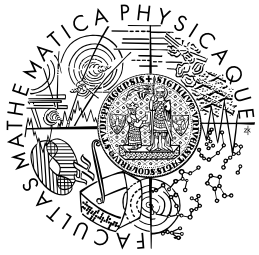


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Michal Wirth

Advanced HDR image viewer

Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Ing. Jaroslav Křivánek, Ph.D.

Study programme: Informatics

Study branch: Software Systems

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Advanced HDR image viewer

Author: Michal Wirth

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education

Abstract: The primary purpose of this thesis is to determine criteria for a high-dynamic range (HDR) image viewer accented by computer graphics artists and other users who work with HDR images produced by physically-based renderers on a daily basis. Also an overview of already existing solutions is present. Based on both of them, a new HDR viewer is designed and implemented giving an emphasis on its memory and performance efficiency.

For these purposes two alternative image data layouts, Array-of-Structures (AoS) and Structure-of-Arrays (SoA), are discussed and their impact is measured on the speed of an algorithm for changing image saturation which has been selected as a representative part of whole tone mapping process of the viewer. It has turned out that the latter type of layout allows the algorithm to run about 3 times faster or more under the conditions of a defined testing environment.

The thesis has two main contributions. First it gives the above users a tool which could help them when working with HDR images. Second it indicates that there may be a potential of significant speed-up of implementations of tone mapping algorithms.

Keywords: HDR, EXR, image viewer, tone mapping

I would like to thank my supervisor, Jaroslav Křivánek, for all the useful comments, remarks and engagement he has kindly provided me. I am also grateful to Michal Prokš for all the help and technical support.

I dedicate this thesis to my parents and to my Hanička because of their endless patience with me. Thank you.

Contents

Introduction **3**

1 Preliminaries **6**

1.1 Dynamic range 6

1.2 High-dynamic range 7

1.3 OpenEXR format 8

1.4 Alternative formats 10

1.4.1 Radiance 11

1.4.2 Portable Floatmap 11

1.4.3 Tag Image File Format 11

2 Related work **13**

2.1 HDRView 13

2.2 qt4Image 14

2.3 exrdisplay 15

2.4 HDRSee 15

2.5 LizardQ Viewer 16

2.6 Moonlight HDR Viewer 17

2.7 FastPictureViewer Professional 18

2.8 IrfanView 19

2.9 XnView 19

2.10 Photosphere 20

2.11 bracket 21

2.12 Panorado 23

3 Problem analysis **24**

3.1 Data producers 24

3.2 Target audience 26

3.2.1 Common users 26

3.2.2 Power users 29

3.2.3 Render farms 30

4 Viewer implementation **31**

4.1 Custom functionality 31

4.2 Corona Renderer means 32

4.3 Graphical framework 35

4.4 Image representation 36
4.5 Image processing 40

5 Evaluation **46**
5.1 Results 47

Conclusion **49**

Bibliography **51**

List of Abbreviations **53**

Introduction

For centuries, painters and photographers have been trying to realistically capture and convey real-world scenes that they see with their own eyes. This is in fact hard to do. Human eye has an extraordinary capability to perceive wide range of light intensities (high contrast), and their ability to adapt pushes their limits even further [1]. On the contrary, capabilities of even today's imaging technologies are limited in this sense. Paints work with reflectance hence any painting or printed photograph can hardly display full range of intensities of the original scene. Progress in the field of digital camera sensors, made in the last two decades, is remarkable, but still they are far from the capabilities of the human eyes in terms of the dynamic range they can capture. Also various computer displays and television screens suffer from the same problem, but from the opposite point of view. This all means it highly depends on skills and experience of the artist, how well he or she can utilize the narrow imaging possibilities to present the original, high-dynamic range (HDR) reality. For some scenes it can be easy, for other it can be much harder.

Let us imagine for a while how the situation would look like in an ideal world. With a sensor having the same capabilities as the human eye, and with a corresponding display allowing to light up each single pixel independently of the others, using a wide range of light intensities, it would allow us to capture any scene and present it in an "uncompressed" form to other people. In such world there would be no problem for example to take a single photograph from inside of a cathedral and see all the details of bright stained glass windows and simultaneously details of dark corners. Furthermore, between capturing and displaying such photograph it would be possible to do some post-processing which would not be affected by any contrast compression common in standard imaging.

In reality this is something where the current development in those fields tends to go, at least partially. Generally maximum contrast of both sensors and displays is being extended allowing to capture and display more visual information in one shot. Such technologies are also becoming reasonably cheap so it can be expected that they will soon spread into common use[1]. Furthermore, taking a photograph is not the only way to acquire HDR images. They can be of course created artificially, i.e. rendered directly on a computer by the means of physically-based

[1]For example, see `http://www.lg.com/us/experience-tvs/hdr/dolby-vision` or `http://clairpixel.co.kr/eng/main/hdr.html`.

light transport simulation. Therefore, it is important to concern oneself with the viewing and processing of such images. This is what the thesis addresses.

Goals

A first goal of this thesis is to explore the current situation around publicly available software primarily oriented towards viewing HDR images. The software doesn't have to be necessarily freeware or open source. Applications that have to be purchased shall be also considered. The main goal is then to learn from possible flaws of the presented HDR viewers and based on them design and develop a new one. It shall not be just an ordinary viewer and nothing more. It shall also introduce certain post-processing qualities in the sense of high-dynamic range imaging (HDRI) that computer graphics (CG) artists could appreciate within their workflow. Similarly, it would be desirable to draw up and implement some functionality supporting distributed rendering (DR) which is an important CG sector. A cooperation with the company Render Legion, producer of Corona Renderer, a world-class rendering software, shall help to achieve these particular goals.

Another goal is purely technical. Operations with HDR images often consumes considerable amount of memory. This thesis will try to answer a question whether it may pay off, from both memory and performance points of view, to arrange HDR image data in a way generally known as a Structure-of-Arrays (SoA) or in a way that is usually easier to implement, i.e. as an Array-of-Structures (AoS).

Thesis outline

The thesis itself is organized into several chapters as follows.

The first chapter provides the reader with some important definitions and concepts that relate to the thesis. We introduce the concepts of dynamic range of an image, high-dynamic range (HDR) images, the OpenEXR format for storing them as well as some other alternative formats.

In the second chapter we give an overview of the state of the art in HDR viewers that are currently publicly available. Both free and commercial software is included.

Next there is the third chapter with problem analysis discussing why a new HDR viewer would be beneficial, for whom it should be designed, and what features it shall offer.

The fourth chapter describes the implementation process of the viewer. We discuss what had to be implemented, what could be acquired elsewhere, what choices had to be made. It also contains a discussion of a suitable HDR image representation and its processing.

The fifth chapter provides a performance evaluation of different approaches to implementing a representative algorithm for changing image saturation. The approaches use two alternative HDR image data arrangements described in the previous chapter. The approaches are measured using a defined methodology in two different environments. Measured results are interpreted.

The conclusion presents an overview of goals that has been achieved in this thesis and some directions of the viewer's future development.

1. Preliminaries

This chapter introduces definitions of some basic terms and concepts that relate to this thesis. It helps to better understand what HDR images are and how we can store them into files. This information is essential to be able to describe the viewer.

1.1 Dynamic range

To talk about HDR we need to first understand what dynamic range actually is. Bloch in his monograph defines it as a ratio between a smallest change and a largest change in brigthness within an image [2]. This is why we sometimes talk about a contrast ratio instead of a dynamic range. For example, an image with dynamic range 1000:1 means that the highest contrast contained within the image is thousand times higher than the smallest one there. As Bloch explicitly points out, size of dynamic range always depends on two factors – not only on the maximum contrast, but also on the smallest contrast [2]. Hence for example if a manufacturer of some camera sensor would like to increase its dynamic range, it should be sufficient to make the sensor capable of differentiating finer changes in intensity of incoming light.

Such a scale for measuring dynamic range is linear. The ratio refers to a difference in light intensity [2]. By doubling the intensity, the ratio would also double. This is not the only way to measure dynamic range. Probably a more practical approach is to measure the range by the number of exposure values.

Exposure value (EV) is a photographic scale referring to the amount of light coming through an aperture per unit of time and hitting the film or sensor [2]. It depends on the size of the aperture and on the shutter speed. It is well standardized. For example, value of 0 EV corresponds to a camera setting with an exposure time of 1 second and using f/1.0 relative aperture size. The f/1.0 denotes an f-stop. The number is a ratio between lens focal length and used aperture diameter. Therefore the value of 0 EV also corresponds to 2 seconds and f/1.4 because the same amount of light hits the film coming through half-area aperture but for twice as long time. Increasing EV by 1 means half of exposure, either using shorter time or smaller aperture diameter (larger f-stop). It is a camera setting suitable for brighter scenes.

Now it is clear that the scale for measuring dynamic range in terms of EV is logarithmic. Each increase by 1 means twice as much light in the scene (to get

the same amount of exposure). So conversion between dynamic range measured as contrast ratio C and as E in terms of EV is clear: $E = \log_2(C)$, or $C = 2^E$ respectively. A dynamic range of 10000:1 corresponds to roughly 13.3 EV. For example, a television screen with contrast ratio 5000:1 differs from a screen with contrast 8000:1 by less than 1 EV.

1.2 High-dynamic range

Now we know what the dynamic range is. Generally HDR images are images that are capable of representing/storing images whose dynamic range is high. Regular images as we understand them today use 24 bits per pixel (ignoring an alpha for now), that is 8 bits for each color channel. We call them "true-color". Such images can represent only low-dynamic range (LDR) information because they can in the end store only 256 brightness levels. At most this is a dynamic range of only 8 EV. As a side note, human vision without considering the ability of an eye to further adapt has dynamic range about 14 EV [2]. This is a huge difference. So today's understanding of the term HDR is that it is everything above those 8 EV.

Let us think about how can we represent HDR images. At a first glance using more bits per color channel may help. For example, extending to 32-bit wide color channels would allow to store HDR images having a dynamic range of up to 32 EV. But it has some drawbacks. Let us inspect them on a situation when taking a photograph with a camera, see Figure 1.1. Generally, the response of a camera sensor is linear to the intensity of incoming light. But the dynamic range of that intensity measured in terms of EV is not linear, as we already know. This would result into an unequal distribution of EV within all available pixel values. In our case we can represent 2^{32} different values (within a single pixel color channel) but most of them would be occupied by the brightest EVs that we only need for highlights, which are usually least frequent. On the other hand, shadows having only small EV are very common in photographs but they would have to "squeeze" into only a small range of available values. We would lose precision there. This is exactly the opposite of what we want. It could be solved by gamma correction [3, 4], as it is being done for regular images where gamma corrected pixel values are always "baked" into image files. By this approach the values would be used more reasonably. Furthermore, because the gamma correction curve is similar to a response curve of human vision, the pixel values would be actually linear with regard to perceived brightness, which is desirable.

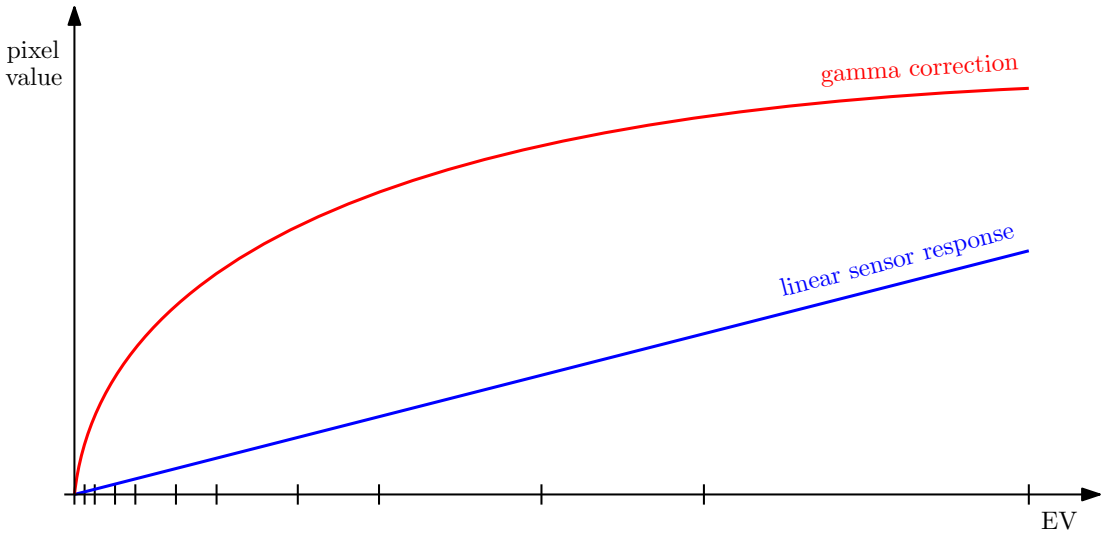


Figure 1.1: Gamma correction curve

But there is still a problem. We have just moved the shortcomings of regular images elsewhere. HDR images represented by this approach, i.e. using only wider integer pixel values, still have some minimum and maximum limit like regular images. This is a problem if such HDR image would be edited. Let us consider an operation of darkening the image. We would probably lose information while mapping pixel values for example from range 0-1000 (still considering a single pixel color channel) into a narrower range 0-100. This is because we cannot represent anything between neighbouring values – there is nothing between 24 and 25 etc. in our representation. This is something we would like to avoid in HDRI.

A solution for these problems is obvious. If we used a floating-point number representation [5] for pixel values instead of integers, we would not lose information there. Furthermore, this representation of numbers has a higher precision for lower values than for higher values and so we will not need to "bake" in the gamma correction either. HDR images could just store raw values which is also convenient for editing purposes. The gamma correction necessary to match human perception could be just postponed to a displaying phase. To sum it up, any seriously meant HDRI requires HDR images using floating-point numbers for storing pixel values of each color channel.

1.3 OpenEXR format

OpenEXR, or just EXR for short, is currently probably the most popular format for storing HDR images into files. It has most advanced features [6], it is well

documented, and its author, company Industrial Light & Magic (ILM)[1], provides a good open-source library written in C++ for it[2]. The library can read HDR images stored in this format and write them back [7]. Originally the format has been developed for the needs of the movie industry, but today it is being used extensively also in other fields. Any reasonable HDR image viewer shall support it. The format divides all image data in two parts: channels and attributes.

HDR images in EXR can contain an arbitrary number and combination of image channels [6]. It purely depends on a particular software how they would be interpreted. For this purpose each channel is uniquely identified by its name which can be any non-empty string. No special rules apply for them but at least R, G, and B channels should be present within every image. They are expected to hold red, green, and blue pixel values, as their names suggest. In rendering community such triplets of channels (quadruplets, if an alpha channel is being considered too) are referred as render elements. In this special case with the R, G, and B the render element is referred to as a "beauty" element.

Each channel can have a different format, i.e. type of values it stores [6]. They could be either 32-bit unsigned integers, or 16-bit or 32-bit floating-point numbers (denoted familiarly as halfs and floats). The R, G, and B channels usually store values of the latter type, as it has been explained in the previous section.

All channels of a single HDR image in EXR have virtually same dimensions (width and height). But each of them can have different vertical and horizontal sampling rate, although with some restrictions [6]. This could be used to store sparse data for which storing them in every single pixel would be a pointless overhead. HDR images using color spaces where the luminance component of each pixel is separated from its chroma can serve as an example where the different sampling can pay off. It is because the chroma component is being often undersampled in such color spaces.

As for the attributes in EXR, they can be also of arbitrary names and various types. The same concept as for channel names is valid also for attribute names. Again it depends on a software how it would interpret them. But there are some attributes which every HDR image in EXR must have [6]. At minimum these are namely:

- ***displayWindow*, *dataWindow***

  The EXR format defines an image as an axis-parallel rectangle in pixel

[1] See `http://www.openexr.com/`.

[2] Available at `https://github.com/openexr` or `http://openexr.com/downloads.html`.

space [6]. The *displayWindow* attribute specifies this rectangle through pixel coordinates of its top-left and bottom-right inner corner. The format is designed in a way that the image does not have to necessarily contain all the data for the whole *displayWindow*. Or other way round it may contain data beyond it. For this purpose there is also the *dataWindow* attribute which similarly specifies a rectangle where the image data are available. The data are stored using the channels that have been described above.

- ***pixelAspectRatio***
  It tells with which aspect ratio is the image, specified by the *displayWindow* attribute, expected to be actually viewed on a display device. The value of this attribute is usually 1.0, meaning to display the image as it is.

- ***compression***
  The image data could be either stored in an uncompressed form, or using one of several lossless or lossy compression methods. For all available methods and their descriptions see the official technical introduction [6]. Fortunately the provided library can transparently read and write pixel values irrespective of the used compression method.

- ***lineOrder***
  Tells whether image scan lines are in file stored upwardly or downwardly. The EXR format also supports tiled images. The library provides a uniform interface to these variants.

- ***screenWindowWidth*, *screenWindowCenter***
  Both attributes describe a perspective projection (camera obscura setting) that has produced the image [6]. This is something that may not be known. In such cases default unit width and center at origin shall be used.

In EXR anybody is allowed to introduce new attributes for new purposes. The format supports plenty of types for that: integers, floating-point numbers, strings, both 2D and 3D vectors, rectangles, matrices, etc.

1.4 Alternative formats

The EXR format is very versatile, sophisticated, robust, and can be used in many scenarios. This is why it is so popular today. But it is not the only one format for storing HDR images.

1.4.1 Radiance

For example, there is also the Radiance format, usually abbreviated as HDR, RGBE, or PIC. It is an older format invented by Greg Ward [8]. He has come up with it for his software called Radiance. This software has actually introduced physically-based rendering to the world. This was really revolutionary in its time. The format uses a nice trick to encode floating-point pixel values in a space-saving manner [2]. Each color channel of a pixel uses only 8 bits to store its mantissa. In another 8 bits there is an exponent shared by all three channels. This is why it is being referred as RGBE. This system offers an enormous dynamic range of over 76 orders of magnitude which is about 253 EV [8]. This may sound like a good thing but it is actually a drawback. There is no real usage for such a high range and one can say that this is a literally wasting of space (most of exponent bits are always zeros) at the expense of accuracy which is not so high [2]. The EXR format is much better in this sense. Because of the wasting, the format offers Run-length Encoding (RLE) compression. It is not surprising that it is usually very effective in this case.

1.4.2 Portable Floatmap

Another way how HDR images can be stored is the Portable Floatmap (PFM) format. It is very simple, like other formats from the "portable" family – Portable Pixmap (PPM), Portable Graymap (PGM), etc. It shares the same concept. A file containing some HDR image starts with a simple header specifying mainly dimensions of the image. After that follows a series of 32-bit floating-point number triplets written in a binary mode. Each triplet forms a single pixel with values of its red, green, and blue channel. That is all. It is very easy to write a code that can read or write such HDR images. This is why it is popular despite the fact that HDR images stored in this format usually occupy a lot of space.

1.4.3 Tag Image File Format

Probably anything can be stored in the Tag Image File Format (TIFF) and HDR images are not an exception. It could be described as a wrapper or container for images. It supports many color spaces, compression methods, and bit depths. Internally all the data are being labeled with "tags". This is how applications could know what to do with them [2]. It is a very versatile format. But this also makes it hard for software developers to add support for it. In practice this results into a lot of incompatibilities and there is no guarantee that an image

saved into TIFF in one application would be able to open in another.

Actually there are more than one way how HDR images could be stored in TIFF files. One of them could be a similar "dumping" approach like in PFM. This is usually referred as FP-TIFF, TIFF32, or TIFF Float [2]. But it is not widely used. Slightly more known are HDR images stored in the so called LogLuv TIFF. Again, this is a format that has been designed by Ward where he has attempted to solve the issues of the RGBE format and to establish a new industry standard. The second goal has been never achieved. As he states [8] there are three variants of LogLuv TIFF. They basically differ in the number of bits dedicated to individual pixel components. The format itself is significantly based on features of human vision. It operates in CIE-Luv color space where any given value change shall uniformly result into the same perceptual difference [9]. The format works as follows. It separates luminance and chroma components, logarithmically encodes the luminance and quantizes the chroma using terms like a "just noticable difference" [8, 2].

All the above formats for storing HDR images are probably the most widespread ones. But they are not the only ones. More extensive format overview can be found for example in monographs by Reinhard et al. [3], or by Bloch [2].

2. Related work

This chapter contains an overview of twelve computer programs that can be considered primarily as HDR image viewers. Each of them is briefly characterized – i.e. on which platforms it can run, what features and functionalities it can offer, what advantages or disadvantages it has, whether it is free or must be purchased. The purpose of the overview is to provide a rough idea of how the situation on the "market" of HDR viewers currently looks like.

2.1 HDRView

HDRView is one of the famous programs from the research lab of Paul Debevec et al. It is a truly pioneering tool in the HDRI area. Although it is deprecated by now and has been abandoned by its authors years ago, it is still working even under the newest version of Windows. Unfortunately the project's official website together with all documentation and help are no longer available. But it is still possible to retrieve the viewer as a portable executable elsewhere. This is because its last version 1.2 has been released as a freeware for non-commercial use.

The viewer allows to open HDR images stored in RGBE or PFM. Unfortunately the most important EXR format is not supported. The Program can also write the opened HDR image back to a file but only the RGBE format can be used for that. In addition, current view of the HDR image with possibly compensated exposure can be saved as a regular image in the Windows Bitmap (BMP) format. Exposure compensation can be increased or decreased by single unit increments.

The program provides only few other functions. It can perform 90 degree clockwise rotation, flip the image vertically or horizontally or display the stored pixel values. The view itself can be zoomed in or out at fixed steps. That is all.

Authors compensate the lack of other HDRI-related operations through a sister tool named HDRShop[1]. Old version 1.0 used to be offered as a freeware for non-commercial use. This is not available anymore and HDRShop in newer versions has to be purchased. One way or another, it is more an HDR editor than a viewer and therefore it isn't covered in more detail within this review.

[1] See `http://www.hdrshop.com/`.

2.2 qt4Image

Edgar Velázquez-Armendáriz, a former Ph.D. student at Cornell University, has developed and still keeps maintaining an open-source HDR viewer named qt4image. It is multi-platform and supports all three major operating systems, i.e. Windows, OS X and Linux. Nevertheless, the author provides only a Windows binary at the official website[2].

A nice feature of this viewer is its support of all mainstream HDR formats. It can read HDR images in the EXR, RGBE or PFM formats and also write them back in any of them. Therefore, the viewer can be used also as a file format converter. In addition, the current view could be always saved as a regular image in variety of formats, namely in the TIFF, BMP, PPM, Portable Network Graphics (PNG), or Joint Photographic Experts Group (JPEG) format.

Like in other viewers the view can be zoomed in or out at fixed steps. Another possibility is to zoom the view to fit current window size. Unfortunately, this can be done only as a one-shot operation and the program does not update zoom accordingly while resizing the window. No full-screen mode is available. The program offers quite a few functions in contrast of others. Exposure can be increased or decreased interactively by any EV. The same thing goes for gamma correction. All these are smooth and fast even for high-resolution HDR images, at least on the Windows platform.

Opened HDR image can be tone mapped by a tone mapping operator, specifically the global variant of the photographic operator published by Reinhard et al. [10]. Other tone mapping operators are not provided. The program can display both the stored and the tone mapped pixel values.

Researchers and developers of rendering algorithms surely appreciate the available binary operations with HDR images, namely image sum or division, and various image comparison operations (absolute difference, positive/negative difference, relative error, positive/negative relative error). A huge disadvantage is the unfriendliness of the user interface. The user is forced to select HDR images as operation operands in a file selection dialog over and over again. This can get pretty annoying, especially when the images are located in different folders.

Another disadvantage is a generally insufficient documentation of the program, e.g. it is unclear what some of the comparison operators listed above exactly do from a mathematical point of view.

The viewer itself is unable to do any batch processing of HDR images. For

[2]See `https://bitbucket.org/edgarv/hdritools`.

this purpose the author provides different but related command-line interface (CLI) tool named batchToneMapper. As its name suggests it allows to map tones (including eventual exposure compensation and gamma correction steps) in several HDR images at once. But the only available tone mapping operator is the same as for the viewer.

2.3 exrdisplay

Author of the EXR format produces also its own library allowing to read and write files in their format[3]. The library is written in C++, is multi-platform and published as an open-source software.

An integral part of the library's distribution is a modest HDR viewer known as exrdisplay. It serves mainly as an example showing how the library can be used. Hence, it does not support viewing HDR images in any other format than EXR.

Its capabilities are also very limited. It is possible to compensate exposure of the viewed HDR image by any positive or negative EV, to compress and clamp its dynamic range through special high and low knee functions and finally in case of need to defog the image, i.e. to compensate stray light which can sometimes occur due an imperfection of the used recording device. The viewer can also show the stored pixel values. Except for that, no other HDRI-related operations are available. The biggest disadvantage is the viewer does not support any kind of zooming. Viewing high-resolution HDR images is therefore very impractical.

2.4 HDRSee

The open-source program HDRSee, previously known as mhdrViewer, has been developed by Romain Pacanowski et al. at the French Institute for Research in Computer Science and Automation (INRIA) and at the University of Bordeaux. It is a unique HDR viewer which should run again on the Windows, OS X, and Linux platforms. It understands HDR images stored in the EXR and RGBE format.

As it is mentioned on the project's website[4], the most interesting feature is the viewer's design. Operations are GPU-accelerated through OpenGL – especially

[3] See `http://openexr.com/`.

[4] See `http://mhdrviewer.gforge.inria.fr/`.

all provided tone mapping operators, which are implemented as fragment shaders. Because of this they are fast and easily modifiable. Users can even implement and add their own operators. They are written in popular OpenGL Shading Language (GLSL) and compiled at run time as any other fragment shaders. For this purpose the project's website offers a useful tutorial and also a decent Application Programming Interface (API) documentation which may come in handy.

By default, the viewer offers several tone mapping operators. First of all there is a straightforward linear operator with a possibility to clamp dynamic range of the viewed HDR image. It is also possible to use the common photographic operator by Reinhard [11]. The viewer furthermore offers an operator by Drago [12] and another by Ward [13].

Each of them has its own set of parameters that can be set approximately by dynamic sliders or by entering an exact value on the keyboard. But the latter way unfortunately does not work well. It is sometimes impossible to write a desired value into the appropriate input box. In addition, not all the operators allow to set e.g. exposure compensation or gamma correction, which is unfortunate. They also provide different sets of information and statistics about the currently loaded image. As a result, the whole program appears inconsistent from a global perspective.

Another way to control the program is a CLI. Its syntax is not clear from the provided help and almost any wrong input makes the program crash. Nevertheless, it is the only way to save any tone mapped HDR image as a regular image. The PNG format is hard-coded for that.

The last feature worth mentioning is zooming. The viewer automatically zooms the currently loaded image according to its window size. It also allows to zoom in by clicking somewhere on the image. Sadly the quality of such zooming is extremely poor because HDR images that can be zoomed sharply in other viewers are blurred in HDRSee.

2.5 LizardQ Viewer

German company LizardQ GbR produces a simple freeware HDR viewer named symptomatically LizardQ Viewer[5] which is actually just a small part of their product – professional panoramic HDR camera system. Therefore, all the viewer's features and capabilities follow this aspect.

[5]See `https://www.lizardq.com/en/viewer/`.

The viewer is being distributed as a standalone executable that suffers from a surprisingly long startup, at least on the Windows platform. It should also run under the OS X. Sadly, the only HDR format which the program understands is an uncompressed variant of the RGBE format. RLE compressed variant is not recognized and the program fails to open such HDR images. Files in the EXR format cannot be opened at all.

Documentation and program help is minimal. E.g. the viewer provides a special feature named "projection mode", but at first sight it is not clear what it actually does. Through a process of trial and error one can find it changes the way how the viewer interprets and displays the viewed image – whether it expects that the image is stored in equirectangular (spherical) geometry or not. Similar situation is with the only tone mapping operator that the viewer offers. There is no information or at least a hint which of the known operators it could be.

Opened image can be viewed in a full-screen mode and zoomed in or out in a discrete fashion. Zooming sometimes behaves a bit strangely and inconsistently, mainly while adjusting the window size. The program also offers so called auto-rotation feature which turns on automatic camera turn around while the view is in the projection mode mentioned above.

Sometimes it may come in handy that the viewer allows to display only a single selected RGB channel. Exposure compensation is standard; it can be increased or decreased repetitively by half f-stop, i.e. in fact by half EV, increments. Otherwise, no additional interesting HDRI-related operations are provided.

2.6 Moonlight HDR Viewer

This really simple HDR image viewer is being developed by Gregor Mückl as a part of a much larger open-source project named Moonlight—3D, which is, according to project's website[6], striving to be a modern modeling and animation tool.

The viewer itself is written in Java and works both on the Windows and the Linux platform as a standalone application without the need of any installation. It should support reading HDR images in the EXR or the RGBE format, but the latter one actually doesn't seem to work at all – neither RLE compressed nor uncompressed variant. Moreover, high-resolution HDR images are opened quite slowly.

[6] See `http://www.moonlight3d.eu/`.

Unfortunately, the program provides absolutely no functions or operations except for viewing the HDR image with any but vague amount of exposure compensation. The only other supported feature is zooming. You can zoom the view in or out at some predefined steps but nothing more.

There is also no user manual and the viewer has no help. But this is really not a big issue considering the lack of program capabilities.

2.7 FastPictureViewer Professional

This shareware image viewer for Windows is being actively developed by the Swiss company Axel Rietschin Software Developments[7]. By default, it cannot view any HDR images, but a special codec package from the same author can be purchased to add such ability. A single license for a bundle of both costs 49.99 USD. After installation of the codec package, the viewer is able to open HDR images in the EXR and RGBE formats. Moreover, the package also integrates directly into Windows Explorer and extends its thumbnailing capabilities to handle among other both mentioned formats. In fact, only uncompressed variant of the latter is well supported, although the official website claims the opposite. Still, the integration is an interesting and useful feature.

The viewer focuses primarily on the needs of photographers, not on HDRI and therefore it provides no HDRI-related operations except for calculating a histogram of luminance or any RGB channel. No exposure compensation or tone mapping operators are available. On the other hand standard viewing functionalities are at a higher level. E.g. zooming can be done in multiple ways according to the current window size, provided full-screen mode is capable of comfortable single-click magnification which may come in handy. Although some filesystem operations are currently possible, it is still not a full-fledged image organizer. A thumbnail mode will be added within the upcoming version 2.0 of the program.

The whole viewer design is smooth but its user interface is slightly non-standard. Nevertheless, it could be effectively controlled by a great amount of keyboard shortcuts and mouse gestures, so this could be an advantage after all. The official website offers a lot of tutorials and introductory videos on this topic.

[7] See `http://www.fastpictureviewer.com/`.

2.8 IrfanView

IrfanView, named after its author Irfan Skiljan, is a world-famous freeware image viewer for Windows with a wide user base[8]. By default, it already supports a huge number of graphics formats. A few others can be added via the official plug-in package. Among them are EXR and RGBE. The truth is that only the first one really works. HDR images in the RGBE format are not recognized and the viewer fails to open them.

Unfortunately the viewer handles HDR images in a naive way. Each one is immediately tone mapped right after its loading, which is in fact very slow in comparison to other HDR viewers, and any further operations are done as with any other regular image. This implies that any result made by the program can be saved only in a regular format. Several common ones are supported; e.g. JPEG, BMP, PNG, or Graphics Interchange Format (GIF).

As for operations, the viewer can adjust brightness, contrast, saturation, or color balance of the opened image. It is also possible to perform gamma correction, decrease color depth, convert the image to grayscale or negative, replace colors, swap color channels, sharpen or blur the image, and apply median filter to it. The image can be rotated left or right or by a custom angle, flipped horizontally or vertically, resized, and cropped automatically or manually. Several artistic effects are available, for example a red-eye reduction tool. But as mentioned above, none of these operations are done on the original HDR data. The viewer does not even offer such a basic feature like exposure compensation.

Almost all above operations can be made also in a batch for several images at once. Like for other features, IrfanView provides a decent user manual for that. The image itself can be viewed at full-screen and can be zoomed in or out or fitted into the viewer's window in many different ways. It is possible to view only a single selected RGB channel of the image as well. Exact values at some pixel can be displayed but only the tone mapped ones. The same goes for luminance or any RGB channel histogram.

2.9 XnView

A long-term rival image viewer to IrfanView is program XnView originally developed by Pierre-Emmanuel Gougelet, nowadays maintained by his company

[8] See `http://www.irfanview.com/`.

XnSoft[9]. As an image viewer it provides roughly the same functionality and a similar set of capabilities. Hence, it does not need such a detailed description.

One of the differences is that unlike IrfanView it is also a powerful image organizer. Various filesystem operations are available for that. Moreover, it is multi-platform and should run on Windows, OS X and Linux. Distribution conditions are same, i.e. XnView is released as a freeware for non-commercial use.

From all HDR images only those stored in EXR can be opened. But again, an additional official plug-in has to be installed. Unfortunately this does not add any HDRI-related operations and therefore XnView suffers from the same flaw as IrfanView; HDR images are immediately tone mapped and eventual operations are done as with any other regular image. Furthermore, result of the tone mapping step seems to be unnaturally underexposed and because the viewer does not offer any way to compensate exposure, overall experience of viewing HDR images is feeble.

2.10 Photosphere

Photosphere, created by Gregory Ward's Anyhere Software, is not only an HDR viewer but also an image browsing and cataloging tool[10]. It runs exclusively on the OS X platform and is provided for free. The program understands a variety of HDR formats, namely it can read and write HDR images in the EXR, RGBE, LogLuv TIFF or the JPEG-HDR format. Images in the Float TIFF format can be read too, but not written.

The program provides two ways to organize images. First it is the classic concept, i.e. using ordinary filesystem operations like renaming or moving. Here any performed image operations has to be manually saved to affect original files. The second way is to create special "catalogs" that are able to reference images across multiple directories. Single image can be involved even in several catalogs at once. This is possible because any changes made to some particular image do not actually alter the image itself but are stored within the catalog and are automatically and transparently applied while viewing the image. This aspect may not be obvious at first sight. Also, because Photosphere is unfortunately missing a user manual or help of any kind.

[9] See `http://www.xnview.com/`.

[10] See `http://www.anyhere.com/`.

Images could be browsed either in a thumbnail view or in a single image view. Each of them has its own window. Thumbnails for the first one are being generated on demand and cached with respect to a defined maximum of total cache size. The latter one allows switching among multiple images previously selected in the thumbnail window or display them automatically one by one as a slide show, perhaps even in a full-screen mode. The latter window also offers a special synchronized mode when the switching keeps various settings like current scroll position, zoom level etc. Ways of zooming are standard; the program allows to zoom in or out at fixed levels or fit the viewed image into the window.

Regarding HDRI-related operations, Photosphere offers several of them. It can do 90 or 180 degree rotations and flip images vertically or horizontally. These could be actually done even with multiple images at once. The program can also crop images, perform red-eye reduction, calculate luminance sum, calculate histogram of luminance or any RGB channel, display images in false colors according to their local luminance. Several overlapping HDR images can be stitched together into a single panorama. On the contrary, multiple regular images just with different amounts of exposure, i.e. the outcome of a technique generally referred as "bracketing", can be fused together and so create a new HDR image (image aligning, ghost and lens flare removal phases can be included).

The program also offers plenty of options to compensate exposure. It can increase or decrease EV by about 1/3 unit increments, turn on automatic exposure based on average luminance of some selected rectangle area or the same thing for a whole luminance histogram. Resulting HDR image can be saved at any time as a regular image in the JPEG or TIFF format.

As for user experience, the program uses several outdated graphical user interface (GUI) elements and hence it doesn't fit flawlessly into today's appearance of the OS X platform. It is also relatively unstable and seems to crash randomly while performing miscellaneous activities.

2.11 bracket

An alternative to Photosphere is the freeware program bracket created by Ahmet Oğuz Akyüz[11]. Unlike Photosphere, bracket should also run on Windows and Linux. Supported HDR formats are similar; bracket can read and write HDR images in the EXR, RGBE or the LogLuv TIFF format.

[11] See `http://www.ceng.metu.edu.tr/~akyuz/bracket/bracket.html`.

General concepts of both programs are similar too. Bracket is also a strong image management tool but images are being organized only using filesystem operations. No cataloging capabilities, like in Photosphere are offered. Its GUI is also a bit atypical and one can consider it outdated because it lacks a modern native look, but that is not much of a problem.

Like Photosphere, it can increase or decrease exposure of the viewed HDR image in a discrete fashion, make a 90 degree rotation, flip the image horizontally or vertically, crop the image or calculate a luminance histogram or any RGB channel histogram. In addition, it can display some other image information and statistics, e.g. the image's dynamic range. It can also show the stored pixel values or their transformation into the CIE-xyY color space, i.e. where the Y part measures the relative luminance. The image can be resized via several commonly used methods. The program can also perform gamma correction with any exponent and map tones by both operators of Reinhard et al. [10, 11]. Unfortunately it is not specified whether only the global or even the local variant is being used for that. Resulting image can be saved like in Photosphere as a regular image in the JPEG or TIFF format.

As for batch processing, the program is able to perform the rotating, flipping, resizing and cropping operation mentioned above repetitively for a whole group of selected HDR images. The program is also capable to merge several bracketed regular images into a single new HDR image. During the process it is able to reduce noise and align the input images, but more detail configuration is missing.

The currently viewed image can be zoomed in, out, or to fit well into the viewing window. It can be also displayed at full-screen.

Bracket attempts to speed image browsing up and therefore it caches all generated thumbnails. Their size is globally adjustable in the program's preferences. This has a positive impact while browsing folders that contain a greater number of high-resolution HDR images. On the contrary loading, resizing or perhaps tone mapping of such images is unfortunately still rather slow.

User experience also suffers because of another aspect. Some image operations warn the user that the original image file will be modified, but some operations does not. Such inconsistent behavior is confusing and can eventually result into a loss of data. Positive thing about bracket is that it provides a user manual explaining at least some its functionalities.

2.12 Panorado

Panorado, a Windows-only image viewer by Karl Maloszek's Simple Software, is among other also able to both view and organize HDR images. The official website claims that both EXR and RGBE formats should be supported[12], but actually not all HDR images in RGBE can be opened. It is because the program does not handle the RLE compressed ones. In any case, loading high-resolution HDR images is relatively slow, including generating thumbnails for them. This is especially problematic because thumbnails are not being cached by the viewer and hence browsing folders containing even a few such images is really tedious.

Offered HDRI-related features are modest. Furthermore, any changes made to the viewed image cannot be saved, neither back as an HDR image nor just as a regular image. The program is only able increase or decrease the image exposure or contrast in discrete steps, calculate its luminance histogram or show the stored pixel values including their EV equivalents. The image view can be zoomed in or out at fixed steps or automatically according to the current window size. Also, a full-screen mode is available but that is pretty much all. For example, no tone mapping operators are provided.

The program features a built-in user manual which gives a good explanation of almost all of the program's functionalities, i.e. not only of the HDRI-related ones. This is indeed helpful.

Panorado is a shareware application and its single license currently costs 21.00 EUR. Because of this and all of the disadvantages mentioned above it can be said that the program has a worse price–performance ratio, at least while working exclusively with HDR images.

[12] See `http://www.panorado.com/`.

3. Problem analysis

As can be seen in the previous section, qualities of presented HDR viewers vary and generally they provide only basic functionalities. This may be enough for HDR image browsing but it definitely cannot satisfy advanced users like CG artists. Such users won't be probably so much thrilled about a capability of some viewer to rotate or crop images. More useful for them would be rich possibilities of affecting tone mapping process or a possibility to alter an image with some advanced effect that benefits from the nature of HDR data itself. This area is marginalized by all the described viewers and hence there is a large space for improvement.

To design a viewer that will exploit and fit into the mentioned gap on the market, let us first find out who actually works with HDR images, how important these images are for them, what their needs are, and on the contrary what they do not need to do at all or not so often. Knowing at least some answers to these questions will help us identify key features that the viewer shall ideally have and establish a road map for its development. The road map is important because it is already clear that the space for inventions in this area is enormous and not all features that will come to mind would be realizable right away. Some may be harder to implement and users won't actually need them so desperately.

3.1 Data producers

Who actually produces HDR images? Basically images can be either captured by a camera, or synthesized by a computer. These two groups have actually only little in common. One of the common things is the data format. Both captured and synthesized HDR images can be stored e.g. as EXR files. Any HDR viewer capable of opening this popular format can be used to view both kinds of images. But their purpose is usually different and so are the operations that are being made with them.

Photographers often use the bracketing technique and fuse several photographs with different exposure time into a single HDR image. This image is usually not their final product. They mostly perform some tone mapping step (and one can say that there is an infinite number of tone mapping operators out there, either global or local ones) which reduces large overall contrast of the HDR image into a narrow range of a regular image [2]. It depends on the operator how it will be done. Often the result have a bit unrealistic artistic look. But that is something

the photographers usually want.

One way or another, it can be said that needs of photographers are already covered by the current state of available software. Even applications like Photosphere or bracket, presented in the viewers overview, support bracketing and hence can merge several photographs into one HDR image. They also offer some tone mapping possibilities. In addition, there are plenty of specialized applications for this purpose. Like Aurora HDR, easyHDR, or Photomatix Pro – although they are not HDR viewers in the common sense (they are more of HDR manipulating applications), they are all great[1] and worth mentioning in here. To sum it up, photographers will probably not appreciate creation of another HDR viewer because all the tools they need may already exists.

Now what about synthesized HDR images, especially photorealistic ones? Such images are mostly results coming out from a specialized rendering software, like Corona Renderer, V-Ray, Arnold, etc., which perform complex light transport simulations within some given 3D scene. The scene has to be modeled before by a CG artist using a modeling environment like Autodesk 3ds Max, Cinema 4D, SketchUp Pro, Blender, or some of many others. So far it sounds nice but it is necessary to realize that it is not that simple.

Unbiased rendering is a by design time-consuming process. This is because it is statistically correct and it intentionally introduces no systematic error – "bias" – into the process. If it is given enough time, the noise will vanish and all will eventually automatically converge to an expected result. This is something the artists like and want – to let the machines do the work without need to tweak everything here and there. But there is a price that has to be paid for that. It is the computation time. This is why various tricks exists to speed things up. One of them could be for example distributed rendering (DR) which splits the job among multiple rendering nodes running in parallel. In practice such dedicated systems are known as "render farms". Another trick could be a premature interruption of the rendering process and afterward reduction of any not yet vanished noise using some filters known from world of image processing.

Not only because of those reasons the rendering is not usually a final step for CG artist and some post-processing is required. This is where comes the hitch. None of the HDR viewers presented in the overview offers any seriously meant post-processing options. But actually it would be so much more convenient to let a renderer do its job and perform all desired post-processing in real-time while

[1] See the overview at `http://captainkimo.com/hdr-software-review-comparison/`.

viewing the result immediately after the rendering or any time later. This is the motivation behind the goals outlined within the Introduction section.

3.2 Target audience

For meaningful specification, what shall be such HDR viewer capable of doing and how shall it behave and look like, it is probably always the best to ask the intended users. This is why some external specialists has been invited to join a discussion about it. First of all it was the thesis supervisor, Jaroslav Křivánek. Next it was Ludvík Koutný, a skilled professional CG artist and hence a potential user of the discussed viewer. He is also very versed in user interface (UI) design. Other experts were from Corona Renderer developer team – Ondřej Karlík and Michal Prokš. They both have extensive knowledge of the needs of users of realistic renderers, which was very helpful. Together we have sketched a suitable proposal how the final viewer should probably look and behave. The first step in the design was an identification of the potential user groups. The software should aim at three kinds of users: common users, power users and render farms. This can be summed up as follows.

3.2.1 Common users

Basically from the discussion has emerged that CG artists would like to view HDR images produced by renderers and be able to apply some post-processing on them later at any time. They need to do it because just rendered images are not finished yet. For all fields of industry, where the rendering takes part in (architectural visualization, automotive, product presentation and commercials, movies, game development, ...), it is often required and necessary to further alter the images to give them more attractive look.

It cannot be expected that CG artists have generally some extensive technical knowledge. The more interactive, comfortable, and easy-to-use the viewer's UI would be, the better it would be for them. The interactivity is essential because then the artists can immediately see and feel all the changes as they make them. As for the comfort, it can be said that being comfortable equals to be known or expected, or at least predictable. This is also based on the discussion. Generally if users know something or have seen a similar concept elsewhere, they will consider it as comfortable and easy to use, which is our goal.

The UI of Corona Renderer itself can be considered as an example of a good UI design, and according to a recent research made by Render Legion its users

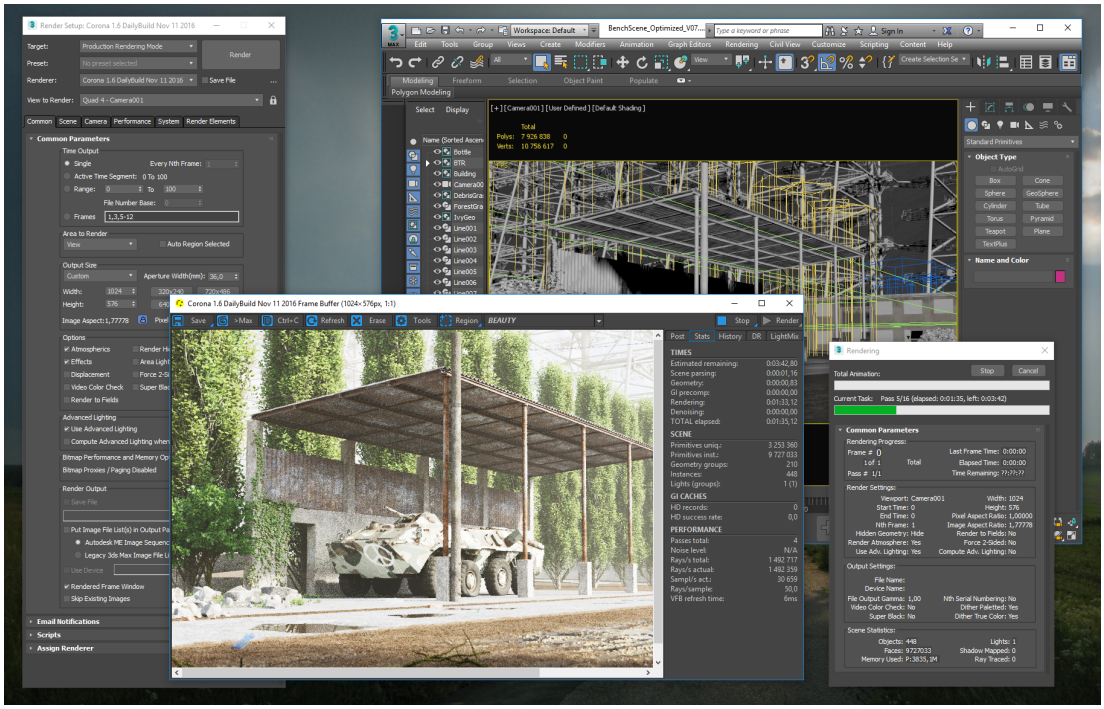


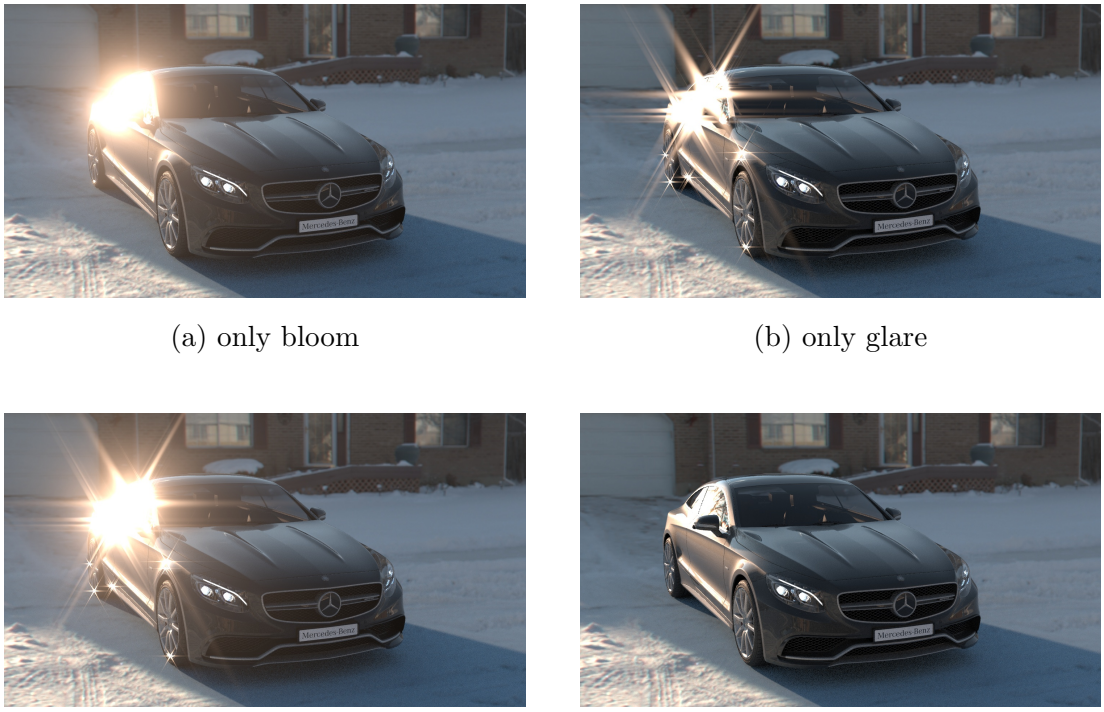
Figure 3.1: Autodesk 3ds Max with VFB of Corona Renderer

are mostly satisfied with it. Hence, it would be good to look for inspiration there. Furthermore, the Corona's Virtual Framebuffer (VFB), shown in Figure 3.1, actually provides similar features as those desired from the standalone viewer, so it would be more than appropriate. (The VFB is a renderer window showing the current state of the rendered image. It allows to perform certain post-processing operations while the image is still being rendered.)

First of all the viewer itself shall have a dark theme. This is kind of today's standard in majority of CG applications. It allows their users to concentrate more on the content than on the application itself. Furthermore, Koutný approves that most of the viewer's controls shall be placed on the right side of the application's main window, and shall be aggregated into collapsable sections. According to him it would be reasonable to organize the controls into a single-column layout, thus having only one adjustable control per row. It is because to Koutný this kind of design seems well-arranged, is easy to use and understand, and feels natural to users of the 3ds Max.

As for useful features, according to Karlík, users of physically-based renderers, such as Corona Renderer, mostly need to reduce noise in their renderings. And they may need to do it completely outside of the rendering process. This is because common users usually have only a limited amount of memory on their systems and with the renderer and especially with the host application (3ds Max, Cinema 4D, etc.) turned on, the memory may be already occupied by the parsed

scene and other data. Although such users can render the scene, they cannot denoise it using one of Corona's denoising algorithms because these also demand a serious amount of memory during their work. The new viewer is an excellent opportunity to solve this problem.



(a) only bloom (b) only glare

(c) both effects (d) original

Figure 3.2: Bloom and glare effects

Corona Renderer in its latest version 1.5 has come up with a new post-processing features like a possibility to add bloom and glare effects into renderings (see Figure 3.2), or mix light coming from several light sources placed within a scene. Especially the latter one has been very well received by a rendering community[2] and for this reason it is important to have them in the viewer too. Also the tone mapping possibilities are very nice there.

Among obvious viewer features shall be an ability to save all made changes into a file. Either again as an HDR image (offering at least the EXR format for that), or as a regular image (allowing to use common formats like PNG, JPEG, BMP). Without such possibility other features would be pointless.

[2]See the comments at `https://corona-renderer.com/forum/index.php/topic,13399.msg86485.html#msg86485`.

3.2.2 Power users

On the half way between common users and render farms are power users. These are people who have more advanced technical skills and who would like to use the viewer's capabilities, originally intended for common users, also through a CLI which is much more suitable for scripting.

What was not so clear from the discussion was a syntax of the intended CLI. The syntax is a part of user experience, and therefore, it needs to be carefully designed, similarly to the GUI. It is a question whether it is better to use longer and more descriptive names for parameters, or use rather shorter and easier to write (albeit possibly harder to remember) ones. Whether the syntax shall be adopted from some 3rd party well-known software like ffmpeg[3], ImageMagick's convert[4] etc., or rather design a new syntax that may better suit the viewer's needs.

In the end an agreement has been made and a syntax preferring long names of parameters compatible with names used within configuration files of Corona Renderer has been selected. Power users of Corona Renderer are accustomed to these names and it allows them to copy and paste appropriate parts of the configuration files directly into a command-line as parameters for the viewer. Furthermore, the names are rather more descriptive which is a good thing for writing scripts. It makes them more readable. On the other hand it is not expected that users of the viewer would use its CLI on a daily basis and hence more typing while writing commands for the viewer might not matter. Also a maximum length of a command for a command-line is not being seriously limited these days in any widely used operating system. So this shall not be a problem either.

There is a one particular feature that power users might need. They may want to compare two HDR images by calculating their difference. Common users probably do not have a need for such a feature but according to Křivánek, for researchers and developers of rendering algorithms it is an essential feature. Because of that, the viewer's CLI shall provide a possibility to calculate at least a positive/negative difference of two HDR images in a same way how it is being done in qt4image application presented within the overview. The other difference methods may come later.

[3] See `https://ffmpeg.org/ffmpeg.html`.

[4] See `https://www.imagemagick.org/script/command-line-processing.php`.

3.2.3 Render farms

According to Karlík, render farms have some special needs. They need to stitch partial results coming from rendering nodes into one single output HDR image. Furthermore, this has to be fully doable through a CLI because render farms may need to script such tasks.

The partial results may be basically of two forms: either they could be only image sub-regions of the larger output image (often narrow horizontal belts taking full width of the output), or they may be full-resolution images but rendered only with a fraction of target number of samples per pixel (number of passes).

One way or another, render farms need a CLI tool to stitch these partial images together. The stitching is basically just a calculation of weighted sum of the images separately for each pixel. Renderers usually store a special weight channel into the images for these purposes. For example in Corona Renderer it is always named *CORONA_FB_WEIGHTS*. The tool shall use it automatically.

They may also need to reduce noise in the stitched image. Denoising image sub-regions before stitching them together may not often produce a seamless results. This is because noise reduction algorithms usually operate with masks and they need to access pixels from whole neighborhood which is not possible when only the sub-regions are being denoised. Hence, the denoising has to be performed at last – after the stitching. And it would be beneficial for users of render farms if the farms would be able to easily do this.

4. Viewer implementation

The viewer itself is written in C++ which is a traditional compiled programming language suiting well to CG purposes. One of the adjectives that could characterize it could be "scattered". Definitely much more than for example Java. Of course there is the Standard Template Library (STL) but it is not almighty. A lot of functionality must be seeked elsewhere. Projects written in C++ may also suffer from a complicated deployment. These are some of the problems that this chapter discusses.

The chapter also describes choices that had to be made, for example considering the used GUI framework. It also contains a discussion over a suitable HDR image representation and its processing.

4.1 Custom functionality

As one can guess from the beginning of this chapter, it has been necessary to implement a lot of custom functionality to bring the viewer to life. The following list enumerates some of it:

- class representing a single HDR image that works as a sophisticated container capable of holding numerous kinds of data and metadata which the image may contain (see the section 4.4 for further details)
- general input/output (IO) backend exploiting the official library for the EXR format [7], and supporting most of the format's features, both for loading and saving of HDR images represented through the special class above
- HDR image processing "pipeline" (see the section 4.5 for further details)
- algorithm for computing a positive/negative difference that has been implemented in the exact same way as in the qt4image application described within the overview of HDR viewers
- algorithm for accumulating/stitching partial results produced within render farms' workflow (see the section 3.2.3 for further details) which handles weights per pixel or per image, and can operate with HDR images having miscellaneous virtual positions to each other

- algorithm using image-based arithmetics to mix light coming from several light sources placed within a scene that has been rendered to a single HDR image (see the section 4.5 for further details)
- a lot of various operating and integration code plugging external functionality into the viewer's context (again see for example the section 4.5 for further details)
- custom well-extensible framework for parsing, validating and processing command-line arguments and sub-arguments (both either as standalone flags, or value-based options)
- smart thread-safe container for storing all viewer's properties of various types that uses previously bound lambda functions to notify other parts of the viewer about any property changes
- basically the whole viewer's GUI – custom controls, control panels, draw panel, art providers, dialogs and frames, etc.
- utilities for iterations over containers, inter-thread procedure calls, layouting GUI elements, some type traits, mathematical operators, and other helpers

4.2 Corona Renderer means

Actually it has been planned that a future development and maintenance of the viewer will be overtaken by developers from Render Legion. Hence it is desirable to adapt the whole project to their coding style and to all means that they are accustomed to use.

Code base of Corona Renderer itself, and all its related tools, is well organized and structured. It basically parts into two main components – into a library and a core. Both are written in C++ and exploit all modern features of this language. For example, lambda functions [14] are being used extensively in a way that helps to separate functionality into more independent pieces and hence make them more reusable. Without doubts this is always a good thing. The code base is also aware of rvalue references [14] and move semantics [14] which may reduce amount of necessary memory operations. For better code readability it conveniently uses range-based loops [14] and type inference through the auto keyword. It massively uses both compile time and run time assertions that help

with debugging. Premake[1] is used as a build system, which is an easy-to-use scriptable system for generation of project files for Microsoft Visual Studio and others. The viewer is expected to comply to all these conventions (and more) to be coherent with Corona Renderer.

The library provides lots of general and useful features. Some of them are already offered by the STL but the developers from Render Legion have decided to implement their own implementations. It is mainly because of binary compatibility reasons. But there are also other reasons like a possibility of better optimizations for CG purposes, or having richer assertions, etc.

The problem with binary compatibility is that Render Legion provide only a public API for their library. Its implementation is not available to others. Any 3rd party developer who wants to use the library does not compile it on his or her own but must link it with a provided precompiled Dynamic-link Library (DLL). This is where the compatibility problems can occur. For example if the exported interface from the DLL would involve some means from the STL, it can happen that the STL version used by the user of the DLL would be different from the version that has been used to compile the DLL. This could lead into an undefined behaviour after linking. It is even worse because one cannot even hope for binary compatibility when using same compiler and STL versions. Just different settings of the same compiler in the same version can break things too[2]. Render Legion solves this situation nicely by exporting only own means from their DLLs and by wrapping all potentially dangerous implementation details using a Pointer to Implementation (Pimpl) technique [14]. Otherwise the task to adhere binary compatibility would be very uneasy or even impossible.

By the way, while working on the viewer it has turned out that not having access to the actual implementation of some library can be sometimes a problem because its documentation might not always be so detailed as one may need. But through a process of trial and error it can be possible to discover at least some missing details.

Anyway, to get the idea what the library actually provides here is a list of some of the offered means:

- custom array-like containers including useful utilities for their addressing and iterating over them
- own smart pointer implementations for dynamic memory management [14]

[1] See `https://premake.github.io/`.

[2] See the comment at `http://stackoverflow.com/a/5736352/857952` for that.

- universal class for string representation supporting various encodings
- better stream implementation
- support for parallel computations including means for securing thread-safety
- support for vectorization using SSE 4.1 instructions at several places (see the image representation section for further details)
- implementation of miscellaneous mathematical functions and operators
- custom simple representation of images and bitmaps
- means for working with colors and color spaces
- classes for representation of geometric primitives
- various useful type traits [14] and enumeration traits
- special handy classes for returning expected, optional, and variable results
- asserting macros
- some platform-specific helpers querying the operating system
- custom GUI elements and utilities (see the section 4.3 for further details)
- and much more

As for the core, it keeps all the rendering and post-processing related functionality. These are much more specific than the functionality provided by the library and everything there is significantly more interdependent. For this reason, developers from Render Legion had to make some code adjustments to make core functionality also suitable for the viewer.

Among other things, the core offers two noise reducing algorithms that the viewer shall expose to the users of Corona Renderer. The first one is an algorithm removing only "fireflies" within HDR images. It is fast but it can only remove outlier pixels. The second is a fully-fledged special denoising algorithm based on non-local means filtering. It works well but it unfortunately consumes a lot of time and system memory. Both algorithms require some auxiliary data to be contained within HDR images. This is not a problem for images rendered by Corona Renderer itself because these data can be automatically dumped using Corona's VFB by a single click. The first algorithm only needs an additional

render element (i.e. a pack of several channels, usually consisting of a red, green and blue channel) containing sum of squared light contribution values in each its pixel. The full algorithm needs it too but in addition it needs also three more special feature elements with their squared variants.

The core also offers functions for computing bloom and glare effects as described in the analysis section. Their computations for high-resolution images might be also time demanding. To overcome this the viewer has been implemented in a way that such long running operations are being launched asynchronously in separate threads (this actually also applies for the denoising). It allows the viewer to be as interactive for users as possible.

Another core functionality of which the viewer takes advantage is tone mapping. The core already implements the global variant of the photographic operator published by Reinhard et al. [10]. It also offers a special filmic operator giving a better control over highlights and shadows. Apart from these two operators the core also provides other algorithms for tweaking the tone mapping process. For example, it is possible to adjust an overall image exposure, saturation, contrast, or add a color tint.

The viewer exploits the above functionality in a way that it can open an HDR image, display it, process it using (not only) the described means, and save it back to some file. For more information see the image processing section where the whole "pipeline" process is described.

4.3 Graphical framework

The viewer is primarily a GUI application and hence it has been necessary to choose some suitable GUI framework for it. In the end wxWidgets library has been selected because of the cooperation with Render Legion. This is a framework they use for GUI of Corona Renderer, even though it is not ideal.

They need a reasonable mature multi-platform framework for C++ with a licensing that would suit them, and this is something what wxWidgets[3] fulfills. That is why they have chosen it. Unfortunately it has also some dark sides. Its development somewhat stagnates. Some of its parts, for example like wxAUI (an advanced user interface library), seems to be abandoned for a long time. It lacks modern eye-candy features like animations. It is designed to provide a native look and feel on any platform it supports. This may sound good but it has its own

[3]See `http://www.wxwidgets.org/`.

drawbacks, especially for CG purposes. Its theming capabilities are extremely limited and turning application's UI into dark colors is really painful. To sum it up, in practice using wxWidgets results often into "better do it yourself" solutions and one can always expect that something will not work as expected with this framework.

Nevertheless it is still probably the best solution for current Render Legion needs, at least according to Karlík, because there is no good enough and suitable alternative to wxWidgets. Of course there is Qt framework[4]. Unlike wxWidgets it is very modern and complex, it has active and rapid development, and it offers an excellent documentation. But its license for commercial purposes is rather expensive. It costs about 300 USD per month per developer (depending on term length). Developers from Render Legion would probably prefer Qt over wxWidgets but it is more business decision than technical.

Other alternatives that fulfill most of the conditions mentioned above and that can be considered are often specifically oriented. For example JUCE[5] is much cheaper than Qt. It costs only about 50 USD per month per developer, but it is not as complete as Qt and it is oriented more on audio processing than on CG. Another possible framework is Crazy Eddie's GUI[6]. It is free and highly customizable. But it focuses on the development of UI for computer games. An interesting project is Chromium Embedded Framework[7] which allows to create really rich application UI very easily and quickly. This is because it seamlessly incorporates a web browser into the application and its UI is expected to be written in HyperText Markup Language (HTML) and JavaScript. These are both fast to write languages. But using it for CG purposes is questionable because it may introduce an unwanted overhead. Anyway, currently Render Legion tend to wxWidgets and that is why the viewer makes use of the same GUI framework.

4.4 Image representation

A way of storing HDR images in memory can be considered as a "cornerstone" of the viewer. For a single image (ignoring its alpha channel for now) it could be something as simple as a continuous buffer keeping interleaved red, green, and blue pixel values in order from the image's top-left corner to its bottom-right. This

[4]See `https://www.qt.io/`.
[5]See `https://www.juce.com/`.
[6]See `http://cegui.org.uk/`.
[7]See `https://bitbucket.org/chromiumembedded/cef`.

approach to data arrangement is easy-to-use. Hence it is very common and it is widely adopted in software development. As it was denoted in the introduction section, this kind of arrangement is known as Array-of-Structures (AoS). Pixels are in the role of "structures" in here.

The opposite arrangement is the Structure-of-Arrays (SoA). This is when all the red pixel values are separated into their own buffer and the same goes for the green and the blue values. Then the "structure" here would be a triplet of buffers. Generally this kind of arrangement may not be so comfortable for software developers because they usually want/need to work with all values related to a single pixel at once, but it might bring some significant speed-up. This is because it gives a better ground for "vectorization" of a code, i.e. for using Single-instruction/Multiple-data (SIMD) operations. These are special processor instructions capable of manipulating several values of same type, either integers or floating-point numbers, in a parallel fashion. There is a whole bunch of such instructions and usually not all of them are supported by every processor. Agner Fog provides very nice instruction tables for that purpose [15]. In this thesis we would consider only two the most important instruction families of today: Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX). The former ones are 128 bits wide and hence can process for example four 32-bit floating-point numbers at once. To be precise we would use SSE 4.1 variant. The latter ones are wider. They process 256 bits, i.e. twice as much as SSE.

In principle, the advantages of using SoA over AoS are obvious. With the SoA arrangement, we can basically use as wide instructions as our processor supports. It is because it can be expected that all algorithms above SoA data are designed with this in mind. It is why we use the SoA arrangement in the first place. Hence when a new wider instruction family becomes available, porting a code manipulating SoA data would not be probably a big problem. We would just use the new instructions instead of old ones and take care of proper memory alignment [16], if that would be necessary. For example, it seems that a new instruction set denoted as AVX-512 will probably come one day. At least Intel has announced it for high-end processors in summer 2013 [8]. This will allow to process sixteen 32-bit floating-point numbers at once with the SoA. That is remarkable.

On the contrary with the AoS we often cannot use wider instruction sets so easily. Usually algorithms used with this arrangement are either not aware of vec-

[8] See `https://software.intel.com/en-us/blogs/2013/avx-512-instructions` for that.

torization at all, or they are vectorized only on a level of single structures (pixels in our case). Width of usable instructions is limited in such cases. A reasonable pixel representation of an HDR image takes 96 bits (three 32-bit floating-point numbers for red, green, and blue pixel components). At best, manipulation with single pixels could be accelerated through 128 bits wide SSE. And even then we would waste a serious amount of memory because the 4th component of every such pixel would be simply extra. It could be used for an eventual alpha value but that is not always present. A fact is that in practice the 4th component is usually unused.

On the other hand, the SoA may not provide such speed gain over the AoS as one could expect. It highly depends on the algorithm that manipulates the data. SIMD operations are strong in nontrivial mathematical computations. For example, it may surely pay off to compute a logarithm of four numbers at once. But with simple arithmetics there could be not so much acceleration. It is because the processor probably will not be the bottleneck there. It would be the memory subsystem. If we are reaching its speed limits, it will not matter how wide instructions we would use for that. It could not be done faster. So in such cases bothering with SoA may not provide significantly better results than the AoS. On the other hand, it will not be slower either.

There is an interesting case study made by Intel. It compares the AoS and SoA data layouts for a compute-intensive loop run using various instruction sets on high-end Intel Xeon family processors [17]. The author of the study clearly says in the conclusion that because of software complexity and diversity for any findings there will probably exist counter-examples. Because of this he makes rather suggestions. He suggests that "it is almost always better to vectorize than not to vectorize on Intel SIMD capable hardware", and that if we are vectorizing then "it is almost always better (faster) to vectorize with the SoA data layout than with the AoS data layout, simply because chances of the compiler doing something wonderful are much better". He suggests to prefer SoA over AoS when we are in doubt because "there are probably not so many examples where a code using AoS would run significantly faster than its SoA variant" [17].

Because all of the above reasons it has been decided that it would be probably better to arrange HDR image data using the SoA approach. Even so if this decision would make the implementation of the viewer slightly more difficult, or conversions between the AoS and SoA arrangements would be required. These conversions are expected to be fast enough because it is nothing more than an in-memory copying. Furthermore, the SoA arrangement is closer to the concept

of the EXR format that understands channels as independent of each other. See the section with preliminaries for that. This is another reason why choosing the SoA is probably a right way to go.

Actually, the EXR format can be considered as the most advanced format among those that are commonly used, at least when it comes to extensibility capabilities. Hence it is good to look for inspiration there for a design of a representation of a single HDR image. That way we shall be able to cover most of the EXR capabilities. It is expected that other common formats can be easily "mapped" to this representation. In addition, such uniform environment should simplify the viewer's implementation.

The viewer implements the representation sketched above within a class named ChannelImage. Like the EXR format it offers the capability of storing channels and attributes. It offers several ways how they both can be accessed, searched, and iterated. By default instances of the class contain no channels or attributes. They have to be inserted first. It is also possible to remove them any time later.

Every channel keeps values of some selected type. Usually the values are 32-bit floating-point numbers, but basically the type could be anything. Channels are always properly aligned in memory with regard to the SSE. Such alignment may help to keep vectorized code effective on some hardware [16]. All channels within a single ChannelImage instance shall always share same dimensions, otherwise the instance will not be valid. The dimensions are defined by the "dataWindow" attribute. See the OpenEXR section for that. Any attribute in ChannelImage holds a single value of some type. Again, it could be just about anything. Technically all of the above uses C++ templates and dynamic type conversions [14].

The class offers two more important features. The first of them is a functionality for retrieving and adjusting whole render elements. These are provided as bitmaps consisting of a red, green, blue, and possibly alpha channel. The channels are arranged in the AoS manner there. The render element bitmaps are being constructed from appropriate channels on demand. This is expected to be fast enough as stated above.

The second feature is that the class defines interfaces for loading and saving its instances. For now, these two interfaces are implemented for the EXR format. Several common types of EXR attributes are supported. EXR channels containing 16-bit floating-point values are always transparently converted to 32 bits. This is because there are no SIMD operations available for them.

See the generated Doxygen documentation attached to viewer sources for further details about the ChannelImage class and its API.

4.5 Image processing

Based on the requirements for the viewer presented in the analysis section, this section describes a process that has, from a technical point of view, led to their fulfilling. The key term here is a "pipeline". This is something that is responsible for all necessary processing of a viewed image. It includes steps needed for loading the image from a file (extraction from the EXR format), making all image transformations according to user requests which has been previously translated into the pipeline settings (for example computing bloom and glare effects, performing denoising, tone mapping, etc.), making all inevitable operations needed for actual displaying of the image (clamping pixel values to low-dynamic range, or applying a gamma correction), or alternatively saving the image back to some file (packing to some selected output file format).

Software development is a process of continuous designing, code rewriting, and functionality broadening. Probably every well managed project starts with just a prototype of final functionality. As the development time progresses, new features are being added in a "snowball effect". Time to time we hit a wall with it. It could be either because of some new requirements that have not been known yet at the time of original design, or it could be just because of something has been unconsidered. In such cases it could be necessary to redesign and rewrite some code, sometimes from scratch. Despite of how hopeless it may look it is a process that converges toward the project goals. With little exaggeration, it can be said that it is just a matter of time. It depends on skills and experience of a developer how well he or she can anticipate eventual problems and design suitable code.

Development of the viewer was not different in this. Several versions of the pipeline had been implemented until all things settled down. Its first version was just a linear sequence of routines with a simple interface which were performed one by one. Each of them was taking care of one particular operation of the pipeline process described at the beginning of this section. It suited well the command-line version of the viewer, but it completely lacked interactiveness which is something we want from the GUI version. Such design was too simple for that. To solve this it has been necessary to rethink the whole design and wrap all the pipeline operations into separate objects. That way they can be easily identified and some other functionality can be bound to them. This is something that the viewer's UI can exploit. For example, when the user clicks an appropriate button, an object implementing the denoising feature may recompute its cache and hence in the end update the noise reduced variant of the viewed image.

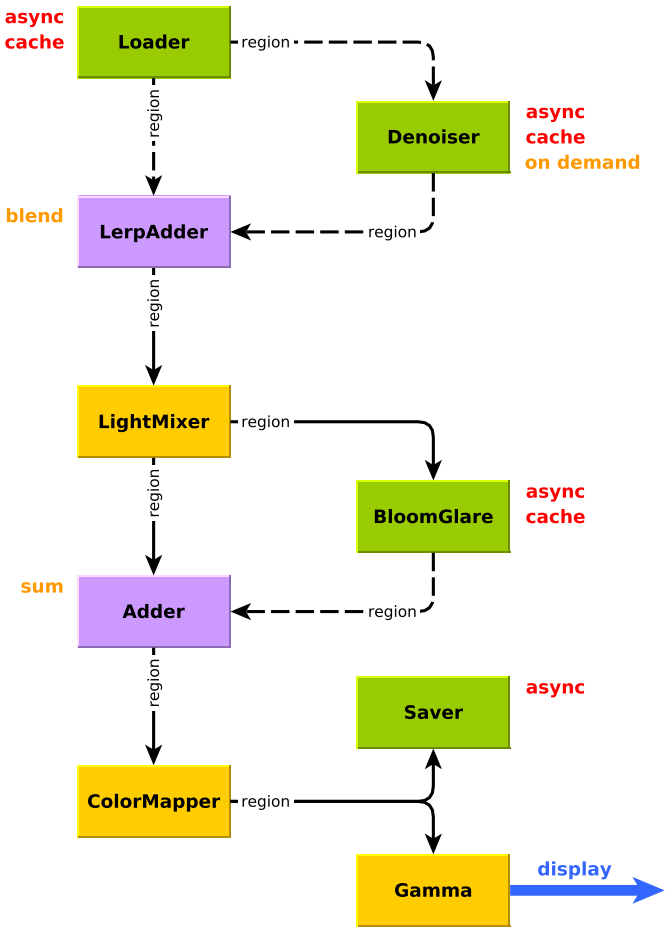


Figure 4.1: Pipeline workflow

The diagram 4.1 gives an overview of the pipeline as it has been implemented. It currently consists of 9 operator objects having clear interface to communicate with each other. Each of them can be understood as an on-demand black box with a strictly defined arity. Anytime it is requested for a "region", it recursively asks its operands and does its job based on the results retrieved from them. The region here is a special optional bitmap keeping pixel values from a portion of some selected render element. Hence such bitmaps are being used by the operators to transfer image data. The key feature of regions is that they can be asked for their availability. This is a way how an operator can let any dependent operator know that the image data it requests are unavailable. This may be because of either a temporary, or permanent reason. For example, an operator that internally operates with some sort of a cache may signalize that any requested region is temporarily unavailable until the operator's cache becomes ready (which may take some time because of the nature of the process behind). Such cases are indicated by a dashed connection line within the diagram.

In the code the pipeline is implemented by the `Grid` class. As can be seen in the diagram, it namely composes the following operators:

- `Loader`

  This is an operator responsible for asynchronous loading and accumulation of HDR images from requested EXR input files. It serves as a data feeder for the whole pipeline. It informs about its state and progress through a set of callbacks.

- `Denoiser`

  On demand, it can run any of the special noise reduction algorithms provided by Render Legion as a part of their core code base. See the analysis section for that. The operator performs its task asynchrounosly and caches its result. It is because the denoising is a very time consuming process. As noted before, it also usually requires a serious amount of memory. Again, it uses callbacks to inform about its progress.

- `Adder`

  It just composes all requested regions from its two other operands. This is being done using only a simple addition.

- `LerpAdder`

  Same as the `Adder` but for the addition it uses a linear interpolation. In the end it serves as a blender of an original image with its noise-reduced variant produced by the `Denoiser` operator.

- `LightMixer`

  Implements a light mixing algorithm which mixes light coming from several different light sources placed within a scene, and which is rendered into separate render elements. The final mixture is controlled by a set of intensity and color multipliers. The operator provides a special virtual render element for presenting the mixture.

- `BloomGlare`

  This is an operator that is used to compute both the bloom and glare effect provided by Render Legion within their core. It could take a non-negligible amount of time for high-resolution HDR images and hence the computation is again asynchronous, cached, and can be monitored by a set of callbacks.

- `ColorMapper`

  An operator which is responsible for all the tone mapping features that

Render Legion offers in their core. See the analysis section for them. It never applies gamma correction as described within the preliminaries section. Such thing is extracted into the separate `Gamma` operator.

- `Gamma`
  Performs the gamma correction. This is being done just before displaying any region to the user. It is separated from the ColorMaper because it is purely a matter of displaying image data on a screen. The gamma correction while saving an image has to be handled separately because it is format-dependent. See the `Saver` operator for that.

- `Saver`
  This is an operator responsible for asynchronous saving a pipeline processed HDR image to some specified output file. It supports either the EXR format, or various regular formats like BMP, PNG, JPEG, etc. for which it automatically performs the gamma correction. It can be used to save only a single render element selected within the viewer, or to save all render elements at once. If the target file format does not support multiple elements within a single file, more files will be created (one for each render element). It also informs about its state and progress through a set of callbacks.

As can be seen from the above overview, technical means required by the individual operators differ. Some need to perform asynchronous tasks, others do not. Some need to access the pipeline settings. Some would like to inform about their current status. Basically operators may need various things that can turn to be so called "orthogonal", i.e. they can be mixed in ways which prevent designing any kind of a reasonable "is-a" inheritance hierarchy [18]. In C++ these situations can be nicely solved through a mixin-based design [19]. This is what the viewer uses in its implementation.

See the diagram 4.2 that illustrates the whole operator inheritance model. Firstly, there is the `IOperator` class which is a lowest base class of all the above mentioned operators. It defines an interface which must be implemented by any pipeline operator, and which implements some basic functionality for them. Then there are the individual mixins, i.e. classes where each of them introduces some unique common functionality. These are C++ templates that can be mixed together just through their template parameters. Every such mixture, with the `IOperator` class at the lowest level, can be used as a base class of some operator and hence enriching it of the mixed functionality. For example, the `LightMixer` operator is derived from a `DependentOperator<1,`

`SettingsOperator<IOperator>` mixture, as can be deduced from the diagram (order of individual mixins within a mixture does not matter).

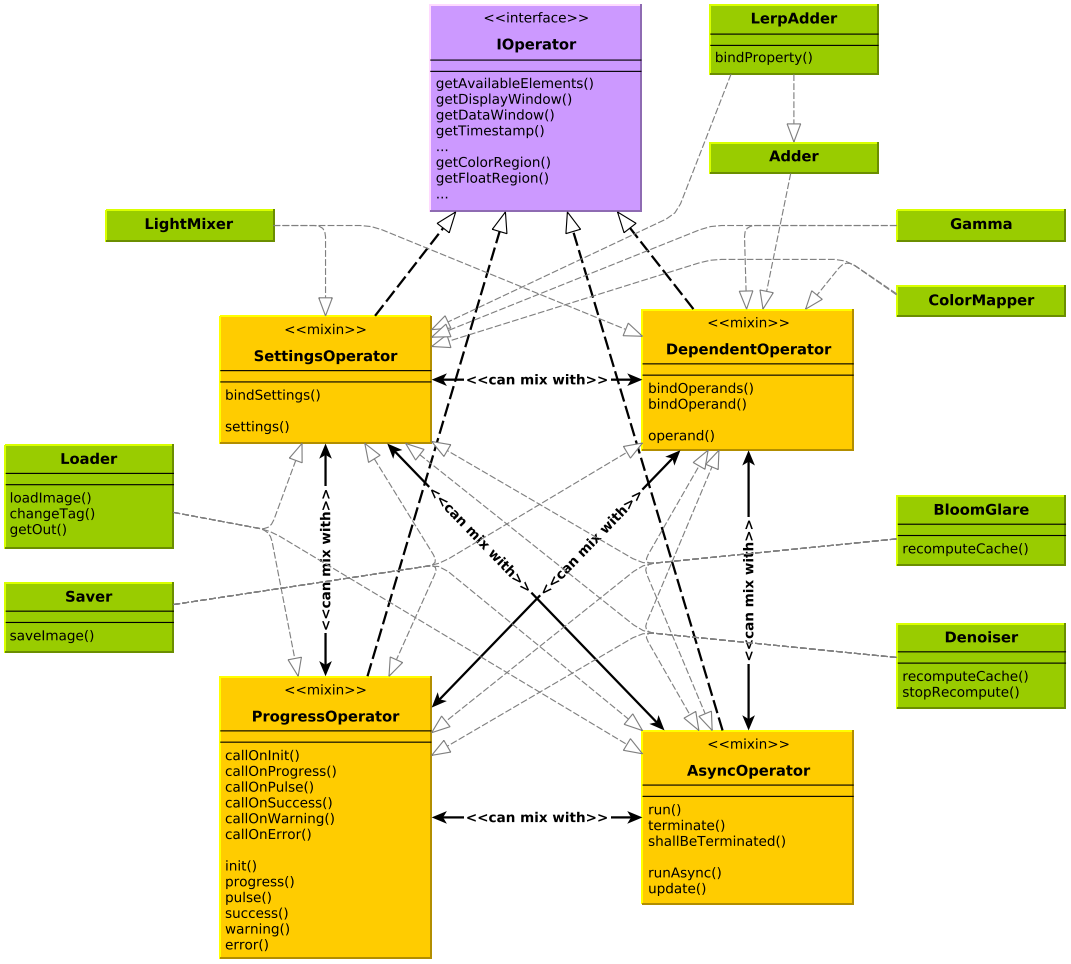


Figure 4.2: Operator inheritance model

The diagram also shows an excerpt from mixins' and operators' API. Their full versions can be found within the generated Doxygen documentation attached to viewer sources.

The above presented design of the pipeline implementation has turned out to be very flexible and dynamic. Because of the clear operator interface and usage of the mixin concept it is very easy to alter the pipeline workflow – either introduce a new operator there, or just reconnect it in a different way. The mixin technique suits well to the presented situation. It avoids code duplication and makes the code implementation effective. All in all, it seems that the presented code design is probably well maintainable to the future.

5. Evaluation

One of the goals of this thesis is to assess whether the concerns about possible ways of image data arrangements, introduced within the Section 4.4 describing image representation, are well-founded and are not pointless. Whether it may pay-off from a memory usage and an overall performance point of view to bother with a massive usage of SIMD operations and prefer the Structure-of-Arrays (SoA) data layout for that over the Array-of-Structures (AoS) layout which is usually more comfortable for a developer.

We have already inspected the memory question in Section 4.4. We would waste memory in cases when a vectorized algorithm is operating on structures of values whose overall size is not a multiple of 128 or 256 bits. And that is also in our case where we like to represent pixels by three 32-bit floating-point numbers. Hence using the SoA approach is more suitable from this perspective.

To answer the performance question we have to measure. We already have a suspicion that the SoA layout may be faster but such a thing has to be always proven for a concrete algorithm, as noted by the author of the case study [17]. Unfortunately this is often hard to do because every such algorithm has to be actually implemented twice for that and in practice applications often contain many such algorithms. The viewer is not an exception. To overcome this only a reasonable and representative (if possible) subset of the algorithms may be tested. This could give a hint how to behave when designing the others.

For purpose of this thesis, an algorithm for changing overall saturation of an HDR image has been selected for testing. This algorithm is part of the core of Corona Renderer and is accelerated by SSE instructions there. It uses the AoS principle. This choice has been made based on a consultation with Michal Prokš who is one of main software developers at Render Legion. The saturation algorithm can probably represent all algorithms hidden behind the viewer's tone mapping process. See the section with Corona Renderer means for that. If the saturation algorithm could gain some speed-up, it may be probably assumed that it would similarly also work for the others.

For the testing purposes, a version of the algorithm utilizing the SoA principle has been implemented. An excellent C++ library[1] made by Agner Fog has been used for that. It is a collection of classes wrapping SIMD instructions at various widths. Hence it is possible to write a code designed for example for AVX

[1] See `http://www.agner.org/optimize/#vectorclass`.

instructions with it. Such thing is not currently offered by the Corona Renderer means.

Both versions of the saturation algorithm were tested using the same methodology. Tests were based on measuring an average wall-clock time and processor time (that is a time spent by performing processor tasks) needed to change overall saturation of a pregenerated random HDR image with a resolution of 4K (4096×2160 pixels). The averages were calculated from 2500 samples made in a sequence. The sequence was divided into 50 separated chunks. Both the code used to measure the averages and the measured averages can be found in the attachments. This text presents only overall results of the tests. The measuring code had been compiled in Microsoft Visual Studio 2015 with all optimizations turned on, and was ran in the following two testing environments:

1. **Laptop Dell Vostro 3560**
   CPU Intel Core i7-3612QM (2.10 GHz),
   8 GB RAM (2× 4GB DDR3 1600 MHz),
   Microsoft Windows 10 Pro 64-bit

2. **Desktop computer**
   CPU Intel Core i7-6700 (3.40 GHz),
   32 GB RAM (4× 8GB DDR4 2667 MHz)
   Microsoft Windows 10 Pro 64-bit

5.1 Results

The table 5.1 shows the measured average times in milliseconds for both environments. There are also calculated speed-ups related to the AoS variant using SSE instructions, i.e. to the variant that is currently used by Corona Renderer. The speed-ups are based wall-clock times because this is the time which users usually care about.

	Laptop Dell Vostro 3560			**Desktop computer**		
	Wall-clock time	Processor time	Rough speed-up	Wall-clock time	Processor time	Rough speed-up
AoS with SSE	88.192 ms	88.162 ms	—	58.689 ms	58.656 ms	—
SoA with SSE	28.579 ms	28.556 ms	**3.09**×	18.219 ms	18.194 ms	**3.22**×
SoA with AVX	18.859 ms	18.856 ms	**4.68**×	11.219 ms	11.188 ms	**5.23**×

Table 5.1: Measured average times

To summarize the table 5.1, one can say that using the SoA approach also pays off from the performance point of view. Three times speed-up over the AoS variant using the same instruction set is a nice positive result. Although the AVX instructions are twice as wide the speed-up is not doubled in this case. One of the reasons may be that we are probably hitting the maximum throughput of the memory subsystem in the tested environments. But it could be caused also by other reasons. A more detailed analysis would be necessary to decide what is the main cause of this. Fog provides a nice overview of some potential bottlenecks at page 100 of his manual [20]. This could be a starting point for this.

Conclusion

This chapter offers an overview of the goals that have been achieved in this thesis and some directions of the viewer's future development.

Achieved goals

The main goal of this thesis was to design and implement a viewer of HDR images that would offer some advanced ways of manipulating this kind of images. Because of this a great emphasis has been given to discovering true needs of professionals who use HDR images on a daily basis. A part of this was also a reconnaissance of already existing solutions. The viewer has been carefully designed based on the discovered findings. This has been an important step because it made the main goal well-defined and its purpose was clear.

Using a standard procedure, the implemented viewer can be associated on Windows to the EXR format and therefore serve there as an easy one-click solution for viewing these images. Professional CG artists will surely appreciate its features, particularly the ability to reduce noise that is present in all HDR images produced by physically-based renderers, and the use the arithmetics-based tool for mixing lights coming from different light sources in the scene. On the other hand, more technically skilled users may take advantage mainly of its CLI. It could be used either by render farms to accumulate/stitch rendered images directly into a final product, or for some batch processing purposes. A great importance has been given to a compatibility with Corona Renderer, hence especially users of this renderer will find the viewer's features useful. But it is well usable for others too.

The final goal was to make the viewer memory and performance efficient. That was a motivation behind the effort to make the viewer's design friendly to the usage of modern SIMD operations. In certain conditions they can make algorithms perform significantly faster. The thesis has discussed, implemented, and then also evaluated that one of the conditions, but not the only one, is a suitable data arrangement. It has confirmed that there is a big potential to run at least the tone mapping process faster.

Future work

As it was stated above, developing a software is a never ending story. There are still things that may be improved. Fortunately the work made as a part of this thesis does not end here. It is planned that Render Legion company will continue with it.

The implemented viewer could be extended in several ways. For example, it could support some other formats of HDR images – the possibility to view images in the RGBE format would be nice. It could be made multi-platform. Basically there are no big obstacles that would prevent such thing. The viewer could be also enhanced with further post-processing capabilities like Look-up tables (LUT) or curves (both in their own way define a color mapping function). Another nice-to-have feature could be a better integration into Windows Explorer for which it could provide thumbnails of files containing HDR images. The viewer would also deserve implementation of some UI "gadgets" like a drag and drop of files into the viewer's window, or copying the currently viewed image into a clipboard. Less common methods of computing a difference of two HDR images could be implemented.

As for the viewer's CLI, its first version has been already sent to some selected render farms for testing purposes. At the moment communication is underway to obtain their feedback. It is possible that some modifications and feature requests would arise from it.

Bibliography

[1] Roger N. Clark. Notes on the Resolution and Other Details of the Human Eye. `http://clarkvision.com/imagedetail/eye-resolution.html`, 2016. [Online; accessed January 4, 2017].

[2] Christian Bloch. *The HDRI Handbook 2.0: High Dynamic Range Imaging for Photographers and CG Artists*. Rocky Nook, 1st edition, 2012.

[3] Erik Reinhard, Greg Ward, Paul Debevec, Sumanta Pattanaik, Wolfgang Heidrich, and Karol Myszkowski. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann/Elsevier, 2nd edition, 2010.

[4] Charles A. Poynton. Rehabilitation of gamma. In Bernice E. Rogowitz and Thrasyvoulos N. Pappas, editors, *Human Vision and Electronic Imaging III*. SPIE-Intl Soc Optical Eng, jul 1998.

[5] *IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[6] Florian Kainz, Rod Bogart, Piotr Stanczyk, and Peter Hillman. Technical Introduction to OpenEXR. `http://www.openexr.com/TechnicalIntroduction.pdf`, 2013. [Online; accessed January 4, 2017].

[7] Florian Kainz. Reading and Writing OpenEXR Image Files with the IlmImf Library. `http://www.openexr.com/ReadingAndWritingImageFiles.pdf`, 2013. [Online; accessed January 4, 2017].

[8] Greg Ward. High Dynamic Range Image Encodings. `http://www.anyhere.com/gward/hdrenc/Encodings.pdf`. [Online; accessed January 4, 2017].

[9] Greg Ward. CIE Luv Color. `http://www.anyhere.com/gward/pixformat/cieluvf1.html`. [Online; accessed January 4, 2017].

[10] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21(3), jul 2002.

[11] Erik Reinhard and Kate Devlin. Dynamic Range Reduction Inspired by Photoreceptor Physiology. *IEEE Transactions on Visualization and Computer Graphics*, 11(01):13–24, jan 2005.

[12] F. Drago, K. Myszkowski, T. Annen, and N. Chiba. Adaptive Logarithmic Mapping For Displaying High Contrast Scenes. *Computer Graphics Forum*, 22(3):419–426, sep 2003.

[13] Greg Ward. Graphics Gems IV. chapter A Contrast-based Scalefactor for Luminance Display, pages 415–421. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[14] C++ reference. `http://en.cppreference.com/`. [Online; accessed January 4, 2017].

[15] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf`, 2016. [Online; accessed January 4, 2017].

[16] Agner Fog. VCL: C++ vector class library. `http://www.agner.org/optimize/vectorclass.pdf`, 2016. [Online; accessed January 4, 2017].

[17] Paul Besl. A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel Xeon processors and Intel Xeon Phi product family coprocessors. `https://software.intel.com/sites/default/files/article/392271/aos-to-soa-optimizations-using-iterative-closest-point-mini-app.pdf`, 2013. [Online; accessed January 4, 2017].

[18] When to use inheritance. `https://msdn.microsoft.com/en-us/library/27db6csx%28v=vs.90%29.aspx`. [Online; accessed January 4, 2017].

[19] Yannis Smaragdakis and Don S. Batory. Mixin-Based Programming in C++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, GCSE '00, pages 163–177, London, UK, UK, 2001. Springer-Verlag.

[20] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `http://www.agner.org/optimize/microarchitecture.pdf`, 2016. [Online; accessed January 4, 2017].

List of Abbreviations

AoS	Array-of-Structures
API	Application Programming Interface
BMP	Windows Bitmap
CG	Computer Graphics
CIE	International Commission on Illumination
CLI	Command-line Interface
DLL	Dynamic-link Library
DR	Distributed Rendering
EV	Exposure Value
EXR	OpenEXR
GIF	Graphics Interchange Format
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDR	High-dynamic Range
HDRI	High-dynamic Range Imaging
HTML	HyperText Markup Language
ILM	Industrial Light & Magic
INRIA	French Institute for Research in Computer Science and Automation
IO	Input/Output
JPEG	Joint Photographic Experts Group
LDR	Low-dynamic Range
LUT	Look-up Table
PFM	Portable Floatmap
PGM	Portable Graymap
PNG	Portable Network Graphics
PPM	Portable Pixmap
RGBE	Radiance
RLE	Run-length Encoding
SIMD	Single-instruction/Multiple-data
SoA	Structure-of-Arrays
STL	Standard Template Library
TIFF	Tag Image File Format
UI	User Interface
VFB	Virtual Framebuffer