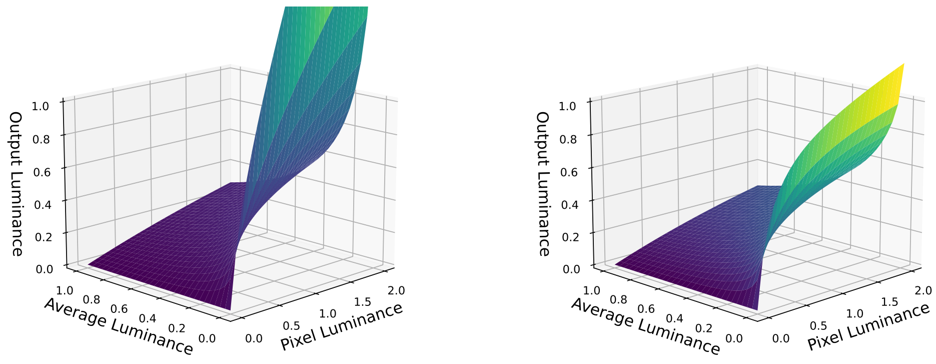


Figure 2.5: Result of equation 2.4 with $L_{white} = 2.0$ (left) vs. $L_{white} = 6.0$ (right).

2.2.2 Filmic tone mapping

Filmic tone mapping operators that are used in computer graphics field were for the most part inspired by the photosensitive characteristics of photographic film. These characteristics are also referred to as *sensitometric curves* or *density curves*. An example of such curve can be seen on Figure 2.6.

Tone mapping operators that mimic density curves of photographic films aim to achieve highly contrasted images with deep black tones and produce visually appealing scenes reminiscent of those created in the film industry.

The images generated by these operators have captivated computer graphics professionals due to their visual appeal, making various operators of this kind favorites among video game developers.

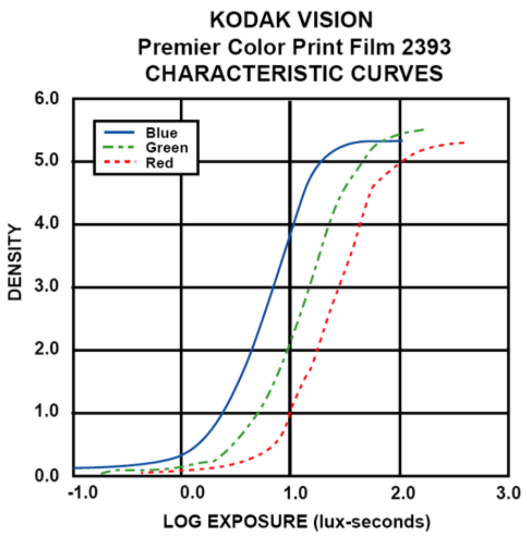


Figure 2.6: Example of Kodak film characteristic curves.
Image taken from [14]

The first mention of a filmic tone mapping operator (TMO) being used in the video game industry is attributed to *Haarm-Pieter Duiker* who presented it in the year 2006 [10] at *Electronic Arts* [5].

However, several years later a more elaborated filmic TMO was presented by *John Hable* in the year 2010 [14] during his *GDC*[6] talk. He developed this operator during his work on the game *Uncharted 2* at *Naughty Dog* game company.

The operator developed by John Hable has the form of an “S”-like curve with three distinct parts: *toe*, *linear phase* and *shoulder* (viz. Figure 2.7)

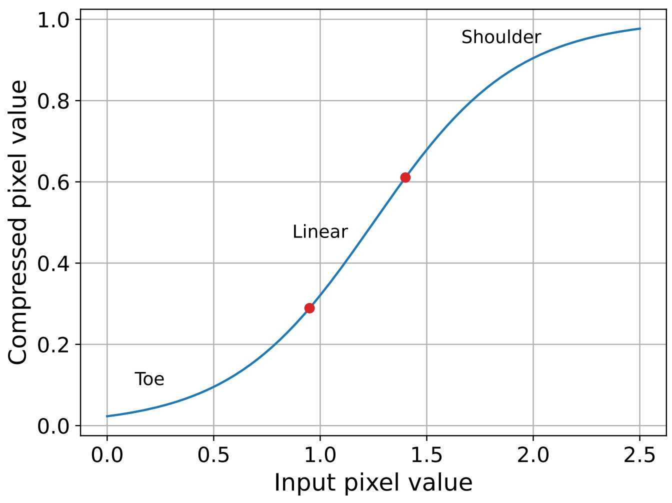


Figure 2.7: Three parts of an “S”-like filmic curve.

The operator has the following formula:

$$f(x) = \frac{x(Ax + CB) + DE}{x(Ax + B) + DF} - \frac{E}{F} \tag{2.5}$$

Where x is the original pixel value of an HDR image, *f(x)* is the compressed pixel value, and parameters A through F are adjustable coefficients determining the configuration of the three distinct parts of the curve.

The meanings of individual parameters are as follows:

- A - shoulder strength, indicates the sharpness of transition from linear phase to the shoulder.
- B - determines the strength of the linear phase.
- C - defines the slope of the curve in the linear part.
- D - toe strength (steepness of the toe phase).
- E, F - numerator and denominator of the toe. Their ratio determines the toe angle.

[5] *Electronic Arts Inc. (EA)* is an American video game company founded in 1982 that played a pioneering role in the early game industry.

[6] *Game Developers Conference (GDC)* is an annual event for video game developers, featuring a variety of game-related tutorials and lectures presented by industry professionals.

The last step is to divide the newly obtained value by compressed white point value (W).

$$y = \frac{f(x)}{f(W)} \tag{2.6}$$

For all the described parameters Hable presents the following default configuration: A = 0.22, B = 0.30, C = 0.10, D = 0.20, E = 0.01, F = 0.30, and W = 11.2.

The resulting graph of the TMO can be seen on Figure 2.8. As we can see from the graph author decided to minimize the toe angle which results in almost linear increase in intensity of darker tones. At the same time more prominent Shoulder part of the curve results in a smooth and dampened increase in brightness of highlights. For the purpose of better visualization Figure 2.9 contains images of Hable's operator applied inside Uncharted 2 game.

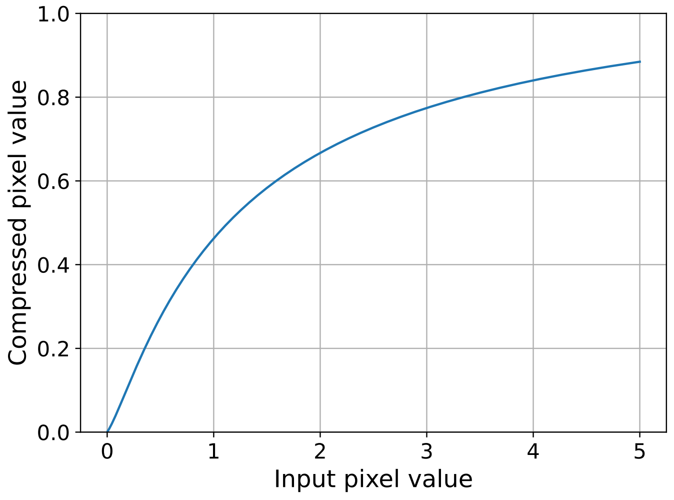


Figure 2.8: Graph of *John Hable's* Uncharted 2 tone mapping operator.

To conclude, it is worth mentioning that this operator has since became widely adopted in the video game industry and even used in such big titles as *GTA V* [7] and *Doom (2016)* [8].

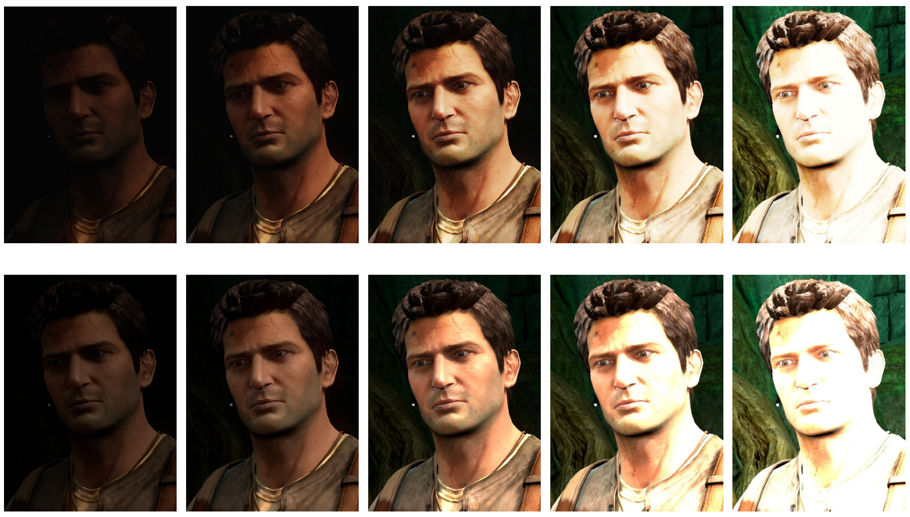


Figure 2.9: *Uncharted2 filmic* (top) vs. *linear* (bottom) mapping. Exposure ranging from -4 to $+4$ *EV steps.*

Images taken from [14]

2.2.3 ACES tone mapping

Traditional motion picture workflow was fixed for the most part and relied on the film negatives and their fixed display properties. However, with the advent of digital technologies, many different media became available for acquiring and presenting the image content.

Source material can come from one or several of the dozens of available capture formats, while the produced digital content may need to be played in theaters, as well as on SDR (Standard Dynamic Range) and HDR TVs, mobile devices or even VR headsets. All of these output devices have different range of displayable colors, otherwise named as *gamut*, which is essentially a subset of a perceivable color space (*CIE 1931*[7]). Figure 2.10 shows comparison of RGB gamut of different display formats.

Another issue is that during post-production nowadays the image needs to come through various post-processing stages that each might require a conversion to a different color format. Without a proper standard to guide the various color format conversions the quality of the image would be lost [19].

This is where ACES comes to the resque. *Academy Color Encoding System (ACES)* began its development in 2004 as an image encoding system that provides color accurate workflow for the motion picture industry (i.e. digital films, video games etc.). As a large color space, ACES preserves high image quality until the final conversion to the viewing formats.

The fundamental components of ACES are [1]:

- Preservation of the available exposure and color range of digital cameras and film negatives, ensuring seamless integration throughout the production pipeline.
- Consistent and predictable display of images across a diverse array of display devices.
- Archive-ready digital image file format along with accompanying metadata.

[7] *CIE 1931* is the color space defined in 1931, which quantifies human color perception based on a standardized model of the human visual system.

[8] https://en.wikipedia.org/wiki/Rec._2020

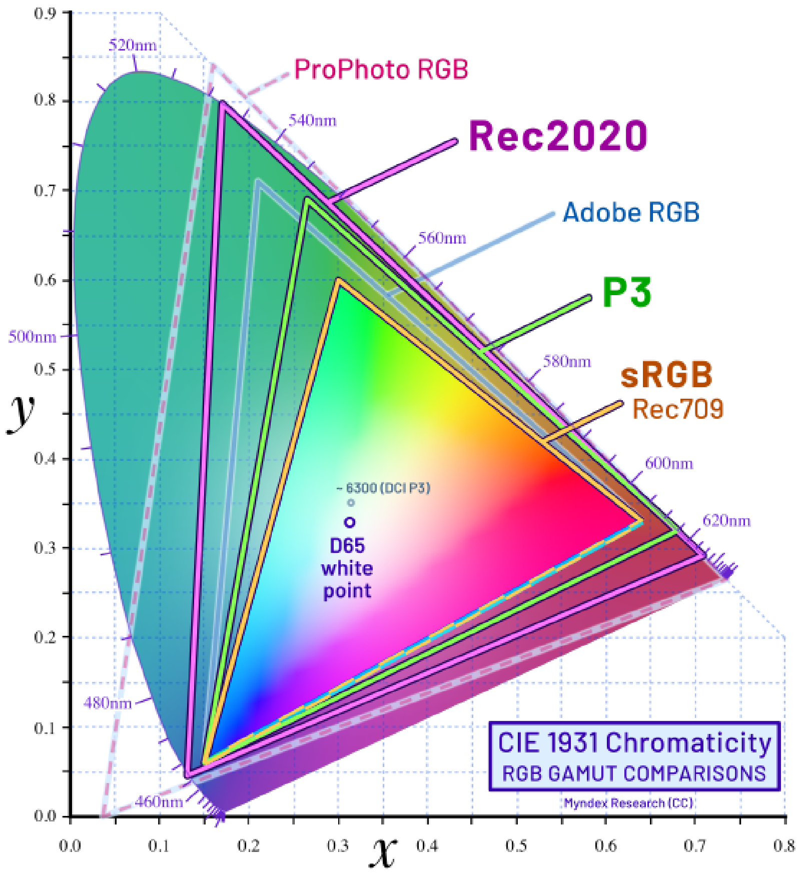


Figure 2.10: CIE 1931 RGB color space. Gamut comparisons. Image taken from Wikipedia[8].

Krzysztof Narkowicz in his blog article [22] introduced a tone mapping operator based on ACES. He developed a *filmic curve* based on the data sampled from ACES *Output Transform*.

According to Wikipedia[9], *Output Transform* is the mapping from scene-reffered colorimetry to the output-referred colorimetry of a specific device or family of devices.

The tone mapping curve can be seen on Figure 2.11.

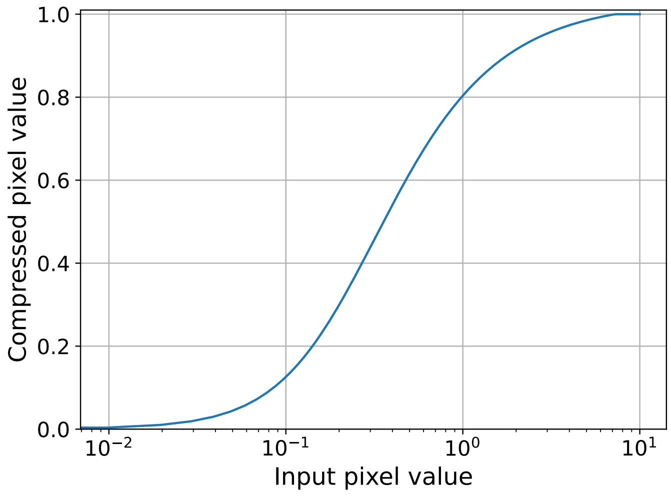


Figure 2.11: Graph of *Krzysztof Narkowicz's* ACES filmic tone mapping operator. For better visibility X-axis is in *log* space.

2.3 Exposure and automatic eye adaptation

Many global tone mapping algorithms map the input values to a log-like curve, which, as a result, makes all intensely bright values to be "squeezed" into a narrow range close to 1 (where 1 is the maximum displayable luminance). This ensures that the HDR values are mapped to an SDR range of e.g. *0* to *1*, but if the image contains big regions of extremely bright values the details are still lost, or at best the contrast is lacking.

One solution to this problem is rather simple and lies in shifting the original values by some chosen offset. But, because in a High Dynamic Range the values it allows for can vary drastically (typically only limited by numeric precision), choosing a certain "raw" offset can be tedious or unintuitive. For this reason, it is useful to look at the concept of *exposure*.

2.3.1 Exposure

The term *exposure* is a base-2 logarithmic scale that comes from the domain of photography, where it was originally used to determine how much a brightness must be increased or decreased to reveal the details of a scene. Similarly as decibel became a unit for measuring sound pressure, exposure has become a unit of measurement for luminance.

[9]https://en.wikipedia.org/wiki/Academy_Color_Encoding_System

In photography exposure is known as a function of *shutter speed* [10] and *aperture* [11] which are camera settings that essentially determine the amount of light that gets captured by the camera.

Thus, the exposure value can be defined as:

$$EV = log_2 \left(\frac{N^2}{t} \right) \tag{2.7}$$

where N is the aperture and t is shutter speed.

But in case where a realistic camera implementation is redundant, this equation may be of no use.

However, unless implementing a physically based camera system ([15]), where various real camera settings are used to determine the brightness of a scene, this equation is not needed.

Because exposure uses a logarithmic scale with a base of 2, each step in *EV (Exposure Value)* corresponds to a doubling or halving of luminance. For example, +1 EV is twice as bright as 0 EV, and +2 EV is four times as bright.

One of the advantages of using EV is that it aligns with our perceptual response to light, which also has a logarithmic nature. This essentially means that the difference between +1 and +2 EV appears the same as the difference between 0 and +1 EV to our eyes.

An EV of 0 represents a luminance of 0.125 cd/m^2, which is not zero light but rather the level of ambient light in a dimly lit room. When a scene is darker than that, its luminance can be represented with negative EV values (e.g. -1 EV, -2 EV, etc.). A schematic illustration of exposure values can be seen on Figure 2.12.

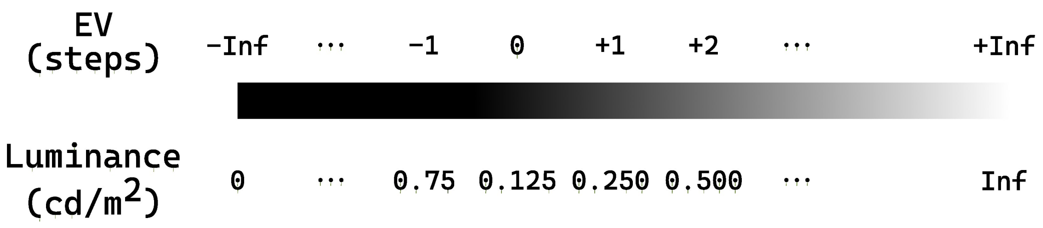


Figure 2.12: Comparison of Exposure Value and Luminance scale.

Another benefit of using exposure values when interacting with HDR colors is the convenience of representing a wide range of luminances. Editing light values in EVs is way more practical than, for example, values between 1 and 1, 000, 000.

This subsection has been adopted from [28] and [9].

2.3.2 Dynamic exposure

We can make use of exposure to alternate the brightness of an image to allow for a more uniform mapping of color values to SDR. But in real-time applications, it is not possible to manually set the exposure value every frame, so there is a need for some automatic way of setting exposure.

The challenge lies in defining what exposure settings are optimal for the current frame. Choosing to prioritize sunlight, shadows, or trying to find a balance in between. This is why

[10] *Shutter speed* is the length of time that the shutter of a camera remains open, allowing light to pass through the lens and onto the camera's sensor or film.

[11] *Aperture* is the opening in a camera lens through which light passes to reach the camera's sensor or film.

sometimes more explicit exposure settings over automatic ones are preferred. It is indeed possible to set exposure explicitly even in dynamic lighting conditions by using some kind of triggers or *post-process volumes* [12] placed around the scene. But for example, in computer games with dynamic levels, expansive open worlds, significant lighting variations, or situations where time constraints prevent manual tweaking of exposure volumes an automatic exposure adaptation mechanism is more suitable and easier to work with [23].

An appealing automatic exposure behavior can be achieved by mimicking the human visual system. The process of human eye adaptation can be modeled using an exponential decay function (viz. Figure 2.13):

$$L_{new} = L + (L_{avg} - L) \cdot (1 - e^{-\frac{T}{\tau}}) \tag{2.8}$$

Where L_{new} is the new pixel luminance, L is luminance in the previous frame, L_{avg} is the average luminance, T is the discrete time step between frames, and τ is the constant describing the speed of the adaptation [20].

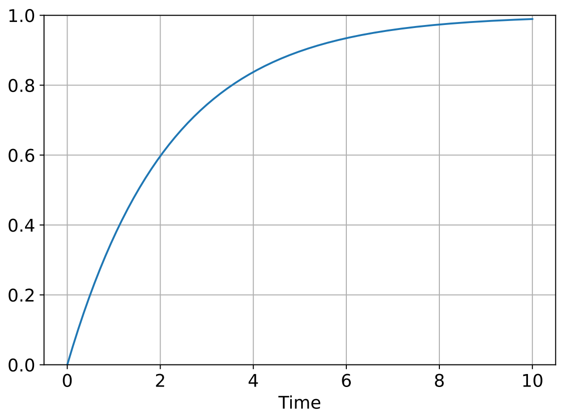


Figure 2.13: Exponential decay function $1 - e^{-\frac{x}{\tau}}$ with $\tau = 2.2$.

The key term of the automatic exposure equation is the average luminance. It is usually calculated as a *geometric mean*, which is equivalent to calculating the arithmetic average of a logarithmic luminance. As described in [13], to calculate the average luminance there are generally two approaches.

- *Luminance downsampling*: successively downsampling the frame image until a 1×1 log luminance buffer is obtained.
- *Luminance histogram*: creating a luminance histogram with a certain resolution (number of bins) from the frame image.

According to *Alex Tardif* ([32]), the main problem with using downsampling to find the average luminance is that extremely bright and dark pixels have a disproportionate impact on the resulting average. These pixels can heavily influence the downsampling results, which can negatively affect the quality of tone mapping. While this technique may work adequately in many cases and the results can be limited to a desired range, when it fails, the flaws become apparent and the visual outcome is unappealing.

[12] *Post-process volumes* are 3D shapes (e.g. boxes or spheres) used in computer graphics to apply specific visual effects or alterations to a scene or portion of a scene.

Further in this chapter, only the histogram approach will be discussed, as this is the one that I ended up implementing.

Following is the pseudocode algorithm of the automatic exposure using luminance histogram. I have summarized the algorithm in 3 distinct steps based on the articles by *Bruno Opsenica* [24] and *Krzysztof Narkowicz* [23].

First step is demonstrated in Algorithm 1. It is executed over each pixel in the image. Histogram $Hist$ is an array of a size R. Each bin of the histogram is filled with number of pixels that have the corresponding luminance value. It is important to note though that logarithmic luminance is used, as it closely resembles EV steps, described earlier 2.3. There is also L_{min} and L_{max} values that specify the EV range that is covered by the histogram resolution.

Algorithm 1: FILL LUMINANCE HISTOGRAM

Input: RGB values of original HDR image, empty histogram $Hist$, histogram resolution R, min log luminance L_{min}, max log luminance L_{max}, value close to zero ϵ

Output: Filled histogram $Hist$

```
1: $L = dot(RGB, 0.2125, 0.7154, 0.0721)$             # Calculate luminance
2: if $L < \epsilon$
3:     $Bin = 0$                                       # Lum close to zero gets into bin 0
4: else
5:     $L_{log} = clamp(\frac{log_2(L) - L_{min}}{L_{max} - L_{min}}, 0, 1)$   # Get log luminance into range 0 - 1
6:     $Bin = L_{log} \cdot (R - 1) + 1$               # Map range 0 - 1 to histogram resolution
7: $Hist[Bin] += 1$                                    # Increment the corresponding bin
8: return $Hist$
```

Next step is the Algorithm 2 which is calculating the average luminance. After the histogram is filled, finding the average luminance is as easy as calculating arithmetic average of the bins.

Algorithm 2: CALCULATE AVERAGE LUMINANCE

Input: Filled histogram $Hist$, histogram resolution R, min log luminance L_{min}, max log luminance L_{max}

Output: Average luminance L_{avg}

```
1: $Sum = 0$                                           # Initialize the sum
2: $N = 0$                                             # Initialize the total number of pixels
3: for $i$ = 0 to R-1
4:     $Sum += Hist[i] \cdot i$
5:     $N += 1$
6: $A_{log} = \frac{Sum}{N}$                           # Calculate arithmetic average of bin indices
7: $AL_{log} = \frac{A_{log} \cdot (L_{max} - L_{min})}{R - 1}$   # Map bin index to log lum
8: $L_{avg} = 2^{AL_{log}}$                            # Convert average luminance to linear
9: return $L_{avg}$
```

Last step deserves some particular attention. Algorithm 3 starts with applying temporal adaptation to the average luminance. To make exposure change gradually, thus simulating the human visual system behavior, the aforementioned Equation 2.8 is calculated.

Then on the line 2 the image key is calculated according to empiric formula presented by Krawczyk et al. [20]. The key value and average luminance are further used to calculate exposure coefficient that every pixel of the image is multiplied by.

Algorithm 3: Apply exposure adaptation

```
Input: $RGB$ values of original HDR image, average luminance of current frame
       $L_{avg}$, average luminance of previous frame $LP_{avg}$, time passed since last
       frame $\Delta t$, adaptation speed $\tau$
Output: $R'G'B'$ values of exposed image
1: $A = L_{avg} + (L_{avg} - LP_{avg}) \cdot e^{\Delta t \cdot \tau}$          # Temporal adaptation
2: $Key = \frac{1.03 - 2}{log(A+1)+2}$                            # Get empiric image key
3: $E = \frac{Key}{A}$                                            # Calculate exposure coefficient
4: $R'G'B' = RGB \cdot E$                                         # Apply exposure
5: return $R'G'B'$
```

2.4 Local tone mapping using bilateral filter

Throughout my studies of local tone mapping operators I have found an operator that is efficient and easy to implement. It was initially introduced in the year 2002 by *Frédo Durand* and *Julie Dorsey* [11] from Laboratory for Computer Science, Massachusetts Institute of Technology. As mentioned in the original paper this operator's purpose is to preserve as much detail as possible but at the cost of reducing overall image contrast. Its central idea is using the *bilateral filter* to split the image into two layers: *base layer* and *detail layer*, perform certain adjustments and combine the layers together.

But before diving into details it is important to describe what a *Gaussian filter* is, because bilateral filter implementation is based on it.

2.4.1 Gaussian filter

The *Gaussian filter* (or Gaussian blur) is a type of image-blurring filter that utilizes a Gaussian function, which also represents the normal distribution in statistics, to calculate the transformation for each pixel in the image. Values from this distribution are used to construct a *convolution matrix* that is then applied to the original image.

Each pixel's new value is determined by taking a weighted average of its surrounding pixels. The original pixel's value carries the highest weight, as it is at the peak of Gaussian function, while neighboring pixels receive smaller weights as their distance from the original pixel increases. Due to the use of Gaussian distribution, Gaussian blur achieves a blur effect that preserves boundaries and edges better than, for example, the *box* filter [13].

To calculate the coefficients for the Gaussian blur kernel typically the following formula is used, which corresponds to a two-dimensional Gaussian function (Equation 2.9):

[13] *Box filter (box blur)* is a primitive filter with a square or rectangular kernel used to blur an image.

$$G(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{2.9}$$

Where x and y are the distances from the origin (at (0,0)) in the horizontal and vertical axes respectively, and σ is the standard deviation of Gaussian distribution.

Equation 2.10 is an example of a 5×5 Gaussian kernel computed with *spacial sigma* (σ) value of 1:

$$\begin{bmatrix} 0.0029 & 0.0131 & 0.0215 & 0.0131 & 0.0029 \\ 0.0131 & 0.0585 & 0.0965 & 0.0585 & 0.0131 \\ 0.0215 & 0.0965 & 0.1592 & 0.0965 & 0.0215 \\ 0.0131 & 0.0585 & 0.0965 & 0.0585 & 0.0131 \\ 0.0029 & 0.0131 & 0.0215 & 0.0131 & 0.0029 \end{bmatrix} \tag{2.10}$$

It is worth noting that the Gaussian blur is a separable filter and thus, can be applied by sequentially convolving with two separate kernels (horizontal and vertical).

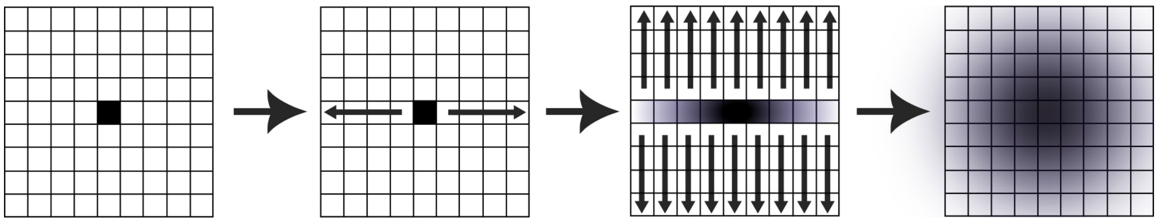


Figure 2.14: Schematic illustration of Gaussian blur executed with a split kernel.

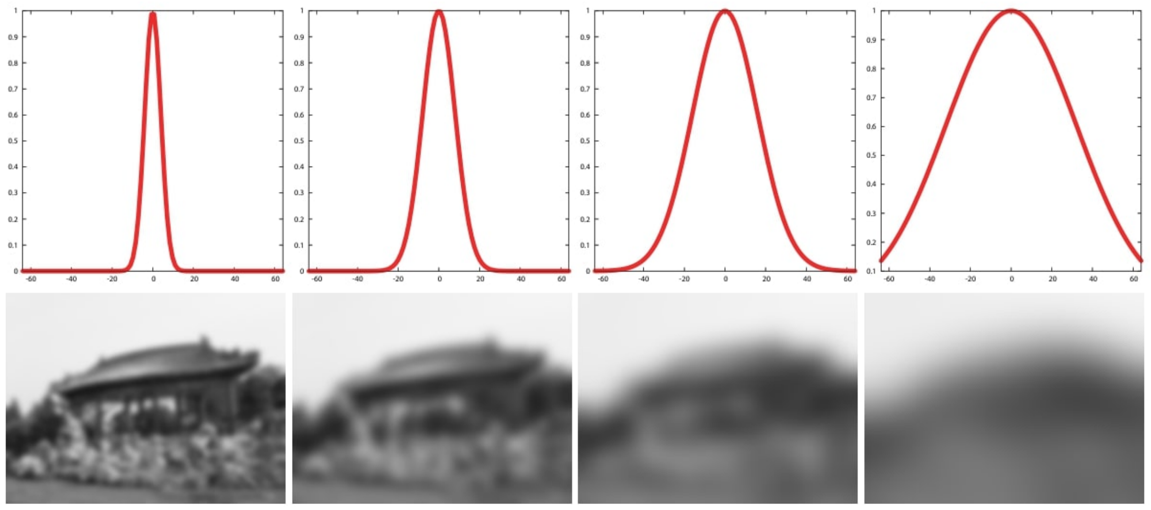


Figure 2.15: Example of Gaussian filtering with different σ (left to right: 4, 8, 16, 32). Top row shows the Gaussian kernel function and bottom row the result obtained by the corresponding Gaussian blur filtering. With high values of σ edges are lost because of pixel averaging.

Images taken from [26].

This subsection is based on the following sources: [26] and [38].

2.4.2 Bilateral filter

Gaussian filter is enough to achieve a uniform blur over the whole image, which can be useful in many cases. But if it is also necessary to preserve certain details, some kind of *edge-aware*[14] approach would be more suitable. Bilateral filter was initially developed for these purposes.

Bilateral filter was first introduced by Tomasi et al. in 1998 [33]. It blurs the image similarly to Gaussian blur but preserves much more details. Detail preservation is achieved because bilateral filter operates not just on spacial difference of surrounding pixels but also on their colors. Color difference, though, is multiplied by a negative weight which results in blurring that is weaker at the edges where this difference is big.

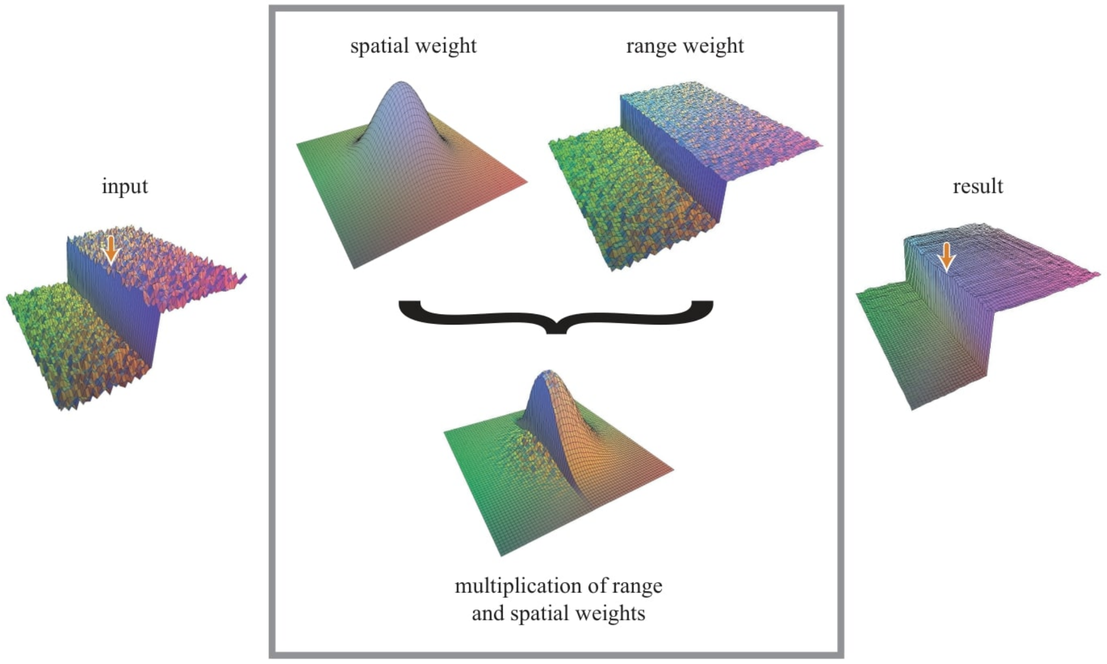


Figure 2.16: Schematic illustration of a Bilateral filter.
Image taken from [26]

Both the spatial and range kernel are often calculated using a Gaussian filter.
To compute the coefficients of a kernel for Bilateral filter this formula can be used:

$$B(x,y) = \exp\left(-\frac{x^2+y^2}{2\sigma_d^2} - \frac{\|I_c - I(x,y)\|^2}{2\sigma_r^2}\right) \tag{2.11}$$

Where

- $B(x,y)$ is the coefficient at position x and y in kernel.
- Ic is the value of a pixel located at the kernel center.
- $I(x,y)$ is the value of the current pixel.

[14]An *edge* in this context means a certain region of the image, where color varies rapidly (i.e. goes from bright to dark across the distance of only several pixels).

- σ_d is a spatial *sigma* [15] that essentially is an inverse weight for the distance from pixel (x, y) to center.
- σ_r is a range sigma that acts as an inverse weight for intensity difference between pixel $I(x, y)$ and Ic.

Then each coefficient must be normalized by dividing with the total sum of the coefficients.

$$B(x,y)^{\text{norm}} = \frac{B(x,y)}{\sum_{x,y} B(x,y)} \tag{2.12}$$

It is probably worth mentioning that bilateral filter is not a convolution, so its kernel cannot be separated into two like Gaussian one (viz. Figure 2.14), which makes it significantly less efficient. However, an optimization for bilateral filter exists which was also presented in the following paper [11].

2.4.3 The tone mapping operator

As was mentioned at the beginning of this section the idea of the tone mapping operator proposed by Durand and Dorsey is based on splitting the image into low and high frequency information. To avoid any confusion, the term *frequency* here can be understood as *the ratio of pixel intensity change to the change of pixel coordinates.*

The approach of splitting the image into low and high frequency was earlier taken by *Chiu et al.* in 1993 [5]. Their work was inspired by what is known in photography and image processing as *unsharp masking* [16]. They tried blurring the image with several filters including a Gaussian one. Then this blurred image was inverted and multiplied with initial image. However, the result was far from perfect, because of the clear halos that were visible around bright regions. To mitigate the halo problem, an edge-preserving filter, like Bilateral filter can be used instead of Gaussian filter.

I have summarized the algorithm by Durand and Dorsey in the form of the pseudocode presented in Algorithm 4. Individual algorithm's steps were to some extent based on [18].

In step 3 and step 8 value of 255 is used to expand the $0-1$ luminance range so that we can get rid of fractional values without loosing too much of the darker tones. It is important to get rid of fractions to avoid getting negative values out of logarithm.

In the original paper it is mentioned that calculations were done on log luminance values because it then corresponds directly to contrast and apparently because human visual system response to light is logarithmic in its nature.

On the Figure 2.17 there are three layers of initial image. Low- and high-frequency layers were obtained by splitting the image using the bilateral filter. The algorithm used to obtain these images is not exactly known, but should be similar to what I have summarized in Algorithm 4

The result of the operator can be seen on Figure 2.18. If we compare the output of Chiu et al. operator published in 1993 and Durand and Dorsey operator from 2002 we can see that the latter has almost no visible halos around shapes. As already mentioned this was achieved mainly due to making use of the bilateral filter.

[15] σ is the *standard deviation* of the Gaussian distribution.

[16] *Unsharp masking* is a popular image sharpening technique, that enhances contrast by subtracting the blurred version of the image from the original and then adding the difference back to the original image.

Algorithm 4: Durand & Dorsey local tone mapping

Input: *RGB* values of original HDR image, base offset o and scale s
Output: *R'G'B'* values of tone-mapped image

```
1: Compute luminance: $L = RGB \cdot (0.2125, 0.7154, 0.0721)$
2: Compute chrominance: $C = \frac{RGB}{L}$
3: Convert luminance to log space: $L = log_2(L \cdot 255 + 1)$
4: Compute low frequency (base): $B = Bilateral(L)$
5: Compute high frequency (detail): $D = L - B$
6: Modify base: $B' = (B - o) \cdot s$
7: Compute new luminance: $L' = B' + D$
8: Convert new luminance back to linear space: $L' = \frac{(2^{L'} - 1)}{255}$
9: Compute tone-mapped image: $R'G'B' = L' \cdot C$
10: return $R'G'B'$
```

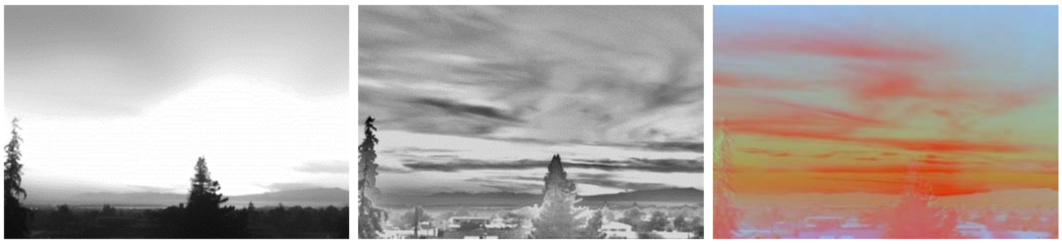


Figure 2.17: From left to right: *low-frequency layer*, *high-frequency layer* and *color layer*. Images taken from [12].

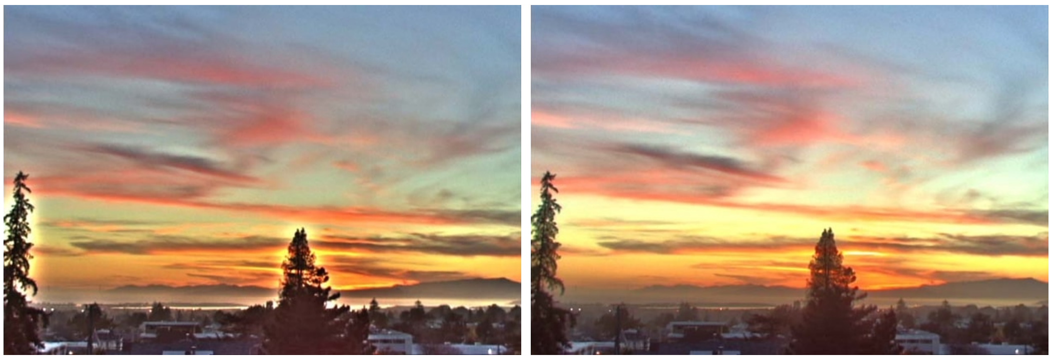


Figure 2.18: Comparison of *Chiu et al. 1993* operator result (left) vs. *Durand and Dorsey 2002* (right).
Images taken from [12].

2.5 Fusing differently exposed images into one

There is a particularly interesting technique that I have found throughout my studies, which is named *Exposure Fusion*. I have decided to implement it, so it is important to first describe it in detail in this chapter.

Exposure Fusion was originally proposed by *Mertens et al.* in their paper [21] as an alternative method for creating an SDR image using a *bracketed exposure sequence*[17].

The main idea of this technique is to fuse (combine) several differently exposed versions of the same image into the final one, that would contain as many details as possible. The images are combined on a per-pixel basis, which means that a certain metric is used to decide from which version of the image the pixel value would be picked. Typically, 3 to 4 differently exposed versions are enough to achieve good result. These images can be obtained by taking photos of a real scene with different exposure or in case of rendering an artificial scene by setting exposure manually or defining some rule according to which it will be set for every image.

To clarify, the details are preserved because e.g. for the shadows zone we pick the pixels from the highly exposed image, while in case of the highlights part the pixels from underexposed version are preferred. This way we are left with the image where all the details are clearly visible, no matter in which brightness region they are located. This process of "picking" the right pixel is controlled by the weight map. The weight for every pixel is based on three metrics: *contrast*, *saturation* and so-called *"exposedness"* (viz. Algorithm 5).

But as mentioned in original paper simply blending the images based on per-pixel weight does not produce plausible results. Disturbing seams appear when weights' variation is quick. It could be possible to smooth the sharp weight map with the help of a Gaussian filter, but this results in undesirable halos, and spills the information across object boundaries [21].

Further, in the paper a more sophisticated approach was proposed which is to blend the images separately on different frequency levels. This is achieved by utilizing a structure called *Gaussian pyramid*, which is a *mipmap* [18], where each subsequent mip level contains a blended version of a previous one. Then a so-called *Laplacian pyramid* structure is used to separate out the details of different frequencies from the images. And only then the fusion process begins in the form of blending the Laplacian pyramids separately on each level. This final Laplacian pyramid which arose from combining all Laplacian pyramids together based on the weight maps is then collapsed (summed up) in a certain way resulting in the final (fused) image.

For schematic illustration of the technique see Figure 2.19. The individual steps of the technique can be summarized like that:

1. Create several differently exposed versions of the image.

2. Compute weight maps for each of them based on *"exposedness"*, *saturation* and *contrast*.

[17] *Bracketed exposure sequence* is a series of photographs taken at different exposure settings, typically varying the exposure time and aperture.

[18] *Mipmap* is a set of precomputed, progressively smaller versions of an original texture image. Each level in the mipmap has half the resolution of the previous level. Typically used to improve rendering quality and performance for objects that appear at varying distances from the camera.

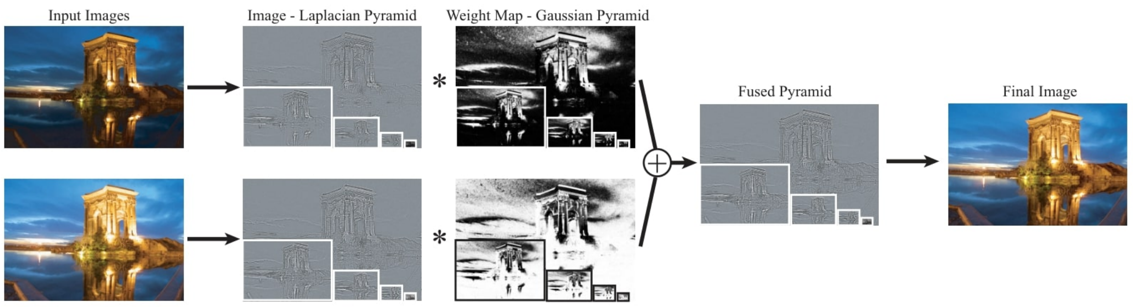
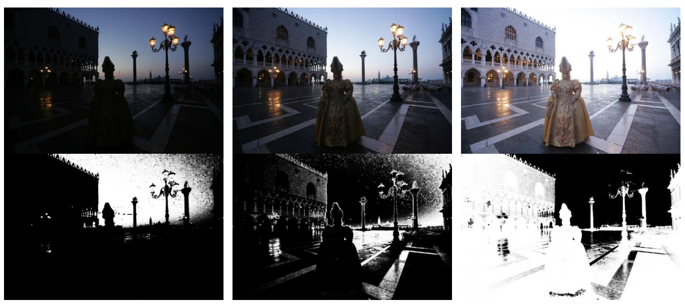


Figure 2.19: Schematic illustration of Exposure Fusion technique from original paper. Image courtesy of Mertens et al. [21].

3. Sequentially blur and downsample each image and its weight map to obtain a *Gaussian pyramid* for the image and its weight map separately.
4. For each image: substract a lower level of image's Gaussian pyramid from higher one to obtain every level of the corresponding *Laplacian pyramid*.
5. Blend Laplacian pyramids of all images into one on per-level basis based on the corresponding Gaussian pyramids of weight maps.
6. *Collapse* (sum up) the obtained blended Laplacian pyramid to get the final image.

The results of this technique as presented in original paper can be seen on Figure 2.20.


(a) Differently-exposed images and their weights.

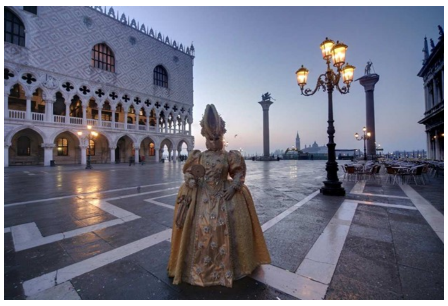

(b) Fused image.

Figure 2.20: Results of Exposure Fusion technique as demonstrated in original paper. Image courtesy of Mertens et al. [21].

I have summarized this technique in the form of pseudocode in Algorithm 5 and Algorithm 6. The equations and individual steps of this summary were based on an article by *Charles Hessel* [16], where you find a deep dive into Exposure Fusion implementation details with various explanation and all necessary equations.

Algorithm 5: Exposure Fusion: functions

```
function contrast($i$)
    foreach $p$ in $i$
        $p = p \cdot (0.2125, 0.7154, 0.0721)$                /* convert to grayscale */
    return $i * K_{Laplacian}$    /* convolve with laplacian kernel (2.13) */

function saturation($p$)
    foreach $p$ in $i$
        $p = \sqrt{\frac{1}{3}\sum_{c'=1}^{3}\left(p_{c'} - \frac{1}{3}\sum_{c=1}^{3} p_c\right)^2}$          /* standard deviation */
    return $i$

function exposedness($p, \sigma$)
    foreach $p$ in $i$
        $p = \prod_{c=1}^{3} \exp\left(\frac{-(p_c-0.5)^2}{2\sigma^2}\right)$       /* gaussian distance from gray */
    return $i$

function downsample($i_{bigger}$)
    $i_{bigger} = i_{bigger} * K_{Burt\&Adelson}$          /* kernel from Equation 2.14 */
    for $h$ = 0 to height($i_{bigger}$)/2
        for $w$ = 0 to width($i_{bigger}$)/2
            $i[h, w] = i_{bigger}[2h, 2w]$           /* leave every second pixel */
    return $i$

function upsample($i_{smaller}$)
    for $h$ = 0 to height($i_{smaller}$)
        for $w$ = 0 to width($W_{smaller}$)
            $i[2h, 2w] = 4 \times i_{smaller}[h, w]$   /* multiply by 4 to normalize */
    $i = i * K_{Burt\&Adelson}$                        /* kernel from Equation 2.14 */
    return $i$

function gaussianPyramid($i$)
    $G^0 = i$                          /* level zero is the original image */
    for $l$ = 1 to $N_{levels} - 1$
        $G^l$ = downsample($G^{l-1}$)
    return $G$
```

Algorithm 6: Exposure fusion

Input:

Output:

```
 1: for $i$ = 0 to $N_{images}$
 2:     $W_i = \texttt{contrast}(I_i)^{w_c} \times \texttt{saturation}(I_i)^{w_s} \times \texttt{exposedness}(I_i)^{w_e}$
 3: for $i$ = 0 to $N_{images}$
 4:     $W_i = \frac{W_i}{\sum_{i'=1}^{N} W_{i'}}$                    /* normalize weight */
 5:     $GI_i = \texttt{gaussianPyramid}(I_i)$          /* gaussian pyramid of image */
 6:     $GW_i = \texttt{gaussianPyramid}(W_i)$         /* gaussian pyramid of weight */
 7:     $L_i^{N_{levels}-1} = GI_i^{N_{levels}-1}$                 /* highest laplacian mip */
 8:     for $l$ = 0 to $N_{levels} - 2$
 9:         $L_i^l = GI_i^l - \texttt{upsample}(GI_i^{l+1})$        /* upsample and subtract */
10: for $l$ = 0 to $N_{levels} - 1$
11:     for $i$ = 0 to $N_{images}$
12:         $L_{blended}^l \mathrel{+}= L_i^l \times GW_i^l$              /* apply weight and add */
13: for $l = N_{levels} - 2$ to 0
14:     $L_{blended}^l \mathrel{+}= \texttt{upsample}(L_{blended}^{l+1})$         /* collapse pyramid */
15: $I_{final} = L_{blended}^0$                        /* the sum remains in level zero */
16: return $I_{final}$
```

$$K_{Laplacian} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.13}$$

$$K_{Burt\&Adelson} = \begin{bmatrix} 0.0025 & 0.0125 & 0.02 & 0.0125 & 0.0025 \\ 0.0125 & 0.0625 & 0.1 & 0.0625 & 0.0125 \\ 0.02 & 0.1 & 0.16 & 0.1 & 0.02 \\ 0.0125 & 0.0625 & 0.1 & 0.0625 & 0.0125 \\ 0.0025 & 0.0125 & 0.02 & 0.0125 & 0.0025 \end{bmatrix} \tag{2.14}$$

2.6 Bloom

Bloom is a technique used to enhance the perception of brightness in visual content. It overcomes the limitations of monitors' intensity range by creating a glowing effect around bright light sources and illuminated areas. This effect gives viewers the illusion of intense brightness, enhances the lighting in a scene, and makes the scene feel more dramatic.

This post-processing effect works best when combined with HDR rendering. Indeed, bloom and HDR are so closely related that in people's minds for many years they were often perceived as the same thing. Although it is possible to implement bloom with standard SDR 8-bit precision framebuffers, HDR rendering greatly enhances its effectiveness [34].

The main reason why a bloom effect can give the scene such a vibrant and realistic look is because this effect naturally occurs everywhere in our world. There are two known reasons for why it appears both in biological and digital media [19]:

- Appearance of bloom in our eyes stems from the inherent limitation of real-world lenses to achieve perfect focus. Even an ideal lens blends incoming images with the so-called *Airy disk*, a diffraction pattern from light passing through a circular aperture of a pupil. These imperfections are usually unnoticeable, but intense light sources reveal them, causing the bright light to extend unnaturally.
- In the case of digital cameras bloom arises from an excess of charge within *photodiodes* – light-sensitive components in the camera's image sensor. When intense light hits a photodiode, the stored charge can spill over to neighboring pixels, forming a halo-like outcome known as "charge bleeding."

In computer graphics, the effect of bloom is often simulated by utilizing Gaussian blur (viz. Section 2.4.1) in some way (as will be demonstrated further).

2.6.1 Naive implementation

One way to implement bloom is to render the lit scene as usual and extract two images: the HDR image itself and the image(texture) containing only the highlights (very bright regions of the image). The highlights texture is then blurred, multiplied with some constant and combined with original HDR scene image. *Joey de Vries* describes this *naive* bloom effect implementation in his article on *LearnOpenGL* [20] [34]. I have summarized this approach in the form of pseudocode in Algorithm 7.

Algorithm 7: NAIVE BLOOM ALGORITHM

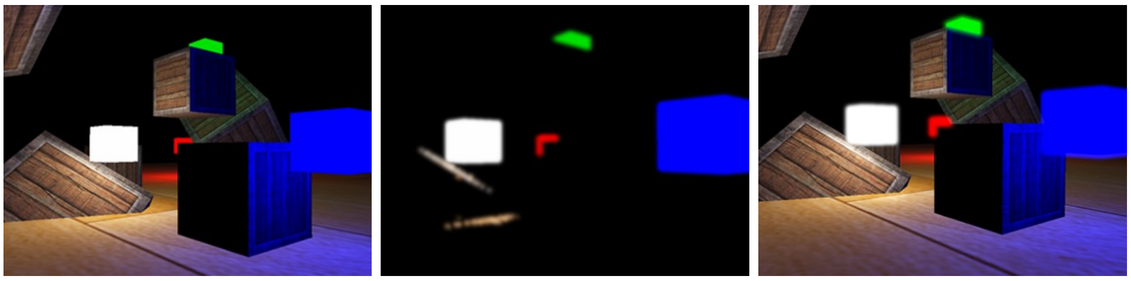
Input: *RGB* values of original HDR image, threshold t, bloom weight w
Output: *R'G'B'* values of resulting image

```
1: Compute luminance: $L = RGB \cdot (0.2125, 0.7154, 0.0721)$
2: Apply luminance threshold: if $L >= t$ then $B = RGB$ else $B = (0, 0, 0)$
3: Apply Gaussian blur to thresholded image: $B = Gaussian(B)$
4: Combine bloom and original image: $R'G'B' = RGB + B \cdot w$
5: return $R'G'B'$
```

As we can see from Figure 2.21 the results of this simple approach are not realistic enough. Just one blur pass for the highlights generally does not produce plausible results. It would probably be possible to achieve better results with more blur passes with different radii (kernel sizes) the results of which would be mixed by individual weights. But more blur passes over the initial (potentially high) resolution is a big performance overhead. So a more sophisticated approach was developed.

[19] The following two points were adopted from https://en.wikipedia.org/wiki/Bloom_(shader_effect).

[20] *LearnOpenGL* is a popular online resource for learning computer graphics programming with a focus on OpenGL.



(a) Original image. (b) Thresholded and blurred image. (c) Final image with bloom applied.

Figure 2.21: Results of *naive* bloom implementation.
Image courtesy of *Joey de Vries* [34].

2.6.2 A more realistic approach

In the year 2014 at *SIGGRAPH* [21] a new approach for a more realistic bloom effect was presented by *Jorge Jimenez* [17]. The approach has a few key points:

- Original image is not thresholded. The benefit of this is that in a non-thresholded HDR image bloom will only be strong where light intensity is indeed high, but will still be, to some extent, present even in darkest regions, which is similar to how it is in the case of human perception.
- Bloom is computed on multiple mip levels. That means that the image will be blurred and then downsampled to a lower resolution (which can be considered as a single operation). The downsampling procedure is performed until the smallest mip level is filled. After that, starting with the lowest resolution the image is recursively upsampled and combined.

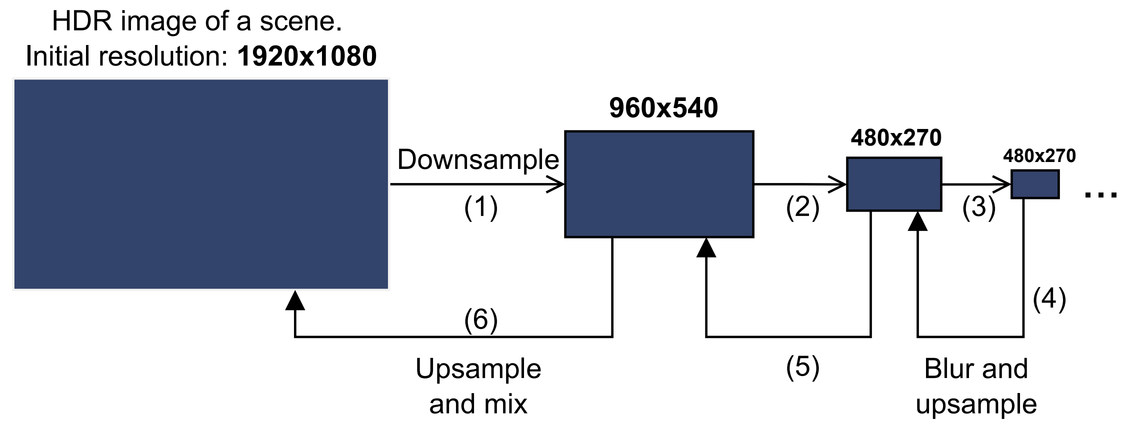


Figure 2.22: The process of downsampling and upsampling of the HDR image.

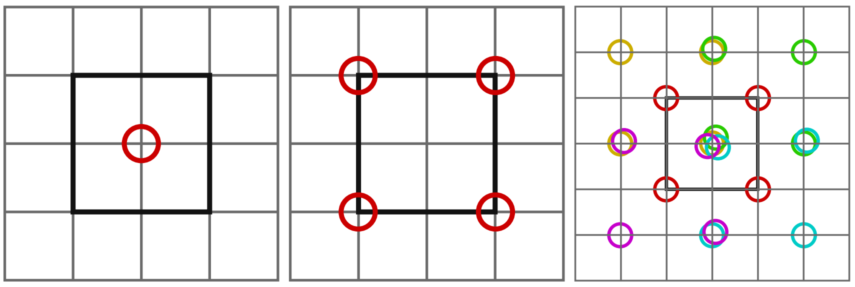
The *downsampling-upsampling* approach makes the bloom much smoother because blur being applied at a lower resolution effectively has a bigger radius, so in the end we get

[21] *SIGGRAPH* is a leading annual conference in computer graphics and interactive techniques, showcasing the latest advancements in these fields through presentations and discussions.

the image that is blurred at different scales. However, the use of mipmaps has its pitfalls because the lower the resolution goes, the *coarser* the blur becomes, so if the blur was initially not smooth enough or had an unsuitable kernel, *stair-step artifacts*[22] may become visible. In the aforementioned presentation, it's stated that simple *bilinear filtering* [23] for downsampling or upsampling produces low-quality results, specifically due to the presence of such artifacts.

So a better filtering method was proposed:

- For downsampling: take 13 bilinear samples around the current pixel and averaging them out based on specified weights. Given the colors in Figure 2.23c the weights are 0.125 for yellow, green, blue, and purple individually and 0.5 for red.



(a) One bilinear sample. *(Bad).*

(b) Four bilinear samples. *(Better but with some artifacts).*

(c) Thirteen bilinear samples. *(Good enough).*

Figure 2.23: Comparison of downsampling filters as shown in presentation by *Jorge Jimenez.*
Images based on [17].

- For upsampling: use a so-called *tent filter* with a 3×3 kernel (viz. Equation 2.15).

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.15}$$

I have summarized this approach in Algorithm 8.

Figure 2.24 demonstrates the result of this bloom technique implemented by *Alexander Christensen* [6].

[22] *Stair-step artifacts* are visual artifacts that appear as jagged or stair-stepped edges in computer-generated images, typically due to the problem, known as aliasing.

[23] *Bilinear filtering* is a filtering method that works by calculating the weighted average of the four surrounding texels, with closer texels contributing more to the final color value.

Algorithm 8: COD: Advanced Warfare bloom

Input: *RGB* values of original HDR image, bloom weight (intensity) *w*, bloom mipmap pyramid *Bloom*, number of mip levels *N*, downsampling filter *DFilter*, upsampling filter *UFilter*

Output: *R'G'B'* values of resulting image

```
1: Copy original image to mip level 0: $Bloom[0] = RGB$
2: Downsample: for $i = 1; i < N, ++i$
3:   | $Bloom[i] = DFilter(Bloom[i-1])$
4: Upsample and add: for $i = N-2; i >= 0, --i$
5:   | $Bloom[i+1] = Bloom[i+1] + UFilter(Bloom[i])$
6: Linearly interpolate between original image and bloom:
     $R'G'B' = mix(RGB, Bloom[0], w)$
7: return $R'G'B'$
```

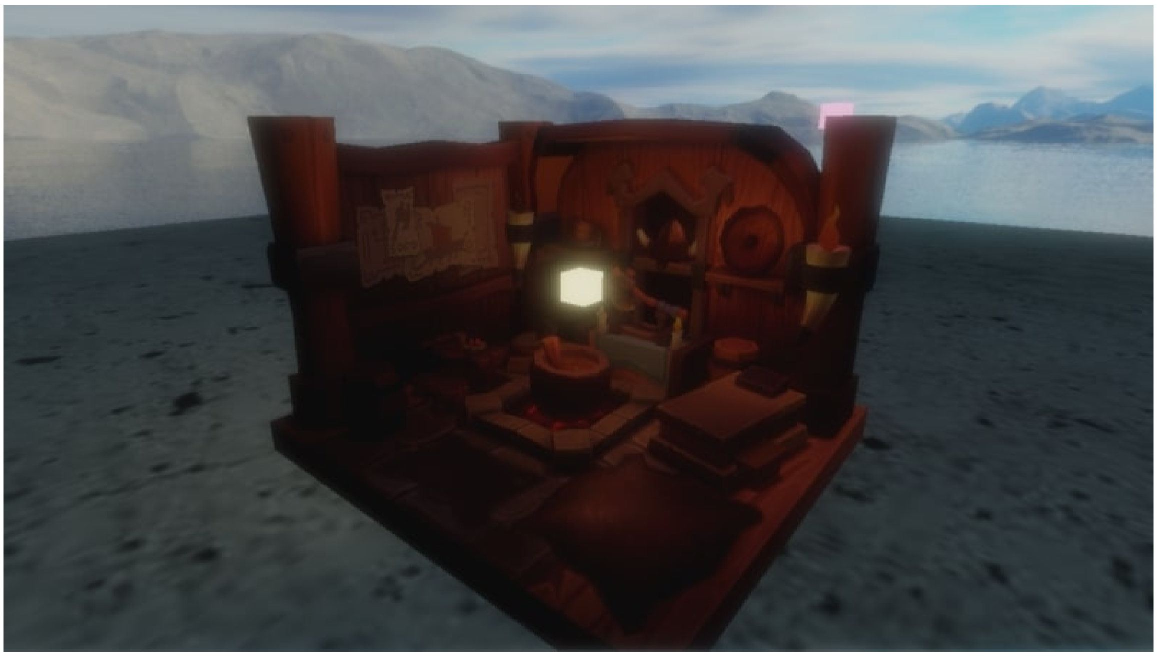


Figure 2.24: Result of a more advanced bloom implementation as presented by *Alexander Christensen*.
Image taken from [6].

Chapter 3

Implementation

The practical part of this thesis is a program [1] written in C++ programming language (version C++20).

- In the following sections I will refer to this program by the title `vulkan-hdr-demo`.

One of the key points of this thesis is to utilize Vulkan graphics API, so it is important to at least briefly introduce what is Vulkan API and what are its key differences compared to other APIs like DirectX or OpenGL.

3.1 Vulkan API

Earlier when graphic APIs like OpenGL were invented, did not provide such performance and such features as nowadays. This promoted the development of a state-based API with relatively simple interface like OpenGL. As the time passed by, interaction with OpenGL became restrictive and cumbersome when trying to achieve maximum performance or utilizing cutting-edge features. The industry was in need of a more modern and robust API that would account for all the new features and technologies. Thus, Vulkan API was invented in year 2016 by Khronos Group.

The idea behind Vulkan was to minimize driver overhead to allow for full control over processing power and memory resources. The key difference of Vulkan is that it shifts many of the functionalities and duties that were traditionally handled by drivers onto the application itself. This puts a lot more responsibility on the application developer, which has to explicitly implement different low-level concepts like memory management, synchronization, presentation of images to screen etc. [31].

The following sections do not aim to serve as a Vulkan API guide or tutorial, but I have tried to include in them a brief description of each Vulkan concept in question. For a detailed and thorough information about Vulkan API it is advised to follow various guides available on the web. Some of the resources that I have used to learn Vulkan API and to write Section 3.3 are [3], [37] and [25].

[1]The program `vulkan-hdr-demo` is available as a *GitHub* repository, located at: `https://github.com/VileDeg/vulkan-hdr-demo`.

3.2 Program structure

Although the program is written in C++ language its design is not exactly object-oriented. This was a conscious decision to avoid unnecessary complications when designing various abstractions because this project has been developed within a long period of time by a person who is new to Vulkan (me). Also, at the beginning the functionality of the final program was not known, so it would be close to impossible to come up with suitable abstractions to fit the future needs. The complexity of Vulkan API itself was also a significant reason not to choose a purely object-oriented approach.

The general `vulkan-hdr-demo` structure is as follows (also viz. Figure 3.1):

1. Creation and initialization of the window with GLFW [2] library.
2. Initialization of Vulkan API. This is a relatively long and complex part which deserves more attention. Section 3.3 describes in detail the initialization process.
3. Loading of a demo scene. Since the purpose of `vulkan-hdr-demo` is to demonstrate various HDR techniques by rendering a 3D scene, this part includes importing a 3D model with *diffuse textures* [3] and *bump maps* [4]. It also includes loading an HDR skybox texture which will allow to better demonstrate the implemented HDR rendering techniques. The process of scene loading is described in detail in section 3.4.
4. Initialization of user interface (UI). In this part *ImGui* [5] library is initialized, necessary font textures are loaded and registered, and also the necessary steps are done that will allow the rendered image to be correctly displayed in the viewport. The user interface itself is described with illustration in Section 3.7.
5. Main program loop. It consists of listening to user input and other window events, rendering of the scene frame-by-frame and updating the user interface. The process of scene rendering is described in detail in Section 3.5.
6. Program exit and correct cleanup of all allocated resources.

[2] *GLFW* is an open source, multi-platform windowing library. `https://www.glfw.org/`.
[3] *Diffuse texture* is a texture applied to a 3D model to give it just a base color without additional details.
[4] *Bump map* is a grayscale texture, that creates the illusion of depth by simulating surface details.
[5] *Dear ImGui* is a popular immediate-mode GUI library for C++. `https://github.com/ocornut/imgui`.

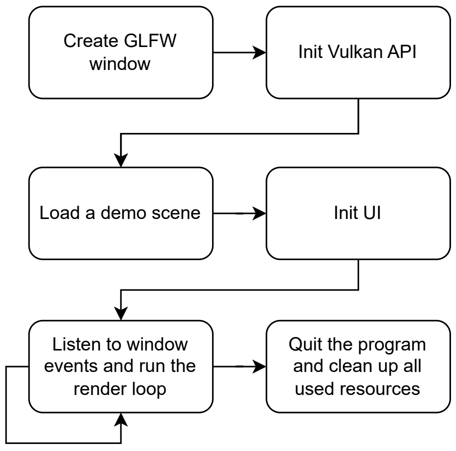

Create GLFW window
Init Vulkan API
Load a demo scene
Init UI
Listen to window events and run the render loop
Quit the program and clean up all used resources

Figure 3.1: Diagram of `vulkan-hdr-demo` structure.

3.3 Initialization of Vulkan API

Vulkan is an explicit API that requires a verbose initialization process which includes various steps such as: loading extensions, selecting the GPU and creating objects such as `VkInstance`, `VkDevice` etc. for further use withing Vulkan commands.

Unlike OpenGL, Vulkan does not have global state, so it is necessary to include relevant objects into each API function call. The initialization scheme on Figure 3.2 illustrates the process of initialization. Individual arrows indicate which objects are needed to create other objects.

At the start of the program `Engine::Init()` is called, which in term sequentially calls all the functions that create and initialize Vulkan API objects needed for rendering.

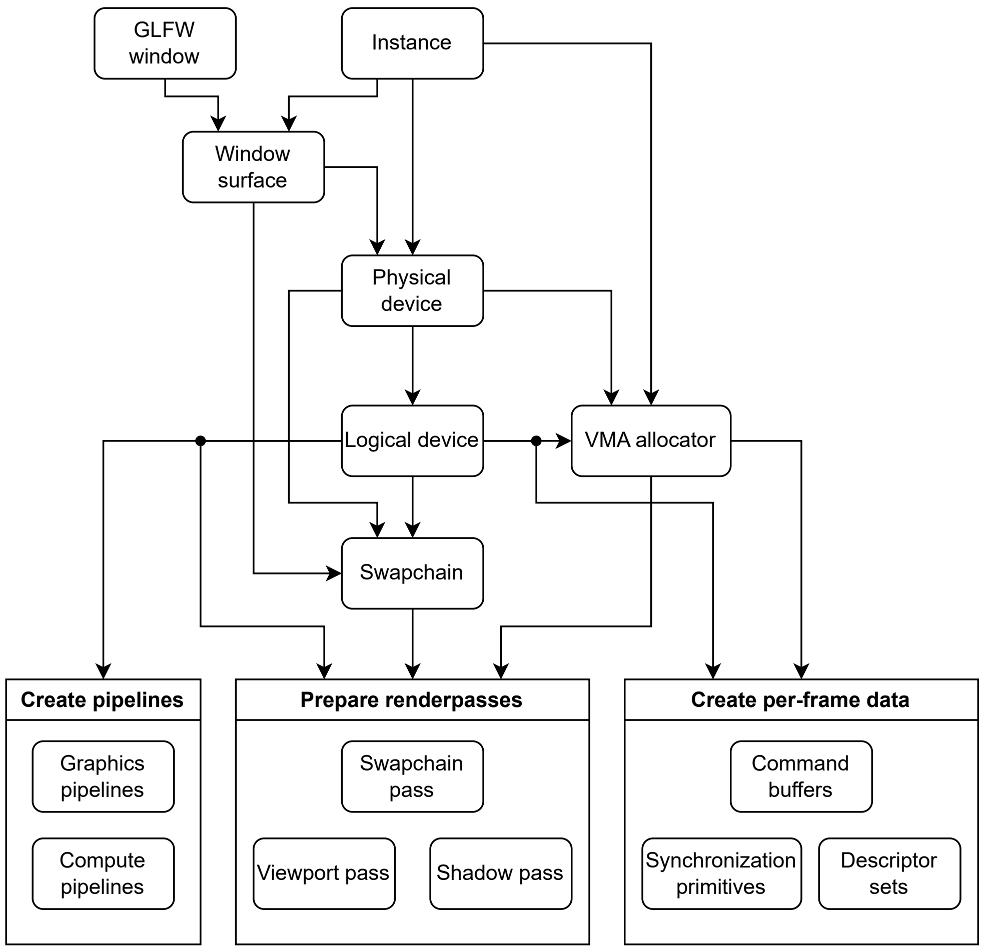


Figure 3.2: Diagram of `vulkan-hdr-demo` initialization.

3.3.1 Instance

At the core of Vulkan is the object named *instance* (`VkInstance`). It represents a Vulkan API context that holds all the global state. It allows to enable any necessary extensions (such as `VK_KHR_surface` which is needed to present images on screen).

After initializing GLFW window, `Engine::createInstance()` is called which creates a Vulkan instance, enables all necessary instance-level extensions and validation layers (if compiled in debug mode).

3.3.2 Physical device

Usually the next necessary step after creating instance is picking a physical device. Vulkan requires the user to explicitly choose a certain physical device (GPU) that will be used for all processing. `VkPhysicalDevice` is the object representing a GPU, and it allows the user to query its features, memory size, and available extensions.

Commands in Vulkan API need to always be sent to a *queue* which belongs to a certain *family* that is presented by a chosen device. So when choosing a GPU user should check whether the device has at least one queue family that supports required type of operations (e.g. graphics or compute).

`vulkan-hdr-demo` does not have any special requirement for a GPU, except the support for graphics and compute operations (`VK_QUEUE_GRAPHICS_BIT` and `VK_QUEUE_COMPUTE_-BIT`). Graphics operations involve rendering the scene through vertex and fragment shaders while compute operations are needed for all post-processing as it exclusively utilizes compute shaders. Most modern GPUs should have no problem supporting these types of operations.

`Engine::pickPhysicalDevice()` function queries the list of available GPUs and iterates through it to find the most suitable one. It ranks every device with a score based on different parameters like whether it is an integrated or dedicated GPU, whether it supports all required extensions and has the necessary queue families. Then it picks the best fitting device based on the score. This approach was borrowed from my supervisor's tutorial [27].

3.3.3 Logical device

Next step is creating a Vulkan object called a logical device (`VkDevice`), usually just referred to as "device". If a physical device is a representation of the actual GPU and its capabilities, the logical device can be considered an actual GPU driver along with its configurations. Most of Vulkan commands from this step on will take the `VkDevice` as an argument because they will relate to a specific device that was selected and initialized. The concept of logical device allows for use of multiple GPUs at once. For every GPU a device is created, and then it is possible for each device to have a specific subset of extensions enabled and process specific commands while communicating of sharing tasks with other devices.

`Engine::createLogicalDevice()` checks the support of all required device features and creates the logical device. From the newly created logical device the aforementioned queue families are fetched using `vkGetDeviceQueue` function.

3.3.4 Swapchain

Swapchain (`VkSwapchainKHR` [6]) is the Vulkan object that is necessary to present images to the screen. Presenting images is technically optional, because Vulkan can be used with

[6] *KHR* postfix indicated that the object is part of a Vulkan extension ratified by Khronos.

GPUs that do not have any output device connected, so swapchain is not a part of the core Vulkan API, instead it is part of a device extension `VK_KHR_swapchain`.

Swapchain functions as a queue of images awaiting display on the screen. Its primary objective is to coordinate the presentation of images with the screen's refresh rate. Application obtains one of the images to draw on it and subsequently places it back into the queue. The specific mechanics of the queue and the criteria for displaying an image vary based on the configuration of the swapchain. Generally the number of images in swapchain is set to be 2 or 3 for double-buffer or triple-buffer rendering respectively, however this number is limited by the minimum number supported by the GPU.

Function `Engine::createSwapchain()` queries the swapchain properties available for current device, chooses the surface image format, color space and present mode. This functioned tries to always pick the best and most suitable *image format & color space* pair from the list of supported ones. Thus, if the connected display device supports some HDR format, this format gets the priority over the SDR ones. Table 3.1 shows the list of *image format & color space* pairs that `vulkan-hdr-demo` supports. Priority drops from top to bottom.

VkFormat	VkColorSpaceKHR
VK_..._A2B10G10R10_UNORM_PACK32	**VK_..._HDR10_ST2084_EXT**
VK_..._UNDEFINED	**VK_..._DISPLAY_P3_NONLINEAR_EXT**
VK_..._UNDEFINED	**VK_..._DCI_P3_NONLINEAR_EXT**
...	
VK_..._R16G16B16A16_SFLOAT	**VK_..._EXTENDED_SRGB_LINEAR_EXT**
VK_..._A2B10G10R10_UNORM_PACK32	*VK_..._BT2020_LINEAR_EXT*
VK_..._R8G8B8A8_SRGB	VK_..._SRGB_NONLINEAR_KHR
...	

Table 3.1: Main supported swapchain *image format & color space* tuples in `vulkan-hdr-demo`.
HDR color spaces are in **bold**, SDR color spaces with wider gamut are *emphasized*.
`VK_FORMAT_UNDEFINED` means any format is accepted.

The present mode is chosen to be `VK_PRESENT_MODE_MAILBOX_KHR` [7]. It uses a list of images, and while one of them is being displayed on the screen, the scene will be continuously rendered to the others in the list. Whenever it is time to display an image, the most recent one is selected. This is the mode normally used for applications that use tripple-buffering which is true for `vulkan-hdr-demo`.

At the end of the function `vkGetSwapchainImagesKHR` is called to get the images (`VkImage`) from the swapchain. `VkImage` can be interpreted as a handle to an actual resource on the GPU, so it does not need to be created by the user. The acquired `VkImage` objects are later used to create `VkImageView` and `VkFramebuffer` objects needed for rendering.

One thing to mention is that because swapchain also creates the associated images, user needs to specify current window dimensions when creating it. For that reason it has to be recreated every time a window is resized. Because all window input in `vulkan-hdr-demo` is handled by `GLFW` library, the occurence of window resize is reported by a callback assigned via `glfwSetFramebufferSizeCallback()`. Once the window is resized

[7]More about presentation modes can be found in Khronos registry: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkPresentModeKHR.html.

and this callback gets executed it calls `Engine::recreateSwapchain()` function that frees the previous swapchain resources and creates a new one with updated window dimensions.

3.3.5 Renderpasses

At this point a renderpass needs to be created. Concept of a *renderpass* is exclusive to Vulkan. Its main purpose is to provide the driver with more information about the state of the images being rendered to allow for potential optimizations. A renderpass encapsulates the execution of graphics commands (compute ones can be executed without a renderpass). `VkRenderPass` object contains information about *attachments* (`VkAttachment`), which are basically the images being rendered to.

There is an object closely connected to a renderpass: a framebuffer (`VkFramebuffer`). The concept of a framebuffer as a rendering target is not a Vulkan innovation. Framebuffer is present in other APIs like OpenGL. But in Vulkan a framebuffer is created with regard to a specific renderpass which was generally the source of frequent questions from the user community.

Setting up renderpasses and framebuffers can be a tedious process, especially when there are multiple rendering steps. This is mainly because renderpass consits of subpasses which are responsible for image layout transitions and synchronization. This further complicates the logic of the program and to keep all the synchronization in order an application usually has no other way left than to use something like a *scene graph* [8] to automatically resolve all the necessary layout transitions and memory barriers operations when creating a renderpass.

These complications that are actually not guaranteed to bring any performance benefit on a regular (non-tiled) GPU had induced *Khronos Group* [9] to release the `VK_KHR_dynamic_rendering` extension that removes the need to use `VkRenderpass` and `VkFramebuffer` objects completely [10]. Furthermore, starting from *Vulkan 1.3* this extensions is now part of the *core* API.

Originally I was developing `vulkan-hdr-demo` without this extension but as soon as the program got more complicated in terms of synchronization I decided to enable this extension which allowed me to reduce and simplify the initialization code. So currently while recording command buffer *vkCmdBeginRenderingKHR* command is used to begin rendering which simply takes as input *VkImageView* objects.

3.3.6 Pipelines

The Vulkan pipeline object is a fundamental concept in Vulkan API that represents the complete GPU configuration for drawing and computational operations. It defines how graphics or compute data is processed and transformed by the GPU.

A `VkPipeline` object consists of several configuration structures that define various aspects of the rendering process, including shader stages, vertex input, rasterization settings, multisampling, color blending, and more.

Implementation of `Engine::createPipelines()` function that creates all graphics and compute pipelines is contained in `pipelines.cpp` module. There are three graphics pipelines:

[8] A scene graph is a hierarchical data structure commonly used to organize and manage the objects within a 3D scene by representing the relationship between objects and their transformations.

[9] *The Khronos Group, Inc.* is an open, non-profit, consortium developing and maintaining interoperability standards for computer graphics and machine learning. https://www.khronos.org/.

[10] Announcement of `VK_KHR_dynamic_rendering` in Khronos blog: https://www.khronos.org/blog/streamlining-render-passes.

for scene objects (main model and sphere model of light sources), for skybox and for *shadow pass* (rendering of shadows to the cubemap). Compute pipelines on the other hand are numerous. It is not even enough to create a single pipeline per a post-processing effect because most of them include several stages each of which is a separate shader. And sometimes even if the same shader is used for several stages, the input data bound through *descriptors* is different for each stage, so it still requires creating a separate pipeline.

The shaders are written in *GLSL (OpenGL Shading Language)*, which is a high-level shader language with a C-like syntax. The code is them compiled into *SPIR-V (Standard Portable Intermediate Representation)*, an intermediate language for graphics and parallel computing. The compilation is done using an open source command line tool maintained by Google named `glslc`. For convenience during development I created a python script that compiles all the shaders that are located in `assets/shaders/src` folder. This script is called every time a project is built. This is achieved through `CMake` [11] custom target dependency.

The compiled shaders for each pipeline are read as binary to create shader modules. All other configuration data structures are set as well and supplied for pipeline creation.

3.3.7 Descriptor sets

Descriptor sets define how resources are bound to shader stages during rendering and compute operations. They are organized by descriptor set layouts and allocated from a descriptor pool. Each descriptor set consists of descriptor bindings that represent specific resources accessed by a shader. During operations, a descriptor set is bound to a pipeline layout, allowing the shader to access the specified resources.

The function `Engine::createFrameData()` creates all descriptor sets along with command buffers and synchronization primitives. All these objects are created together in this function because a separate copy of them all is needed per each *frame in flight* [12], so in case of `vulkan-hdr-demo` 3 copies of them is created in this function. If there was only a single command buffer and single group of descriptor sets, program would need to wait until these resources are no longer in use thus disallowing to supply multiple frames for rendering at once. This is also true for *Uniform Buffers* [13] and *Shader Storage Buffers* [14] which are created in the same function. They are used in specific shaders and are thus supplied to specific descriptor sets.

Managing descriptor sets is a crucial part of any Vulkan application because this process gets incredibly complicated as soon as more shaders and pipelines get involved. Manually creating all descriptor sets is definitely far from optimal, so implementing some kind of automatization for this task is very important.

I borrowed the abstraction from Victor Blanco's *Vulkan Guide* [3], and it proved to be enough for the needs of `vulkan-hdr-demo`.

The interface of descriptor set creation is defined in `vk_descriptors.cpp` module. It includes 3 classes:

[11] *CMake* is a popular open source build automation software for building C and C++ applications.

[12] *Frame in flight* is a frame that is currently being rendered to or presented to the screen.

[13] *Uniform Buffer Object (UBO)* is a GPU buffer that stores read-only data, which is constant across multiple shader invocations.

[14] *Shader Storage Buffer Object (SSBO)* is a GPU buffer that allows a shader to perform both read and write operations with its data. As a result, the data may not remain constant across multiple invocations.

- `DescriptorAllocator` is the class that handles descriptor set allocation. It creates a new descriptor pool when the first set is created and then the subsequent sets are allocated from the same pool. When the pool has not sets left to allocate a new pool is automatically created. On cleanup the allocator destroys all the pools which also results in all the allocated sets being destroyed.
- `DescriptorLayoutCache` is the class that contains all the created descriptor layouts. Layouts are created and added to the cache by `create_descriptor_layout()` method. When the method is called, it searches through the `std::unordered_map` cache of layouts to check if there is one with the same bindings and returns it if there is one. Main purpose of this class is to avoid creating duplicate layouts.
- `DescriptorBuilder` is the central class of the abstraction. The process of creating a descriptor set comes down to calling the `Build` method which takes `DescriptorAllocator` and `DescriptorLayoutCache` classes as parameters. Then for every binding that is present in the shader either `bind_buffer()` or `bind_image()` is called to supply all the necessary data needed to create a `VkDescriptorSetLayoutBinding` and a `VkWriteDescriptorSet` and store them into lists. After that the creation process is finalized with a call to `build()` method which creates (or retrieves) a layout with `create_descriptor_layout()` using the supplied bindings data, allocates the set and updates it with supplied `VkWriteDescriptorSet` objects.

For the most part the code was taken as-is, only the two functions were added by me: `bind_image_empty()` used to postpone the image descriptor writing for later and `bind_image_array()` to bind a whole array of image descriptors at once, which is useful for mipmap pyramids used in a couple of post-processing techniques.

3.3.8 Command buffers and synchronization primitives

Command buffers in Vulkan are data structures that store a sequence of commands that are sent to the GPU for execution. They act as containers for commands that instruct the GPU on how to perform various operations, such as drawing objects, performing computations or updating buffers.

In Vulkan *semaphores* are binary signaling mechanisms that are used for synchronization between certain GPU operations. An operation can either *wait* at a semaphore or *signal* it. Every operation that is waiting at the semaphore will not start execution until the semaphore is signaled which will happen when a certain operation ends.

Fences, however, are used for *GPU to CPU* synchronization. Using a fence the program can tell when a certain operation or set of operations have finished execution on GPU.

The aforementioned `Engine::createFrameData()` function creates a single command pool per frame from which a command buffer is allocated. Then there are two types of synchronization primitives that are created: two **semaphores** and a **fence**.

- *First semaphore* (`imageAvailableSemaphore`) will be signaled when an image finishes being presented and the `vkQueueSubmit()` function will wait at this semaphore to submit the image rendering commands for execution. This is done to ensure that rendering to a particular swapchain image does not start before this image has finished being presented on screen.

- *Second semaphore* (`renderFinishedSemaphore`) is the opposite of the first one. It will be signaled when the frame is completely rendered and `vkQueuePresentKHR()` will wait for that point in time to supply the frame for presentation on screen.

- *A single fence*, (`inFlightFence`) is used to tell when all previously submitted commands for the current frame finish execution to only then begin next command buffer recording.

Same as for the previously mentioned resources it is important to have separate copies of these synchronization objects per *frame in flight* to ensure that synchronization is always related to a specific frame.

3.4 Scene loading

The application allows loading and saving different scenes containing a sample `OBJ` model. There is a number of scenes that already come together with the program in the GitHub repository at relative path: `assets/scenes`.

The scenes are stored in JSON format which is known to be lightweight and human-readable. This lets the user conveniently modify any settings of a saved scene or easily create a new scene.

The library utilized for JSON parsing is *JSON for Modern C++* [15] by *nlohmann*. This library is very easy to integrate and has minimalistic and intuitive object-oriented API.

A scene *save file* contains:

- *Light properties*: *enabled/disabled*, *light intensity* and *position*. There is a total of 4 lights in a scene, any of them can be disabled or configured. The light parameters can be set through *user interface*(viz. Section 3.7) which allows for additional customization of the scene's visual appearance.

- *Main model properties*: *name*, *position* and *scale*. Not all models have an appropriate size when loaded and are often misplaced. To achieve better visual experience, the aforementioned properties are provided for customization on the user side.

- *Skybox texture name.* Skybox *cubemaps* [16] are always stored in HDR format to allow for correct demonstration of implemented post-processing techniques. All skybox textures are stored at the path `assets/images/skybox/` in their individual folders.

The scene loading process starts right after Vulkan initialization and can be summarized in the following steps:

- *Parse the `.json` save file.*

- *Import all models from their `.obj` files.* This includes: loading the main (sample) model using the path contained in the save file, loading sphere model that is used to display light sources and a cube model to render the skybox on.

- *Load skybox texture* and execute all the necessary initialization steps to enable skybox rendering.

The user is also able to load and save a scene at runtime.

[15] *JSON for Modern C++*: https://github.com/nlohmann/json

[16] *Cubemap* is texture, that consists of six square images, each corresponding to one face of a cube.

3.4.1 Model import

The scene models are stored in `.obj` format and are located at the relative path `asset-s/models`. The *tinyobjloader*[17] library is used to parse the `.obj` file and extract all the necessary information about the model. Information about vertices and normals (optional) is contained in `.obj` file, while materials that correspond to certain vertex groups are described in `.mtl` file.

To be able to render models that consist of multiple materials, each model contains a list of meshes that all have their own material parameters that will be supplied to GPU when rendering starts.

`Engine::loadModelFromObj()` function first calls `tinyobj::LoadObj` that parses the `.obj` and `.mtl` files into library's internal data format, which is represented by several data structures: `tinyobj::attrib_t`, `tinyobj::shape_t`, `tinyobj::material_t`.

These structures are then processes to extract all the necessary information about vertices, normals and materials. For each distinct material that has a diffuse and/or bump texture the textures are loaded using `stb`[18] library.

Other material properties like ambient, diffuse and specular color are stored as parameters for each mesh.

3.4.2 HDR skybox loading

Each scene is loaded along with an HDR skybox, which name is specified in `.json` scene file. The individual skybox cubemaps are located at the path `assets/images/skybox` in individual named folders. Each cubemap is loaded as 6 separate `.hdr` images, each corresponding to a side of the cube, where `(n|p)(x|y|z).hdr` is the name of the specific side (viz. Figure 3.3).

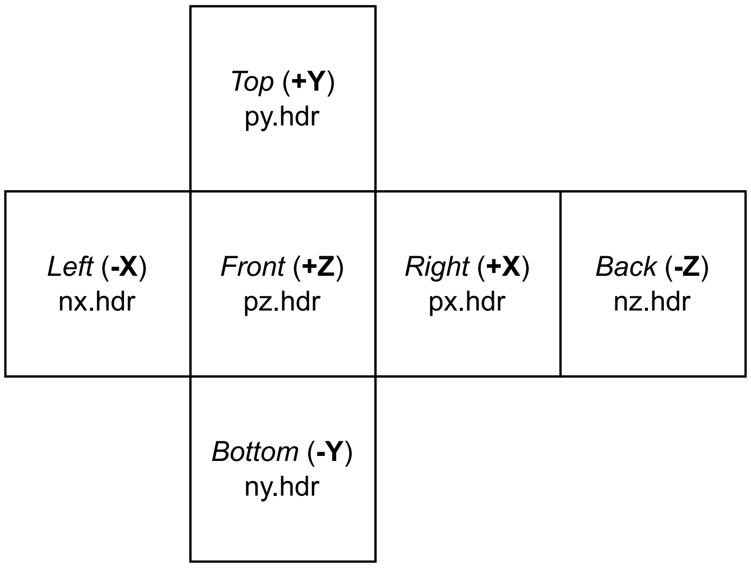


Figure 3.3: Six sides of an HDR skybox.

The implementation of the skybox loading process is based on *Sascha Willems'* Vulkan cubemap demo (source code located at: https://github.com/SaschaWillems/Vulkan/

[17] *tinyobjloader*: https://github.com/tinyobjloader/tinyobjloader
[18] *stb*: https://github.com/nothings/stb

blob/master/examples/texturecubemap/texturecubemap.cpp). All cubemaps were downloaded from *Poly Haven* website (https://polyhaven.com/).

3.5 Rendering of a scene

After Vulkan initialization is finished and a scene is loaded the render loop starts which is contained in `Engine::Run()` function. All the functions that execute the rendering operations are defined in `draw.cpp` module. The main one is `Engine::drawFrame()` that contains all the logic of drawing a frame to the screen.

The logic of this function is schematically illustrated on Figure 3.4. The stage *Record command buffer* is the one where all the rendering operations are listed. This stage is represented by the `Engine::recordCommandBuffer()` function.

Essentially the whole rendering process consists of several stages or, so-called, *passes*.

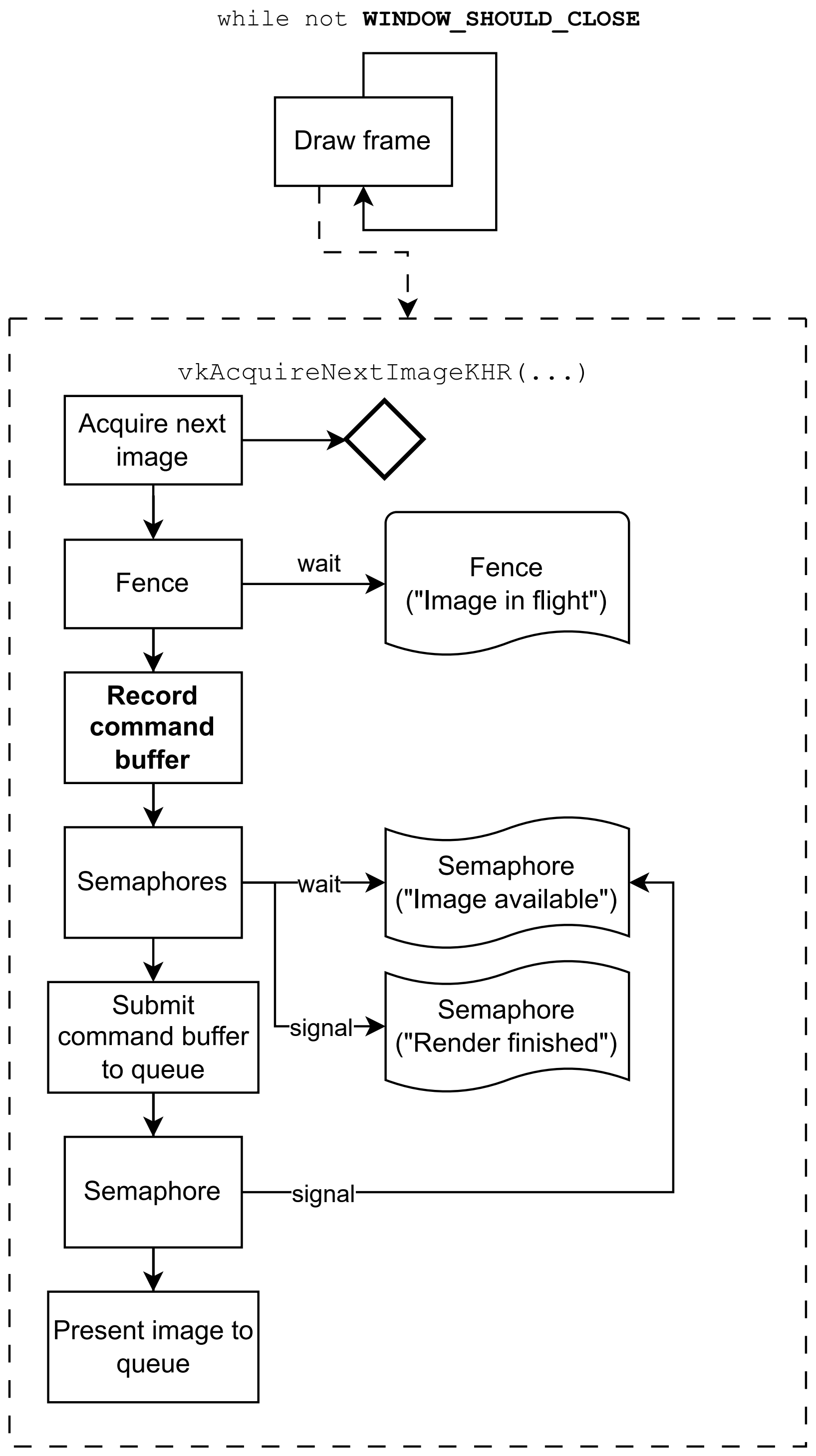

while not WINDOW_SHOULD_CLOSE
Draw frame
vkAcquireNextImageKHR(...)
Acquire next image
Fence
wait
Fence ("Image in flight")
Record command buffer
Semaphores
wait
Semaphore ("Image available")
signal
Semaphore ("Render finished")
Submit command buffer to queue
Semaphore
signal
Present image to queue

Figure 3.4: Execution flow of `Engine::Run()` function.

3.5.1 Shadow Pass

This is where all the objects' shadows are rendered. A cubemap array is used to store omnidirectional shadow map per each light source. For every side of the light source (Up, Down, Right, Forward etc.) all objects are rendered with the corresponding projection matrix, and the depth buffer is stored to the specific side (layer) of the cubemap.

The implementation of shadow mapping technique is well known and probably doesn't need another explanation especially as it isn't related to the objective of this thesis. I have decided to implement shadows because I believe it allows to better demonstrate any HDR effect, otherwise the contrast of the scene would probably be unrealistic.

The sources that I have based my implementation of the shadow mapping on are [36] and [35].

3.5.2 Viewport Pass

All objects of the scene are rendered in this pass. I have named it *viewport pass* because it renders the content that will be displayed in the *viewport*(viz. UI Section 3.7), and the rendered image will have the resolution of the viewport.

The objects are rendered in `Engine::drawObjects()` function. It loops through every mesh of every model and loads the required parameters to GPU according to the mesh material.

The data for material is sent through push constants, these include mesh indices that are used to index the SSBO that contains all the materials data. This SSBO is loaded to GPU beforehand by `Engine::loadDataToGPU()` function along with other buffers.

There is also a UBO that contains scene parameters, including camera position, shadow parameters and all parameters of light sources. There are 4 light sources in the scene that the user can move and configure through user interface.

HDR Skybox is rendered last, after all objects.

Viewport pass uses *Phong lighting model* [19] to light up the scene taking into consideration the shadow map produced by the previous pass. For the materials that have bump textures, bump effect is applied.

3.5.3 PostFX Pass

Depending on which post-processing effects are currently enabled, this pass will execute them in the correct order. User has the ability to turn individual effects on and off through UI. More about all the effects that are implemented will be written in the following Section 3.6. That section will also include the description of various abstractions that were implemented to support the correct execution of multiple effects, enabling and disabling them.

3.5.4 Swapchain Pass

This is the last pass that is necessary only because of the presence of user interface. I have named it *swapchain pass* because it outputs the image with the resolution of the swapchain (i.e. resolution of the window itself). It executes all rendering commands that are provided

[19] *Phong reflection model* is an empirical model of local illumination that describes rough and shiny surfaces with the combination of diffuse and specular reflection.

by ImGui to render the interface. ImGui takes the viewport texture (the one rendered as a result of viewport pass) and combines it with the rest of UI to produce the final image.

3.6 Post-processing effects

This chapter will finally introduce to the reader the implementation of HDR techniques that were described in the *State of the art* chapter (2).

All HDR techniques in `vulkan-hdr-demo` are implemented as *shaders* to make use of modern GPU capabilities. These techniques are essentially post-processing effects, so I have decided to exclusively use *compute shaders* [20] to implement them. There is a couple of reasons for that:

- They prove to be efficient for image processing tasks such as filtering, luminance computation, sampling etc.
- In Vulkan API compute shaders are easier to integrate into the rendering pipeline than *fragment shaders* [21], which cannot be executed without a vertex shader. Fragment shaders would also require the user to create a *renderpass*, bind vertex buffers etc.

Given that `vulkan-hdr-demo` allows the user to turn individual effects on/off and combine them freely, it was necessary to come up with a suitable abstraction that would allow that functionality.

3.6.1 PostFX abstractions

Because every post-processing effect requires invocation of multiple shaders, each effect is split into so-called *stages* (`PostFXStage`) that encapsulate a certain shader. The map of stages is contained in the class named `PostFX`, which is the cental point of post-processing abstraction layer. Each stage in the map is indexed by a string key which starts with the prefix of the effect this stage is part of. `PostFX` class also contains the maps for *attachments* (`Attachment`) and *mipmap attachments* (`AttachmentPyramid`). Attachments are essentially the render targets or sources for individual shaders that get executed as part of a post-processing effect. This means that each `PostFXStage` has a certain number of attachments that it interacts with. When a stage is executed, the attachments it requires need to be *bound* to be accessible from the shader.

Usually a shader would only require a single image per attachment (i.e. an input image or output image). However, certain more sophisticated techniques like *Exposure Fusion* (Section 2.5) or *Bloom* (Section 2.6) require whole mip chains as attachments, because some of their shaders may be executed sequentially over mips of different sizes, for example, in case of *downsampling* or *upsampling*. This is why I have decided to add `AttachmentPyramid` type, which provides a uniform interface for the whole image pyramid (mipmap). Figure 3.5 shows an example scheme of post-processing effect stages and their attachments.

As already mentioned in the Section 3.5, post-processing effects are executed after the scene is rendered as their input is the rendered image of a 3D scene. Depending on which effects are currently enabled, its stages get sequentially executed. For every stage:

[20] *Compute shader* is a programmable stage in modern GPUs that allows for execution of general-purpose parallel computing tasks.

[21] *Fragment shader* is a programmable stage of graphics pipeline responsible for computing the color and other attributes for individual pixels on the screen.

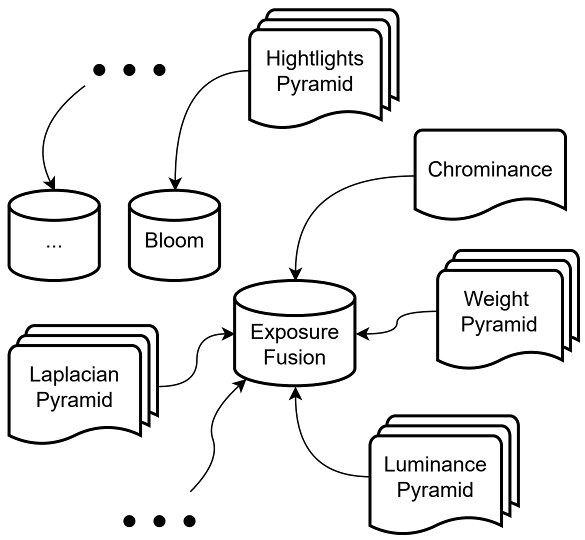


Figure 3.5: Interaction scheme of `PostFX` stage.

- The pipeline is bound, which enables the use of the corresponding shader.
- The necessary image attachments are bound to descriptor sets and the *descriptor sets* are updated which transfers the updated memory to GPU. *This step is optional* as it only needs to be done if some of the attachments have changed. Practically, for performance reasons this step is only executed once during initialization because for each stage the attachments do not change.
- Descriptor sets are bound to the pipeline to make their updated memory available in the shader.
- The compute shader operation is *dispatched* (executed).

This functionality is implemented in methods of `PostFXStage` class. Figure 3.6 illustrates the process of execution. Usually, because input for a stage is the output from the previous stage, a memory barrier is needed to ensure correct memory synchronization between individual stages.

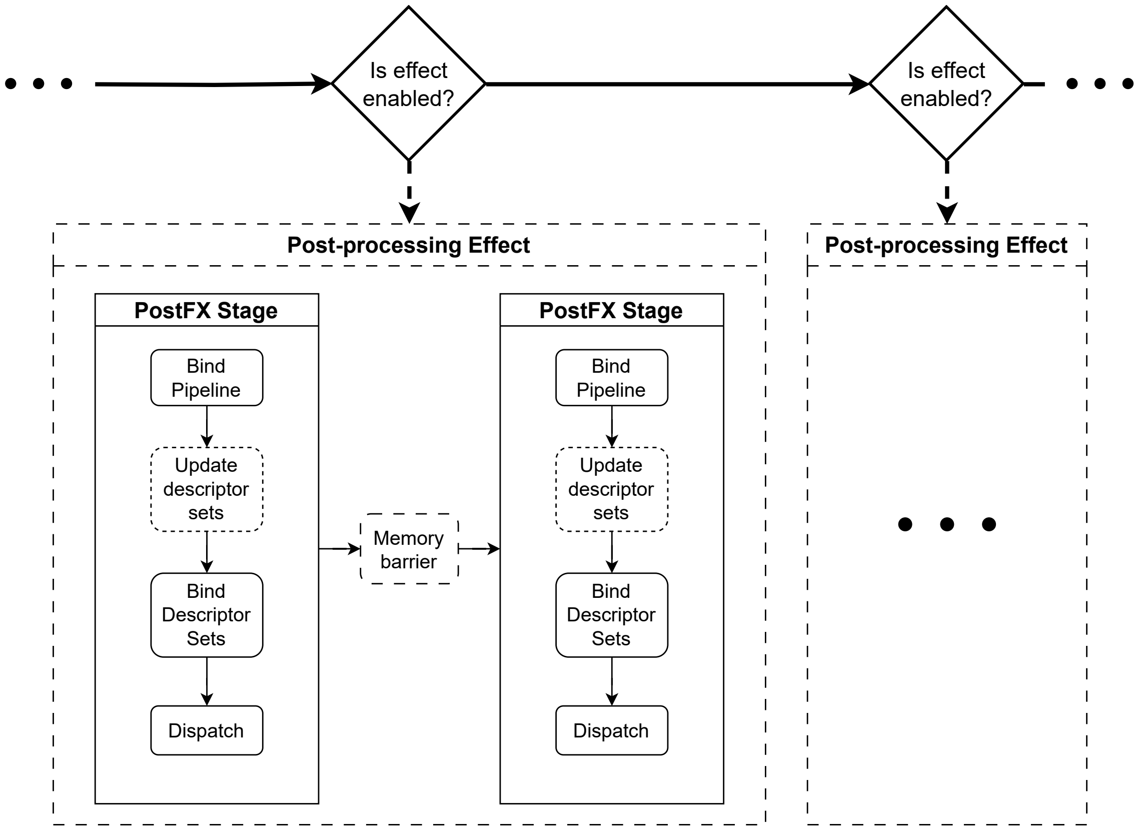


Figure 3.6: Execution scheme of post-processing effects.

3.6.2 Uniform tone mapping operators

The following figures show comparison of individual global tone mapping operators against the original clamped HDR image. All the images have bloom effect applied, because this allows to better demonstrate each of the effects.

The assessment of individual operator is subjective, and the reader is free to decide which of the operators he finds the most visually appealing. I will try to evaluate them from my point of view.

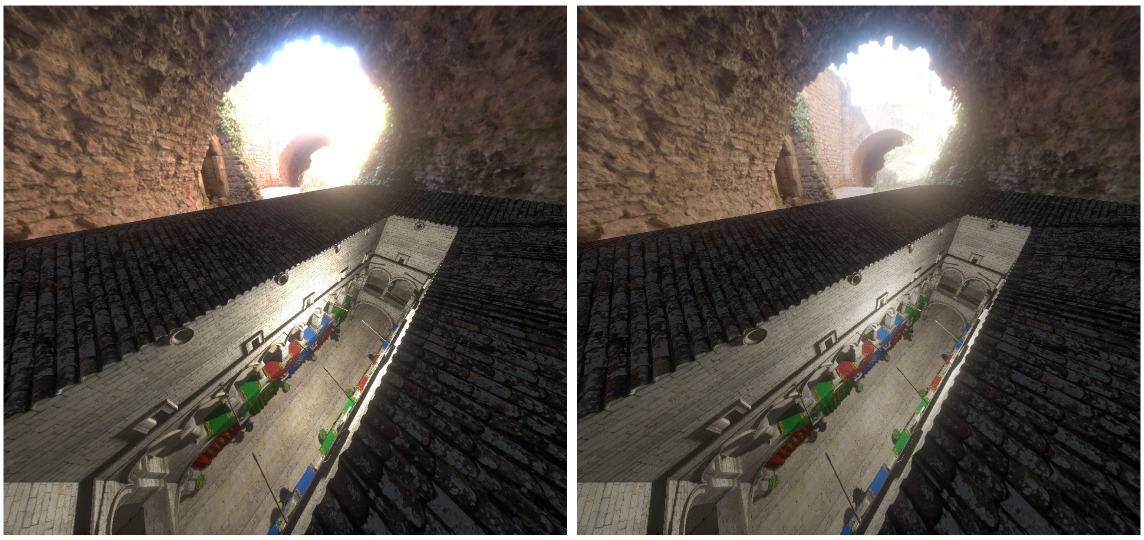


Figure 3.7: Original image (left) vs. applied *Photographic (Erik Reinhard)* tone mapping (right).

On the Figure 3.7 we can see the effect of *Reinhard's photographic operator* that was described in Subsection 2.2.1. We can see that it allows the viewer to see a big portion of the overexposed outdoor area (on the skybox). It, however, gives the image a bit of a "washed out" look.

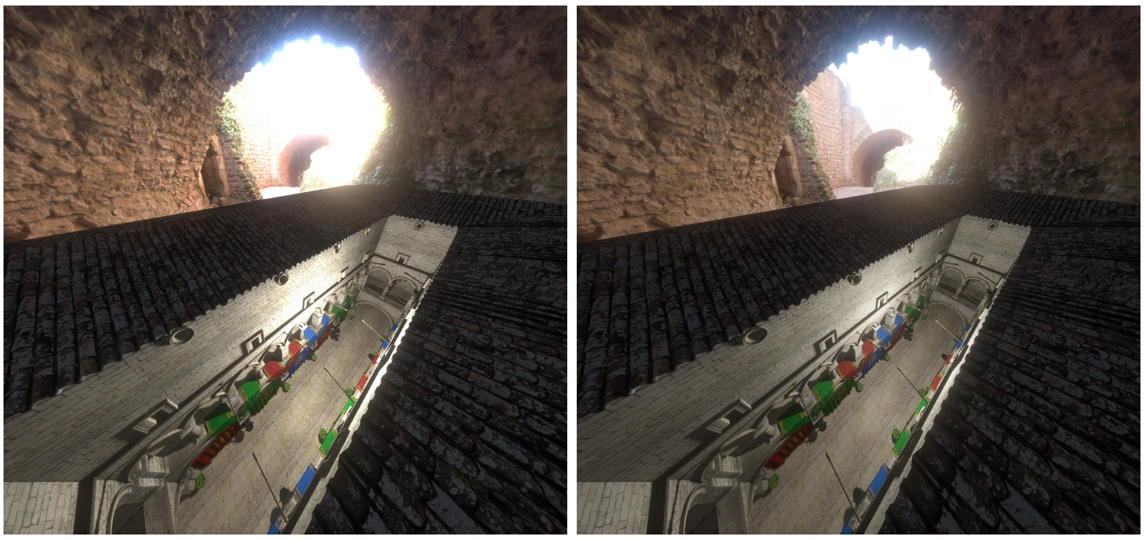


Figure 3.8: Original image (left) vs. applied *Uncharted2 (John Hable)* tone mapping (right).

Uncharted 2 filmic operator by *John Hable* that was described in Subsection 2.2.2 is shown on the Figure 3.8. At the first glance, it is hard to spot any difference between Reinhard's operator and this one, but by taking a closer look, it is possible to see, that this operator produces a bit more colorful and rich image, because it doesn't compress the brighter values so much. However, as a result, slightly more details are lost in the overexposed area on the skybox.

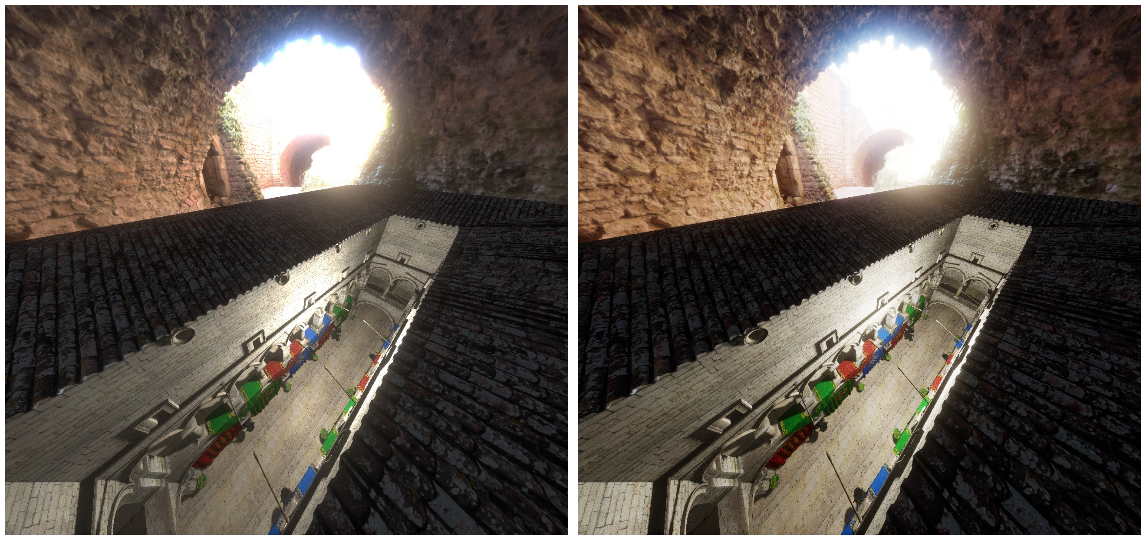


Figure 3.9: Original image (left) vs. applied *ACES (Krzysztof Narkowicz)* tone mapping (right).

Figure 3.9 demonstrates the *ACES-based operator* by *Krzysztof Narkowicz* (Subsection 2.2.3). It provides even less compression of the highlights region, which can be desired if our intention is to produce an impression of a highly lit outdoor scene.

The differences between the demonstrated operators are subtle, and their visual impression depends on many factors, including the selected skybox, light intensities in the 3D scene, lighting conditions in the room around the viewer etc.

Implementation of all global operators can be found inside the *GLSL* shader `shaders/src/incl/tone_mapping.glsl`.

3.6.3 Bilateral filter tone mapping

Figures 3.10 and 3.11 show the results of bilateral filter tone mapping by *Durand and Dorsey*, which was described in Section 2.4.

We can see that this relatively simple spatially-variant tone mapping operator allows to extract a lot of details from the image. It is possible to see the highly exposed outdoor region on the skybox, while also seeing a lot of details in the darker regions. Unfortunately, as a result of this operator some color values become oversaturated.

The user can configure the following parameters:

- *Base offset* - a variable that is added to the pixel values of the base layer.
- *Base scale* - a variable that the base layer is multiplied with (after the offset).
- *Bilateral radius* - the radius of bilateral filter (in pixels). *Increasing this parameter can negatively impact performance!*.

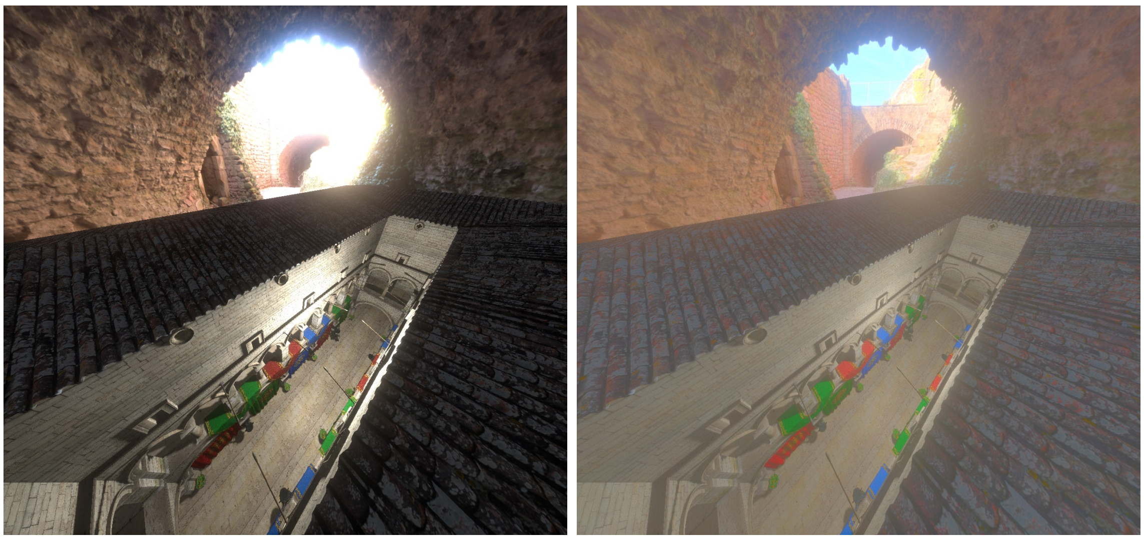


Figure 3.10: Original image (left) vs. applied *Bilateral (Durand and Dorsey)* tone mapping (right). With enabled bloom effect.
(Base scale = 0.3, base offset = 14, bilateral filter radius = 5).

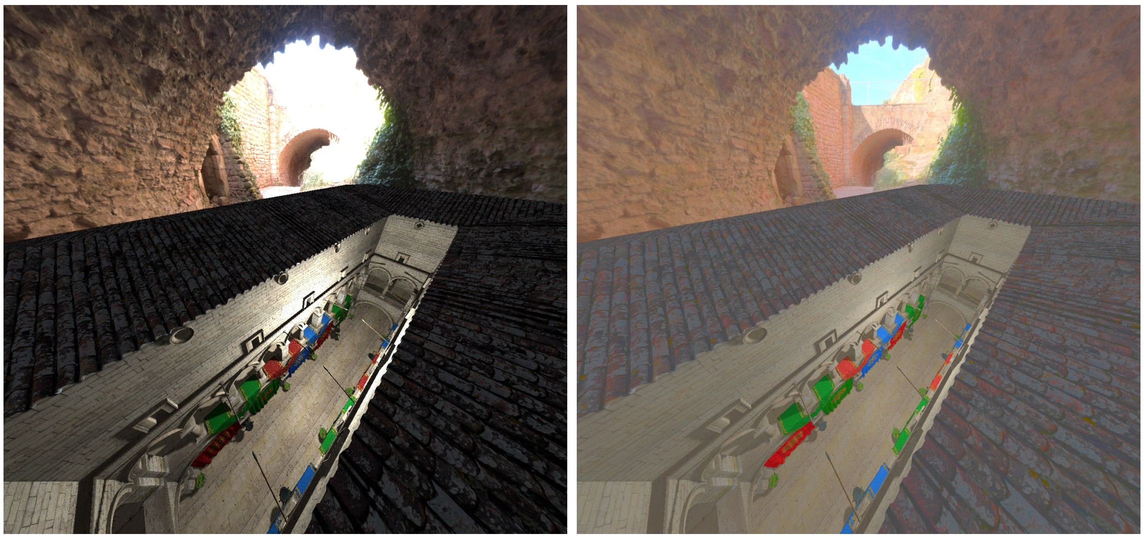


Figure 3.11: Original image (left) vs. applied *Bilateral (Durand and Dorsey)* tone mapping (right). No bloom effect.
(Base scale = 0.3, base offset = 14, bilateral filter radius = 5).

- *Range sigma* - the sigma used for value difference in bilateral filter. *Normally shouldn't be modified.*

The *spacial sigma* is automatically set to 2% of the viewport size.

Because this operator requires several subsequent image processing operations the implementation is split between multiple compute shaders that can be found at the path: `shaders/src/ltm_durand_*.comp`. There are several preprocessor macros defined in `shaders/incl/defs.glsl` that modify the implementation of the operator. Most of them are inactive and are only there for historical reasons.

3.6.4 Exposure fusion

This effect was arguably the most hard to implement. It consists of multiple stages, which include recursive downsampling and upsampling of mip maps.

Figure 3.12 demonstrated the effect applied with bloom turned on. We can see that the extremely exposed outdoor region on the skybox becomes clearly visible, but it is surrounded by a relatively big halo.

It is for the reader to decide whether the halo effect looks realistic or not, considering the enabled bloom effect, but I would rather call it a technical limitation of this method. Specifically, the amount of exposure levels used in my implementation seems to not be enough for this particular case. I have used only 3 exposure levels, the so-called *shadows exposure* (for the darker regions), *normal exposure* (normal brightness of the input image) and *highlights exposure* (for the brighter regions of the image). Introducing more levels would produce a better visual quality but would be much less performant and harder to implement.

The halo could also potentially be decreased by tweaking the parameters of the bloom effect.

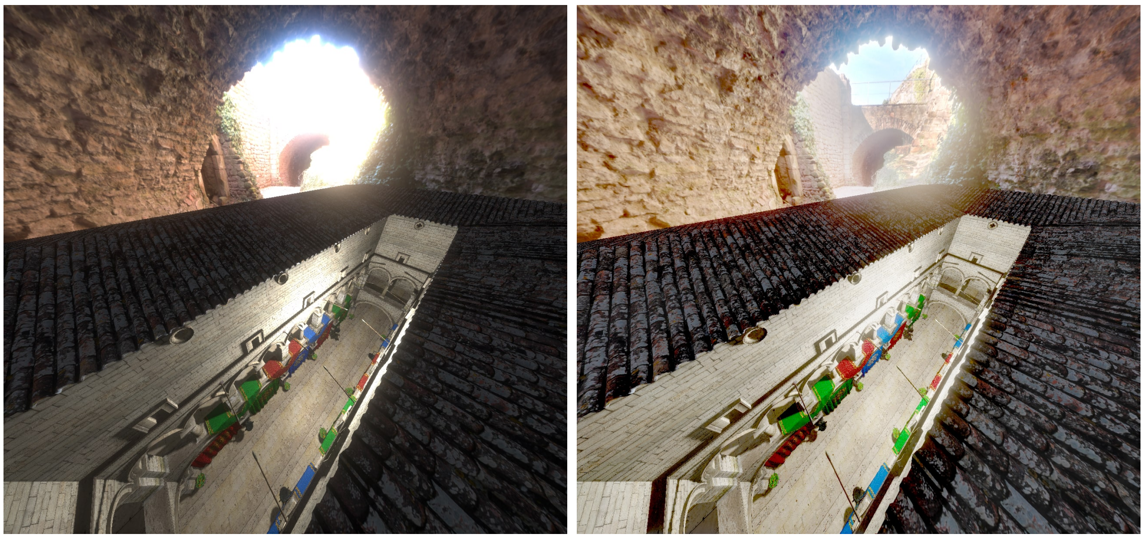


Figure 3.12: Original image (left) vs. applied *Exposure Fusion* (right). With enabled bloom effect.

Shadows exposure: 2.0, *Highlights exposure*: -5.5.

If we take a look at Figure 3.13, where bloom effect is disabled, we can see that there is almost no halo visible (only slightly around the outdoor region of the skybox) and the amount of details preserved in the image is truly impressive. However, to achieve a good result it is necessary to set the exposure levels accordingly, which can be a bit of a problem in case this effect is used in a dynamic real-time rendering application. It could be a goal for the future research to find a way to make the effect look right in any dynamic environment.

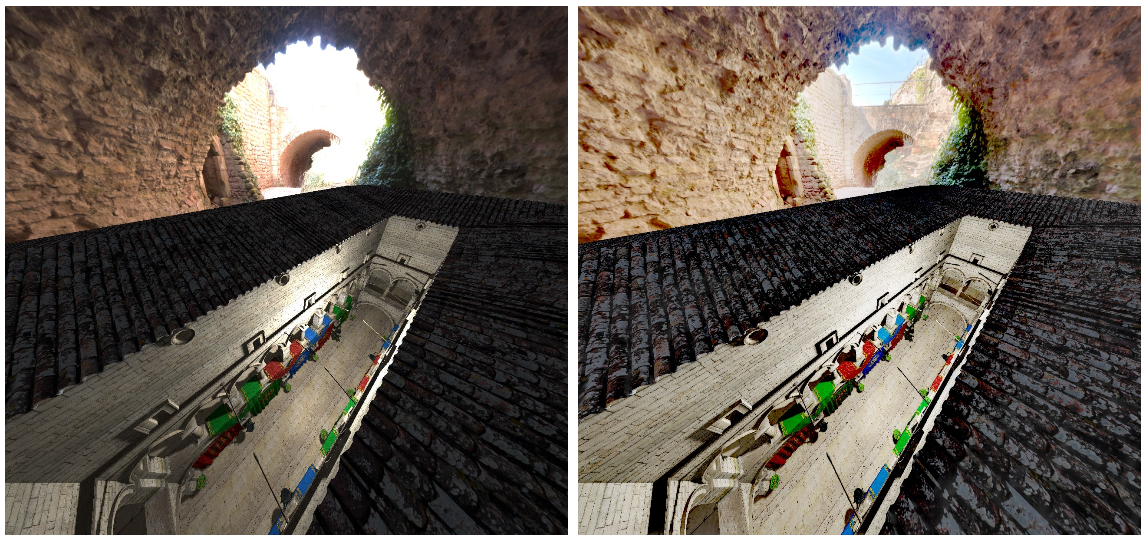


Figure 3.13: Original image (left) vs. applied *Exposure Fusion* (right). Bloom effect disabled.
Shadows exposure: 2.0, *Highlights exposure*: -5.5.

There are several parameters that user can alter:

- *Shadows exposure* - the exposure level of the darker region of the image.
- *Highlights exposure* - the exposure that is to be applied to the brighter regions.
- *Exposedness weight sigma* - defines the weight of the *exposedness* criterion. *For more details see Section 2.5.*

Individual shaders can be found at the path `shaders/src/ltm_fusion_*.comp`.

3.6.5 Bloom effect

Out of two variants of bloom effect described in Subsection 2.6.2, I have implemented the more realistic one in my application, following all the individual stages of the algorithm.

Figure 3.14 demonstrates the result of my implementation of the bloom effect in comparison with original unmodified image. It makes little sense to use such effect without tone mapping so the images with enabled bloom are also tone mapped with *ACES-based global operator* (was described in Subsection 2.2.3).

From my point of view, the effect seems relatively realistic. The bright regions of the image are covered with a smooth bloom halo which makes them naturally blend with the surroundings.

There is just a single parameter that the user is able to set for this effect:

- *Bloom weight* - multiplier for the bloom layer that is combined with the original image. It defines bloom intensity/brightness.

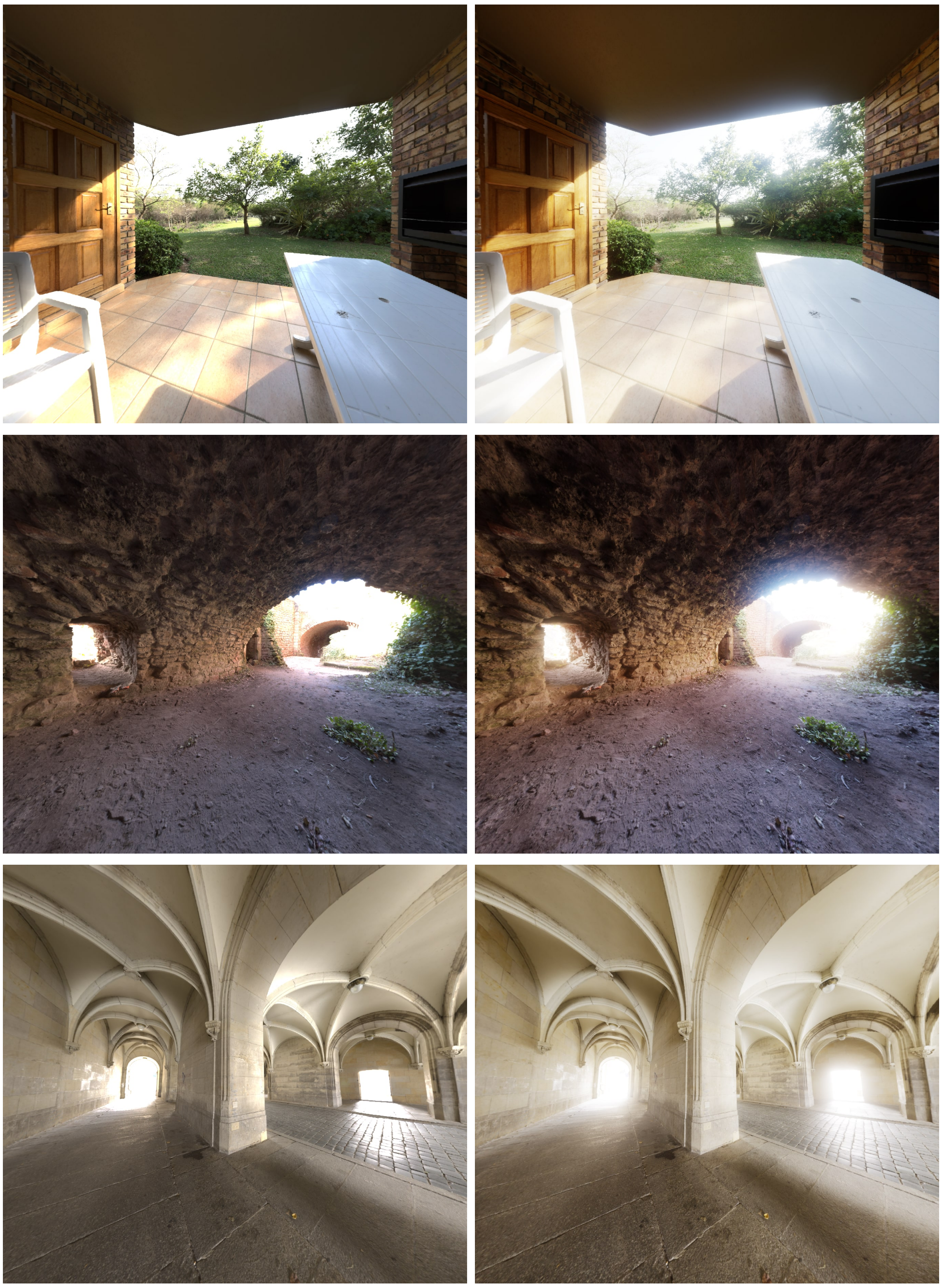


Figure 3.14: Results of bloom effect implementation demonstrated on various HDR skyboxes. Original image (left), final tone-mapped image (right).
Bloom weight = 0.3, number of bloom mips = 7.
Tone mapping used for final image: *ACES Narkowitz* (viz. Section 2.2.3)

My implementation of the effect has a minor flaw, which is the presence of subtle *stair-like artifacts* around the brightest regions of the image (demonstrated on Figure 3.15).

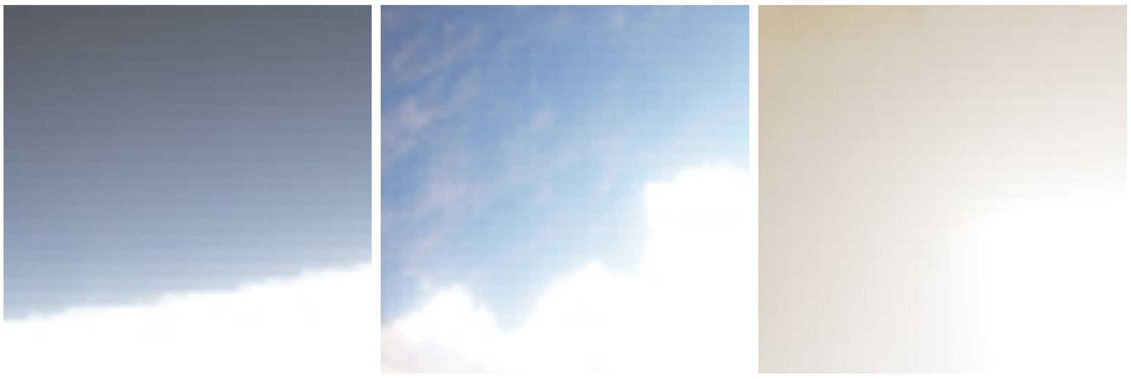


Figure 3.15: *Stair-like* artifacts in my implementation of the bloom effect.
Artifacts become visible around the brightest regions.

Even though I have used the proposed sampling method of *13 bilinear samples* for downsampling and *tent filter* for upsampling, the artifacts are still visible. I am not entirely sure what is the cause of this problem, but I suspect this might be because I have implemented the bilinear sampling manually. This is because the *image* object (`image2D` in GLSL) does not support subpixel sampling (i.e. it can only load/store at integer pixel coordinates). I have chosen to sample from the *image* object in all compute shaders, because unlike `sampler2D` it allows for both read and write operations on the image, thus, it is also easier to integrate it into the compute pipeline. If I had used `sampler2D`, I would have to add a lot of image layout transitions between read and write access and also the shader code would probably become more complicated.

The bloom shaders are located at the path `shaders/src/bloom_*.comp`, the shader code for various sampling methods can be found in `shaders/incl/sampling.glsl`.

3.6.6 Dynamic exposure mechanism

It is hard to demonstrate the process of dynamic exposure adaptation by taking images of the scene, but Figure 3.16 tries to somehow show the adaptation process.

As described in Subsection 2.3.2, first the luminance histogram is computed, then the histogram is summed up and averaged to get the average luminance of the image and finally the eye adaptation coefficient is calculated based on the average luminance. This coefficient is used to shift the overall luminance of the scene.

The individual parameters of eye adaptation, that are controlled by the user are:

- *Minimal/maximal* log_2 *luminance* - specify the range of luminance values in the histogram. When calculating the bin index for the input luminance, the input luminance is converted to log_2 space and mapped to the $[MINlog_2 - MAXlog_2]$ range.
- *Lower/upper luminance histogram bounds* - allow to skip some percentage of the darkest or brightest pixels, to make the effect generally smoother. The percentage that should be skipped may vary depending on the scene and artistic preferences.
- *Histogram bin index weight* - the weight that the histogram bin index is multiplied with at the stage of average luminance computation. The higher this weight is, the

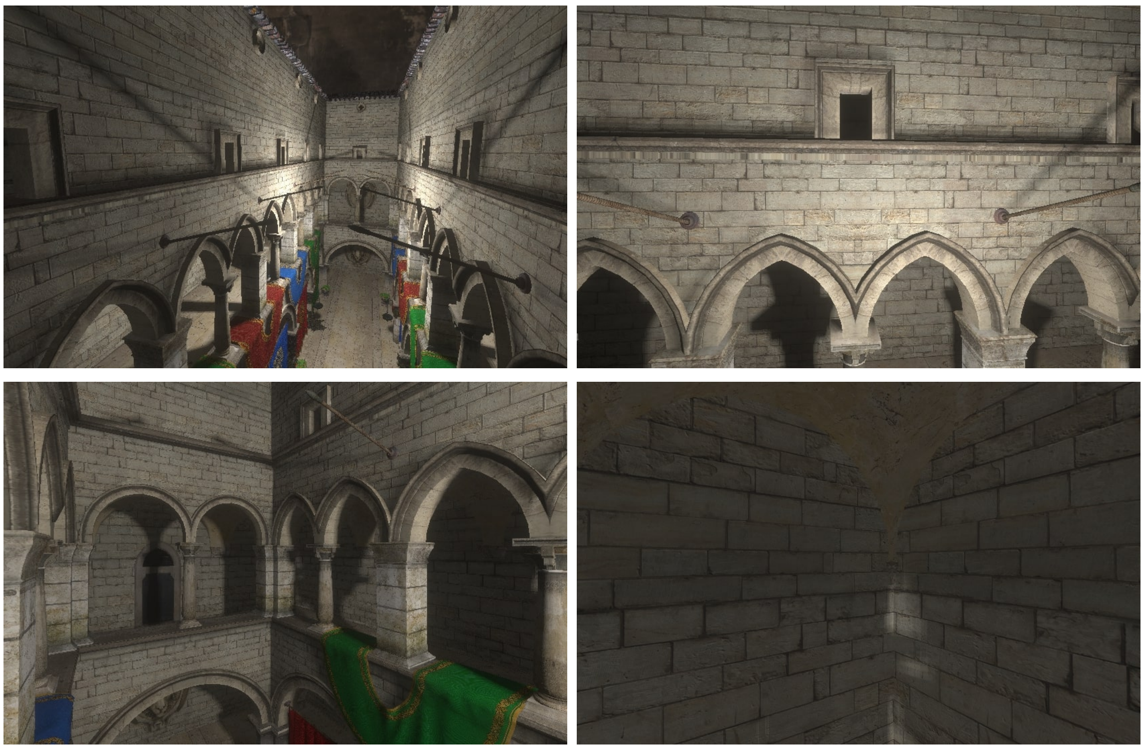


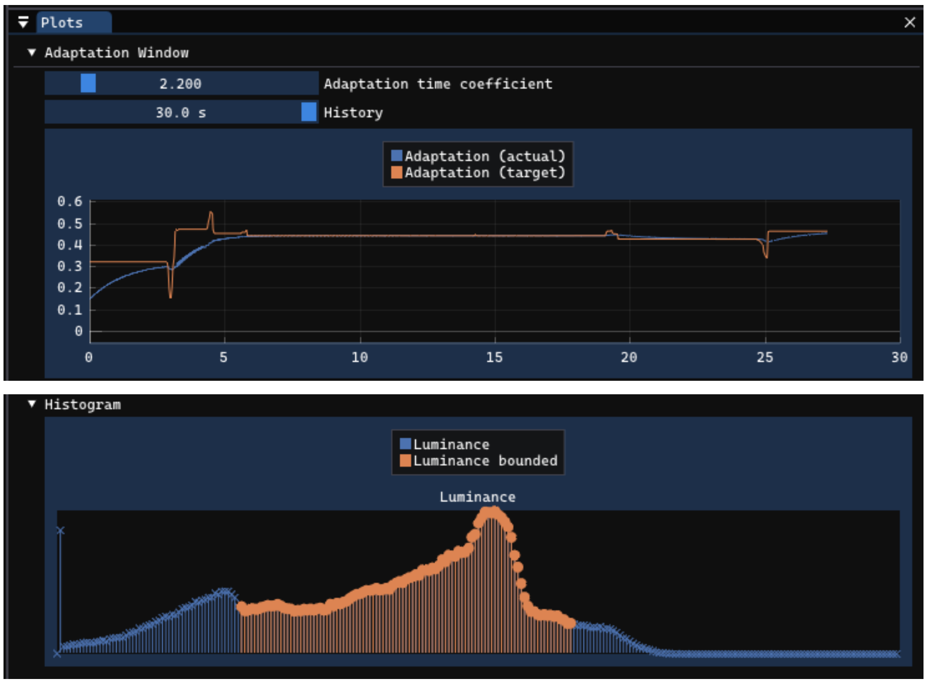
Figure 3.16: Approaching highly lit area (top) and shadow area (bottom) from far to close distance.

The image darkens as the camera gets closer to the bright area and gets brighter when approaching the area in shadows.

brighter the image will be in general, but also when it is higher than 1, brighter regions of the image will have bigger influence than darker ones. *This parameter was not described in the original implementation* of eye adaptation technique and was added by me because it provides more options for configuration to achieve a more prominent visual effect. It is left for the user to decide whether to use this parameter or not.

The mentioned parameters are supplied to GPU through special *Shader Storage Buffer Object (SSBO)* and a *Uniform Buffer (UBO)* for read-only data.

Figure 3.17 shows the UI plots that I have added both for debugging purposes and for visualization of the adaptation process.


2.200 Adaptation time coefficient

30.0 s History

Figure 3.17: Real-time UI plots of eye adaptation window (top) and luminance histogram (bottom).

Both graphs are plotted in real-time.

All 3 stages of exposure adaptation are implemented on GPU as compute shaders. The individual shaders can be found at the path: `shaders/src/expadp_*.comp`. The only operation that was implemented on CPU is the calculation of time coefficient used in inverse exponent function. It was done to speed up performance and to avoid loading *time delta* (time elapsed since last frame) to GPU.

3.6.7 Gamma settings

Since it is unknown which *format & color space* combination will be chosen for the swapchain on the user's device, it makes sense to allow the user to choose gamma settings manually. The user is presented with 3 options:

- *Off* (0) - no gamma correction is applied. *This is the default option.*

- *On* (2.2) - applies gamma of 2.2, which is a relatively close approximation to EOTF [22] of sRGB space.
- *Inverse* ($\frac{1}{2.2}$) - inverse gamma (approximate sRGB OETF [23]).

By default, it is assumed that both the format and color space are non-linear, so there is no need to manually add any gamma correction. However, it can potentially happen that only a linear HDR color space is supported, so the user is able to change the gamma settings to achieve better visual outcome.

Because there is a discrepancy between the format of ImGui color values and the non-linear color spaces that are preferred for the swapchain, I have decided to modify ImGui vertex shader source code to make the colors look right on sRGB render target. I have simply added a conversion from sRGB to linear space for the input color values (viz. Listing 3.1).

```
vec4 sRGBtoLinear(vec4 sRGB) {
    bvec3 cutoff = lessThan(sRGB.rgb, vec3(0.04045));
    vec3 higher = pow((sRGB.rgb + vec3(0.055)) /
        vec3(1.055), vec3(2.4));
    vec3 lower = sRGB.rgb / vec3(12.92);

    return vec4(mix(higher, lower, cutoff), sRGB.a);
}
```

Listing 3.1: sRGB to linear conversion

3.7 User interface and navigation

To implement the user interface in my demo application (`vulkan-hdr-demo`) I have decided to use *Dear ImGui*, an easy to integrate, immediate-mode graphical user interface library.

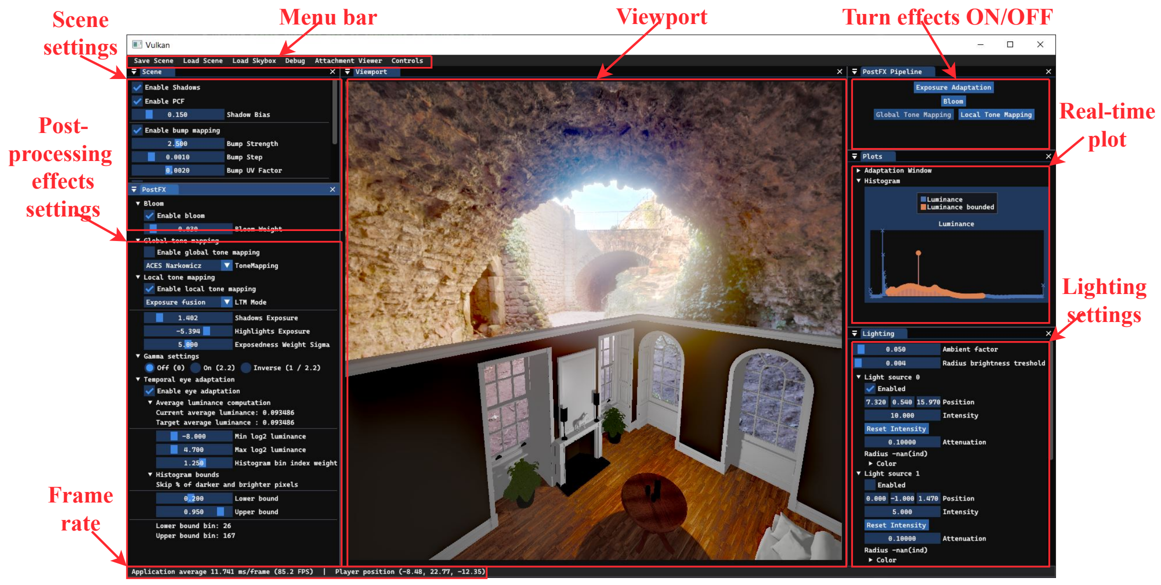
The application provides a complex user interface with separable windows, which can be accessed through the *menu bar*. The user has multiple options for configuration of individual effects and scene parameters. Figure 3.18 shows the breakdown of individual UI windows.

The user is able to:

- Freely move around the rendered 3D scene using *W, A, S, D* keys and rotate the camera using *mouse movement*.
- Assess the visual quality and efficiency of individual post-processing effects by toggling them on and off with the help of *PostFX Pipeline* window.
- Modify and configure scene settings using *Scene* window. This includes enabling/disabling shadows, setting shadow quality, moving main model and modifying camera field of view.

[22] *Electro-Optical Transfer Function (EOTF)* defines how brightness values in the digital domain are mapped to the brightness levels that are actually displayed on the screen.

[23] *Optical-Electronic Transfer Function (OETF)* is the inverse process of EOTF, converting an optical signal into digital visual data.



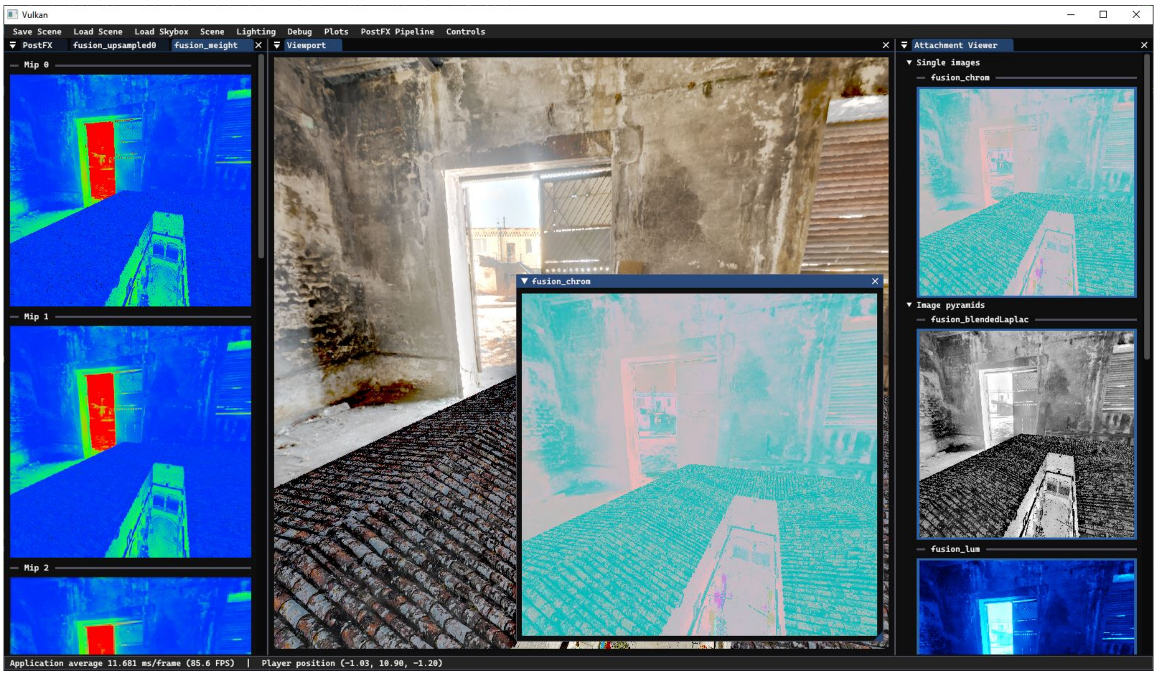
```
Scene
[✓] Enable Shadows
[✓] Enable PCF
0.150    Shadow Bias
[✓] Enable bump mapping
2.500    Bump Strength
0.0010   Bump Step
0.0020   Bump UV Factor

PostFX
▼ Bloom
  [✓] Enable bloom
  0.030    Bloom Weight
▼ Global tone mapping
  [ ] Enable global tone mapping
  ACES Narkowicz    ToneMapping
▼ Local tone mapping
  [✓] Enable local tone mapping
  Exposure fusion   LTM Mode
  1.402    Shadows Exposure
  -5.394   Highlights Exposure
  5.000    Exposedness Weight Sigma
▼ Gamma settings
  (●) Off (0)  ( ) On (2.2)  ( ) Inverse (1 / 2.2)
▼ Temporal eye adaptation
  [✓] Enable eye adaptation
  ▼ Average luminance computation
    Current average luminance: 0.093486
    Target average luminance : 0.093486
    -8.000   Min log2 luminance
    4.700    Max log2 luminance
    1.250    Histogram bin index weight
  ▼ Histogram bounds
    Skip % of darker and brighter pixels
    0.200    Lower bound
    0.950    Upper bound
    Lower bound bin: 26
    Upper bound bin: 167

Lighting
0.050    Ambient factor
0.004    Radius brightness treshold
▼ Light source 0
  [✓] Enabled
  7.320 0.540 15.970 Position
  10.000   Intensity
  Reset Intensity
  0.10000  Attenuation
  Radius -nan(ind)
  ▶ Color
▼ Light source 1
  [ ] Enabled
  0.000 -1.000 1.470 Position
  5.000    Intensity
  Reset Intensity
  0.10000  Attenuation
  Radius -nan(ind)
  ▶ Color

Application average 11.741 ms/frame (85.2 FPS)  |  Player position (-8.48, 22.77, -12.35)
```

Figure 3.18: Breakdown of `vulkan-hdr-demo` application UI.

- Tweak lighting settings, which includes moving individual light sources, setting their intensity and light radius, or turning them on and off. This can be done through the *Lighting* window.
- Take a screenshot of the viewport. This can be particulary useful, because it allows to later compare the results of different effects, or to analyze the imagess using some other software.
- View individual image attachments in real-time in the *Attachment viewer* window to get a better understanding of how individual effects are implemented. This visualization capability can also be extended relatively easily in case new effects are added to the application. The *Attachment viewer* is can be seen on Figure 3.19.

All windows can be closed and reopened through the *Menu bar*.

UI code is contained in `ui.h` and `ui.cpp` files. In case the user would like to add new HDR effects, it would be relatively easy to extend the UI to display various parameters of a newly added effect.



```
Application average 11.681 ms/frame (85.6 FPS)  |  Player position (-1.03, 10.90, -1.20)
```

Figure 3.19: *Attachment viewer* window in `vulkan-hdr-demo` application.

3.8 Limitations and possible improvements

I was working on this thesis in a restricted time interval and without much previous knowledge in computer graphics field, so, obviously, this work may have various flaws and limitations.

Following is the list of flaws of this thesis that I think would be great to fix and improvements that could serve as the next steps in further research on this topic. The individual points are ordered descending by priority:

1. Implement Physical Based Rendering (PBR) pipeline in the application instead of Phong lighting model. PBR has become a de facto standard for the rendering pipeline in all modern rendering engines. I believe it would also make any HDR effects look more natural, especially the Bloom effect.

2. Fix stair-like artifacts in Bloom effect implementation (viz. Subsection 3.6.5).

3. Find a way to improve Exposure Fusion effect performance and add more exposure levels to increase visual quality. Currently, this effect is probably too taxing in terms of performance to be used at runtime, for example, in a video game. However, adding more exposure levels would further decrease performance, so it can be a complicated problem to find a compromise in this case.

4. Try interconnecting dynamic exposure with Exposure Fusion. For example, by setting exposure levels based on the current average exposure in the scene. It would require some trial and error to figure out if it makes sense to try connecting these effects like that, but in the end it may lead to great visual results.

5. Improve dynamic exposure mechanism by automatically setting the bounds for min/-max luminance. It could be achieved by adding a shader stage at the beginning

that would calculate min and max luminance in the rendered image. These calculated bounds would be used in further stages to correctly position the luminance histogram.

6. Implement environment lighting with HDR skybox. Look for a method to add some kind of environment lighting using HDR skybox in the application. It would benefit the realism of the scene and would allow to better demonstrate various HDR effects.
7. Refactor and improve application design. Designing a Vulkan application can be a very challenging task, and because I was completely new to Vulkan, it is understandable that the design, that I came up with is far from perfect. If someone wanted to extend my application, it would be a good idea to rethink the design.
8. Conduct more testing on various devices with monitors that have HDR support. During the testing of my application I only had access to one device with a single HDR monitor. It would be great to test the application on other HDR-supporting devices to see if certain effects play well with HDR display.
9. Make UI more intuitive and allow for more customization of the scene. E. g. allow the user to add other models to the scene at runtime to test HDR effects on different scene setups.

Chapter 4

Experiments

Application was tested on *2* devices with different technical parameters. Specifications of individual test setups are listed in Table 4.1. *Second computer* also has a monitor with HDR support connected.

Device	GPU	CPU	Monitor	OS
D1	Nvidia GTX1650	Intel Core i7 9750H	DELL P2422H	Win 10
D2	Nvidia RTX 4090	AMD Ryzen 9 7950X3D	*DELL AW2723DF*	Win 10

Table 4.1: Specifications of testing device setups.
Name of the monitor with HDR support is *emphasized*.

Table 4.2 and Table 4.3 contain list of *format & color space* combinations that were supported during testing on test setup *D1* and *D2* respectively. Additionally, it contains the combination that was chosen by the application in the case of each device.

`VkFormat`	`VkColorSpaceKHR`
`VK_FORMAT_B8G8R8A8_UNORM`	`VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`
`VK_FORMAT_A2B10G10R10_UNORM_PACK32`	`VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`
`VK_FORMAT_B8G8R8A8_SRGB`	`VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`
`Selected tuple`	
`VK_FORMAT_B8G8R8A8_SRGB`	`VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`

Table 4.2: Available *image format & color space* combinations on test setup *D*1.

We can see that on device *D1* a typical SDR color space (`VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`) was chosen, because this device does not support any HDR formats.

In the case of test device *D2*, the selected color space is `VK_COLOR_SPACE_HDR10_ST2084_EXT`, which is an HDR color space, defined by *ITU-R Recommendation BT.2100*. It uses the *perceptual quantizer (PQ)* transfer function, that replaces the SDR gamma curve, which allows representing luminance level in the range 0.0001 *to* $10000\ cd/m^2 (nits)$.

Table 4.4 contains estimated performance on different testing device setups. Performance is listed as *Frames per second (FPS)*, which indicates how many times per second the application was able to update the content of the window (i.e. to render the scene and redraw the UI). FPS was measured for the interval of *15 seconds*, while rapidly moving around the scene to ensure that the measurement is not restricted to a specific camera view.

VkFormat	VkColorSpaceKHR
VK_FORMAT_B8G8R8A8_UNORM	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
VK_FORMAT_B8G8R8A8_SRGB	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
VK_FORMAT_R8G8B8A8_UNORM	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
VK_FORMAT_R8G8B8A8_SRGB	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
VK_FORMAT_R16G16B16A16_SFLOAT	VK_..._EXTENDED_SRGB_LINEAR_EXT
VK_FORMAT_A2B10G10R10_UNORM_PACK32	VK_COLOR_SPACE_HDR10_ST2084_EXT
VK_FORMAT_A2B10G10R10_UNORM_PACK32	VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
Selected tuple	
VK_FORMAT_A2B10G10R10_UNORM_PACK32	VK_COLOR_SPACE_HDR10_ST2084_EXT

Table 4.3: Available *image format & color space* combinations on test setup *D*2.

	FPS	
Effect	D1	D2
No effect	185.26	1040.33
Global TMO	178.33	1027.6
Eye Adaptation	165.60	982.4
Bilateral TMO	153.06	937.133
Bloom	140.33	883.333
Exposure Fusion	109.13	819.0

Table 4.4: Performance of individual effects on different test setups.

From the table it is clear that *global tone mapping* operators are the most efficient in terms of performance, next by efficiency is the *eye adaptation*, which is also expected as this effect doesn't require any complex computations. *Bilateral filter tone mapping*, however, is less efficient, most likely because of bilateral filter itself, which is doing a lot of computations (especially if bigger *radius* is set). Next comes the *bloom effect*, which is expected to be hard on performance, because the *downsampling, upsampling* and *blur* is executed for each mip level. And by far the most performance-heavy effect is *exposure fusion* because of how many processing stages it requires to achieve the result.

There is no doubt that there is room for optimizations for every effect, but it would require more proficiency and time for experiments.

Chapter 5

Summary

The main goal of this thesis was to describe and demonstrate an extensive range of techniques that can be used to enhance visual quality of 3D virtual environment rendered in High Dynamic Range. Before starting to implement various methods of HDR rendering it was necessary to conduct an extensive study. This study is thoroughly described in the first half of the thesis, starting with the general understanding of HDR concept and continuing with different methods of HDR processing.

The study contains description and illustrations of global tone mapping operators, followed by a discussion of exposure and automatic eye adaptation mechanism. Subsequently, two local tone mapping techniques were examined: one utilizing a bilateral filter and the other employing exposure fusion. The theoretical part is concluded with a section about the bloom effect.

Prior to starting work on the demonstrational application it was necessary to study Vulkan API, which was a time-consuming and complicated process. During the development of the application, the challenge arose which was to come up with a robust structural layer around Vulkan API that would allow for selection and configuration of multiple HDR post-processing effects. Another challenge was to utilize the computational capabilities of modern GPUs for implementation of HDR post-processing effects, which involved writing an extensive amount of compute shaders.

The resulting application should be viewed as a 3D rendering engine that can be used for exploring different HDR techniques. The application can be extended relatively easily by implementing, for example, another post-processing effect. Extensive UI would allow the user to display any parameters of their newly added effect, for example, for debug purpses.

My work, obviously, doesn't come without limitations. There are still many improvements that I have left out due to time restrictions or general lack of extensive experience in the topic. First, it would be a great improvement to incorporate the Physically Based Rendering (PBR) rendering pipeline as it is the standard of modern 3D rendering engines. In the context of further studies on the topic of HDR it would be useful to explore the ways to improve performance and visual quality of exposure fusion effect, specifically in the context of real-time rendering. As the general aim of further work on this topic I would call finding a way of interconnecting all the described HDR effects in a single complete pipeline to achieve a realistic visual appearance of the scene under different lighting conditions. Specifically that would require improving and correctly configure the eye adaptation effect and to connect it seamlessly with the bloom and tone mapping techniques.

Bibliography

[1] *ACES. Academy of Motion Picture Arts and Sciences* online. December 2014. Available at: https://www.oscars.org/science-technology/sci-tech-projects/aces. [cit. 2024-05-03].

[2] Adams, A. and Baker, R. *The Ansel Adams Photography Series*. Little, Brown andCompany, 1983.

[3] Blanco, V. *Vulkan Guide* online. 2020. Available at: //www.vkguide.dev/. [cit. 2023-07-09].

[4] Boitard, R.; Pourazad, M. T.; Nasiopoulos, P. and Slevinsky, J. Demystifying High-Dynamic-Range Technology: A new evolution in digital media. *IEEE Consumer Electronics Magazine*, october 2015, vol. 4, no. 4, p. 72–86. ISSN 2162-2256.

[5] Chiu, K.; Herf, M.; Shirley, P.; Swamy, S.; Wang, C. et al. Spatially nonuniform scaling functions for high contrast images. In: *Graphics Interface*. Canadian Information Processing Society, 1993, p. 245–245.

[6] Christensen, A. *LearnOpenGL: Guest Articles: Physically Based Bloom* online. 2022. Available at: https://learnopengl.com/Guest-Articles/2022/Phys.-Based-Bloom. [cit. 2023-24-08].

[7] Courrèges, A. *GTA V - Graphics Study* online. November 2015. Available at: http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/. [cit. 2024-17-02].

[8] Courrèges, A. *DOOM (2016) - Graphics Study* online. September 2016. Available at: http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/. [cit. 2024-17-02].

[9] Cox, S. *Exposure Value (EV) Explained - Plus EV Charts* online. December 2019. Available at: https://photographylife.com/exposure-value. [cit. 2023-01-09].

[10] Duiker, H.-P. and Borshukov, G. *Filmic Tonemapping and Color In Games* online. 2006. Available at: http://duikerresearch.com/2015/09/filmic-tonemapping-ea-2006/. [cit. 2024-17-02].

[11] Durand, F. and Dorsey, J. Fast bilateral filtering for the display of high-dynamic-range images. In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, July 2002, p. 257–266. SIGGRAPH '02. ISBN 9781581135213. Available at: https://doi.org/10.1145/566570.566574.

[12] Durand, F. and Dorsey, J. *Fast Bilateral Filtering for the Display of High-Dynamic-Range Images: Presentation Slides* online. 2002. Available at: https://people.csail.mit.edu/fredo/PUBLI/Siggraph2002/BilateralSlides.pdf. [cit. 2023-21-08].

[13] Guy, R. and Agopian, M. *Physically Based Rendering in Filament* online. Available at: https://google.github.io/filament/Filament.html. [cit. 2023-01-09].

[14] Hable, J. *Uncharted 2: HDR Lighting* online. 2010. Available at: https://gdcvault.com/play/1012459/Uncharted-2-HDR. [cit. 2024-17-02].

[15] Hennessy, P. *Implementing a Physically Based Camera: Understanding Exposure* online. November 2014. Available at: https://placeholderart.wordpress.com/2014/11/16/implementing-a-physically-based-camera-understanding-exposure/. [cit. 2024-19-02].

[16] Hessel, C. An Implementation of the Exposure Fusion Algorithm. *Image Processing On Line*, november 2018, vol. 8, p. 369–387. ISSN 2105-1232. Available at: https://www.ipol.im/pub/art/2018/230/.

[17] Jimenez, J. *Next Generation Post Processing in Call of Duty: Advanced Warfare* online. September 2014. Available at: https://www.iryoku.com/next-generation-post-processing-in-call-of-duty-advanced-warfare. [cit. 2023-24-08].

[18] Jooma, N. *CS129: Computational Photography: Project 5* online. Available at: https://cs.brown.edu/courses/cs129/results/proj5/njooma/. [cit. 2023-22-08].

[19] Kelly, C. *The Essential Guide to Color Spaces* online. February 2020. Available at: https://blog.frame.io/2020/02/03/color-spaces-101/. [cit. 2024-06-03].

[20] Krawczyk, G.; Myszkowski, K. and Seidel, H.-P. Perceptual effects in real-time tone mapping. In: *Proceedings of the 21st Spring Conference on Computer Graphics*. New York, NY, USA: Association for Computing Machinery, May 2005, p. 195–202. SCCG '05. ISBN 9781595932044. Available at: https://doi.org/10.1145/1090122.1090154.

[21] Mertens, T.; Kautz, J. and Van Reeth, F. Exposure Fusion. In: *15th Pacific Conference on Computer Graphics and Applications (PG'07)*. October 2007, p. 382–390. ISSN 1550-4085. ISSN: 1550-4085.

[22] Narkowicz, K. *ACES Filmic Tone Mapping Curve* online. January 2016. Available at: https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/. [cit. 2024-05-03].

[23] Narkowicz, K. *Automatic Exposure* online. January 2016. Available at: https://knarkowicz.wordpress.com/2016/01/09/automatic-exposure/. [cit. 2023-29-08].

[24] Opsenica, B. *Automatic Exposure Using a Luminance Histogram* online. April 2019. Available at: https://bruop.github.io/exposure/. [cit. 2023-29-08].

[25] Overvoorde, A. *Vulkan Tutorial* online. 2016. Available at: https://vulkan-tutorial.com/. [cit. 2023-07-09].

[26] Paris, S. A gentle introduction to bilateral filtering and its applications. In: *ACM SIGGRAPH 2007 courses*. New York, NY, USA: Association for Computing Machinery, August 2007, p. 3–es. SIGGRAPH '07. ISBN 9781450318235. Available at: https://doi.org/10.1145/1281500.1281604.

[27] Pečiva, J. *Seriál Tutoriál Vulkan* online. July 2021. ISSN 1212-8309. Available at: https://www.root.cz/serialy/tutorial-vulkan/. [cit. 2023-08-09].

[28] Reed, N. *Artist-Friendly HDR With Exposure Values* online. June 2014. Available at: https://www.reedbeta.com/blog/artist-friendly-hdr-with-exposure-values/. [cit. 2023-29-08].

[29] Reinhard, E.; Stark, M.; Shirley, P. and Ferwerda, J. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, july 2002, vol. 21, no. 3, p. 267–276. ISSN 0730-0301. Available at: https://dl.acm.org/doi/10.1145/566654.566575.

[30] Salih, Y.; Md Esa, W. bt.; Malik, A. S. and Saad, N. Tone mapping of HDR images: A review. In: *2012 4th International Conference on Intelligent and Advanced Systems (ICIAS2012)*. June 2012, vol. 1, p. 368–373.

[31] Sellers, G. and Kessenich, J. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. 1st editionth ed. Boston: Addison-Wesley Professional, october 2016. ISBN 9780134464541.

[32] Tardif, A. *Adaptive Exposure from Luminance Histograms* online. Available at: https://www.alextardif.com/HistogramLuminance.html. [cit. 2023-01-09].

[33] Tomasi, C. and Manduchi, R. Bilateral filtering for gray and color images. In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. January 1998, p. 839–846.

[34] Vries, J. de. *LearnOpenGL: Advanced Lighting: Bloom* online. 2014. Available at: https://learnopengl.com/Advanced-Lighting/Bloom. [cit. 2023-24-08].

[35] Vries, J. de. *LearnOpenGL: Advanced Lighting: Shadows: Point Shadows* online. 2014. Available at: https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows. [cit. 2024-07-04].

[36] Willems, S. *Vulkan Examples: texturecubemap* online. Available at: https://github.com/SaschaWillems/Vulkan/blob/master/examples/texturecubemap/texturecubemap.cpp. [cit. 2024-07-04].

[37] Willems, S. *Vulkan Examples* online. 2023. Available at: https://www.saschawillems.de/creations/vulkan-examples/. [cit. 2023-08-09].

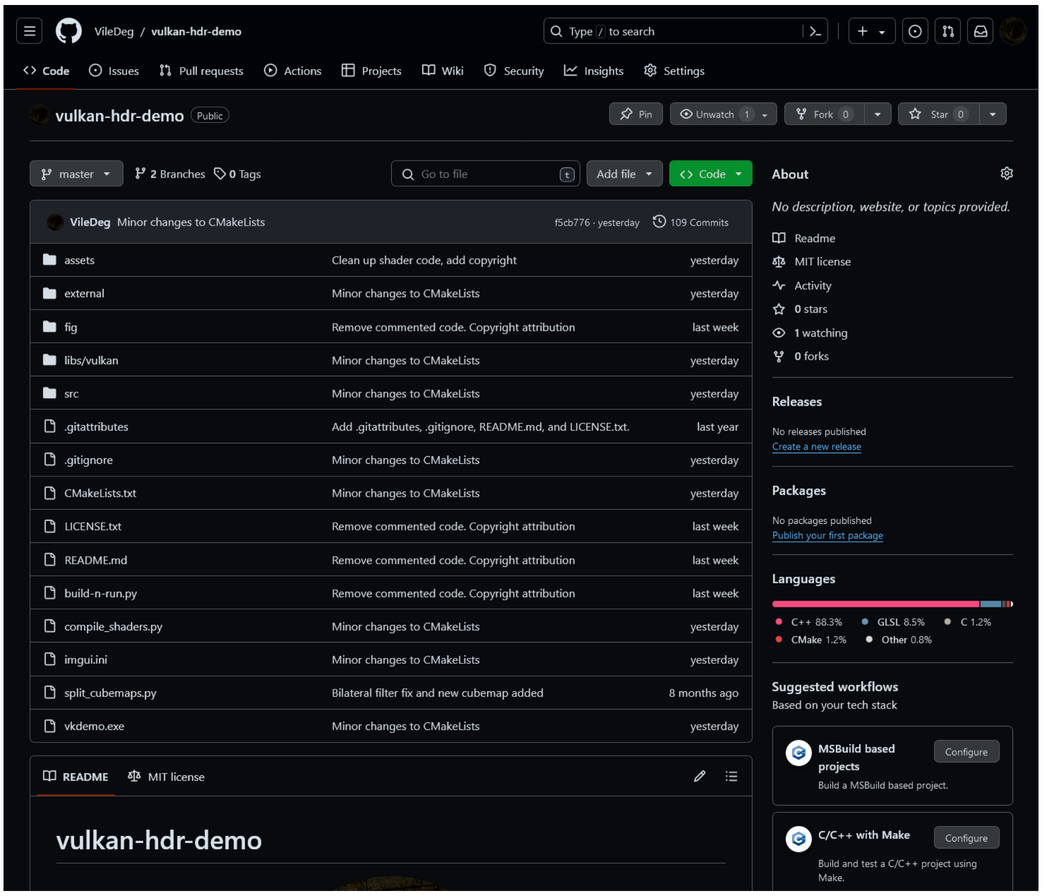
[38] Wronski, B. *Separate your filters! Separability, SVD and low-rank approximation of 2D image processing filters* online. February 2020. Available at: https://bartwronski.com/2020/02/03/separate-your-filters-svd-and-low-rank-approximation-of-image-filters/. [cit. 2024-07-04].

Appendix A

Published repository

Repository containing the demo application (`vulkan-hdr-demo`) is public and accessible on the internet at: https://github.com/VileDeg/vulkan-hdr-demo/tree/master. It is public and accessible to anyone under `MIT license`. Figure A.1 contains a screenshot of `vulkan-hdr-demo` GitHub public repository page.

The repository is also accessible under `Vulkan FIT` organization: https://github.com/Vulkan-FIT/vulkan-hdr-demo.


VileDeg / vulkan-hdr-demo

Type / to search

Code · Issues · Pull requests · Actions · Projects · Wiki · Security · Insights · Settings

vulkan-hdr-demo Public

Pin · Unwatch 1 · Fork 0 · Star 0

master · 2 Branches · 0 Tags · Go to file · Add file · Code

VileDeg Minor changes to CMakeLists · f5cb776 · yesterday · 109 Commits

assets	Clean up shader code, add copyright	yesterday
external	Minor changes to CMakeLists	yesterday
fig	Remove commented code. Copyright attribution	last week
libs/vulkan	Minor changes to CMakeLists	yesterday
src	Minor changes to CMakeLists	yesterday
.gitattributes	Add .gitattributes, .gitignore, README.md, and LICENSE.txt.	last year
.gitignore	Minor changes to CMakeLists	yesterday
CMakeLists.txt	Minor changes to CMakeLists	yesterday
LICENSE.txt	Remove commented code. Copyright attribution	last week
README.md	Remove commented code. Copyright attribution	last week
build-n-run.py	Remove commented code. Copyright attribution	last week
compile_shaders.py	Minor changes to CMakeLists	yesterday
imgui.ini	Minor changes to CMakeLists	yesterday
split_cubemaps.py	Bilateral filter fix and new cubemap added	8 months ago
vkdemo.exe	Minor changes to CMakeLists	yesterday

README · MIT license

vulkan-hdr-demo

About

No description, website, or topics provided.

Readme
MIT license
Activity
0 stars
1 watching
0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

● C++ 88.3% ● GLSL 8.5% ● C 1.2%
● CMake 1.2% ● Other 0.8%

Suggested workflows
Based on your tech stack

MSBuild based projects · Configure
Build a MSBuild based project.

C/C++ with Make · Configure
Build and test a C/C++ project using Make.

Figure A.1: Screenshot of `vulkan-hdr-demo` GitHub public repository page.

Appendix B

Copyright attribution

Following is the list of files and folders of `vulkan-hdr-demo` application that may contain copyrighted material. The files/folders are listed along with the source of copyrighted material.

- `external/` - folder contains external libraries along with their license files.
- `assets/` - folder contains downloaded resources along with the license files.
- `src/assets.cpp` - may contain parts of code from `https://github.com/vblanco20-1/vulkan-guide` (`MIT license`).
- `src/initialization.cpp` - may contain parts of code from `https://github.com/pc-john/VulkanTutorial/` (`MIT license`).
- `src/memory.cpp` - may contain parts of code from `https://github.com/SaschaWillems/Vulkan/tree/master/examples/dynamicuniformbuffer` (`MIT license`).
- `src/model_loader.cpp` - may contain parts of code from `https://github.com/tinyobjloader/tinyobjloader` (`MIT license`).
- `src/swapchain.cpp` - may contain parts of code from `external/imgui/imgui_impl_vulkan.h` (`MIT license`).
- `src/types.h` - may contain parts of code from `https://github.com/vblanco20-1/vulkan-guide` (`MIT license`).
- `src/vk_descriptors.h/cpp` - is based on `https://github.com/vblanco20-1/vulkan-guide` (`MIT license`).
- `src/vk_initializers.h/cpp` - is based on `https://github.com/vblanco20-1/vulkan-guide` (`MIT license`).
- `src/vk_utils.h/cpp` - is based on `https://github.com/SaschaWillems/Vulkan/blob/master/base/VulkanTools.cpp` (`MIT license`).

The demo application itself (`vulkan-hdr-demo`) is licensed under `MIT license` (`https://opensource.org/license/mit`).

Appendix C

Contents of the attached storage medium

- `assest/` - Assets required for `vulkan-hdr-demo` to run.
- `docs/` - Thesis and documentation:
  - `xgonce00-vk-hdr.pdf`- This bachelor's thesis document.
  - `src/` - Sources required to build this bachelor's thesis document (`xgonce00-vk-hdr.pdf`).
- `external/` - External source used to build `vulkan-hdr-demo`.
- `libs/` - External libraries for `vulkan-hdr-demo`. E.g. Vulkan lib (in case not found on current device).
- `linux-build/vkdemo` - Linux executable of `vulkan-hdr-demo`.
- `scripts/` - Scripts that may be useful for building the project and other tasks.
- `src/` - Source code of the demo application (`vulkan-hdr-demo`).
- `CMakeLists.txt` - CMake file used to configure project for `vulkan-hdr-demo`.
- `imgui.ini` - ImGUI's `.ini` configuration file.
- `LICENSE.txt` - License of `vulkan-hdr-demo`.
- `README.md` - README.md file with description of `vulkan-hdr-demo` and build & run instructions.
- `vkdemo.exe` - Windows executable of `vulkan-hdr-demo`.