

WYSIWYG COMPUTATIONAL PHOTOGRAPHY
VIA VIEWFINDER EDITING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jongmin Baek
December 2013

© 2013 by Jongmin Baek. All Rights Reserved.

Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.

<http://creativecommons.org/licenses/by-nc/3.0/us/>

This dissertation is online at: <http://purl.stanford.edu/ry541tg7661>

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Marc Levoy, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Kari Pulli

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The past decade witnessed a rise in the ubiquity and capability of digital photography, paced by the advances in embedded devices, image processing and social media. Along with it, the popularity of computational photography also grew. Many computational photography techniques work by first capturing a coded representation of the scene—a stack of photographs with different settings, an image obtained via a modified optical path, et cetera—and then computationally decoding it later as a post-process according to the user’s specification.

However, the coded representation, available to the user at the time of capture, is often not sufficiently indicative of the decoded output that will be produced later. Depending on the type of the computational photography technique involved, the coded representation may appear to be a distorted image, or may not even be an image at all. Consequently, these techniques discard one of the most significant attractions of digital photography: the what-you-see-is-what-you-get (WYSIWYG) experience.

In response, this dissertation explores a new kind of interface for manipulating images in computational photography applications, called viewfinder editing. With viewfinder editing, the viewfinder more accurately reflects the final image the user intends to create, by allowing the user to alter the local or global appearance of the photograph via stroke-based input on a touch-enabled digital viewfinder, and propagating the edits spatiotemporally. Furthermore, the user specifies via the interface how the coded representation should be decoded in computational photography applications, guiding the acquisition and composition of photographs and giving immediate visual feedback to the user. Thus, the WYSIWYG aspect is reclaimed, enriching the

user’s photographing experience and helping him make artistic decisions before or during capture, instead of after capture.

This dissertation realizes and presents a real-time implementation of viewfinder editing on a mobile platform, constituting the first of its kind. This implementation is enabled by a new spatiotemporal edit propagation method that meaningfully combines and improves existing algorithms, achieving an order-of-magnitude speed-up over existing methods. The new method trades away spatial locality for efficiency and robustness against camera or scene motion.

Finally, several applications of the framework are demonstrated, such as high-dynamic-range (HDR) multi-exposure photography, focal stack composition, selective colorization, and general tonal editing. In particular, new camera control algorithms for stack metering and focusing are presented, which takes advantage of the knowledge of the user’s intent indicated via the viewfinder editing interface and optimizes the camera parameters accordingly.

Acknowledgments

Meaningful achievements are seldom realized in their entirety by an individual, and this dissertation is no exception. There are many people who directly or indirectly shaped not only the direction and the content of this dissertation, but also the experience of creating it. I am blessed to have known great colleagues and friends for the past five years at Stanford.

I should credit my advisor, Professor Marc Levoy, for always being cognizant of the big picture and rescuing me from being bogged down in details. His style very much complemented my strengths, and in the end, his boundless knowledge in computer graphics, great acumen and patience were instrumental in reforming the myopic mathematician in me to be a computational photography researcher. Kari Pulli and Professor Mark Horowitz were also great mentors and sources of advice.

Thanks to Andrew Adams, Jen Dolson and Abe Davis, all colleagues at Stanford, for their collaboration on some of the basic building blocks of this project, and for their role in making my office life brighter with their camaraderie. Zhengyun Zhang, Eddy Talvala, Sung Hee Park, Noy Cohen, Michael Broxton, Sid Chaudhuri also were great colleagues, friends or officemates, full of ideas and (sometimes bad) jokes.

Thanks to my external collaborators, namely Dawid Pająk and Kihwan Kim at NVIDIA Research, for commiserating with me through the numerous sleepless nights of coding and experimentation. Without their dedication, the implementation of this dissertation surely would not have achieved its current performance. Kari Pulli and other NVIDIA Research staff were also helpful in vetting the progress of the dissertation project and provided valuable feedback.

Thanks to my friends who kept my endorphin level up for the past five years,

specifically from the Stanford Korean Football Club and Stanford '85 Korean group—there are too many to enumerate. Also thanks to friends from my undergraduate days who helped me through both good times and bad times, especially Clare Dean, Rui Niu, and Heymian Wong.

Very special thanks to David Jacobs, my long-time officemate and partner-in-crime, who is everything one could ask for in a colleague, confidant and close friend. I can only hope to have been as much of a positive in his life and graduate experience as he has been to mine. Thanks for being available for everything, whether it be a chat about the next project idea or a late-night gaming session.

I should sneak in a mention of my M. Eng. advisor at MIT, Professor Frédo Durand, for exposing me to computational photography. Without the wonderful new undergraduate course he introduced at MIT, which immediately hooked me into the field, I doubt I would have any business with computational photography, let alone computer graphics.

Lastly and most importantly, I would like to thank my mother Hye Kim, for supporting me throughout the years and prodding me to keep learning. If my colleagues taught me how to become a good researcher in my field, then she taught me how to become a whole person. Of course, it was she who got me motivated to go down the road of higher learning in the first place, assembling science kits for me when I was but a little kid. For that, I am grateful. Mom, this one is for you.

Contents

Abstract	iv
Acknowledgments	vi
1 Introduction	1
1.1 The WYSIWYG Paradigm	2
1.2 The Viewfinder as a Canvas	4
1.3 Contributions	5
2 Prior Work	8
2.1 Accelerating Gaussian Filtering	9
2.1.1 Signal-Processing Approach	11
2.2 Camera Control for Computational Photography	14
2.2.1 High Dynamic Range Imaging	15
2.2.2 Focal Stack Photography	17
2.2.3 Composition	18
2.3 Edit Propagation	19
2.3.1 Edge-Aware Smoothing	20
2.3.2 Appearance-based Smoothing	21
2.3.3 Note on User Interface	21
2.3.4 Other Alternatives based on Tracking	22
3 The Permutohedral Lattice	23
3.1 Definition	24

3.2	Prior Work on Gaussian Filtering	27
3.2.1	Splatting	27
3.2.2	Blurring	28
3.2.3	Slicing	30
3.2.4	Summary	30
3.3	Fast Quantization in the Permutohedral Lattice	30
3.3.1	Further Improvement	36
3.4	Adaptations for Streaming Texture Retrieval	39
3.4.1	Importance Measure	40
3.4.2	Deleting Vertices	40
3.5	Summary	41
4	Viewfinder Editing	42
4.1	Affinity-Based Edit Propagation	43
4.2	Representing and Specifying Edits	44
4.3	Descriptor Design	47
4.3.1	Evaluation	51
4.3.2	Discussion	52
4.4	Computing Edits Quickly and Robustly	54
4.4.1	Subsampling Edit Masks	54
4.4.2	Multi-scale Lookup	56
4.4.3	Spatiotemporal Smoothing	56
4.4.4	Offline Processing	57
4.4.5	Discussion	58
4.5	Summary	59
5	Appearance-Based Camera Control	60
5.1	Appearance-Based HDR Metering	61
5.1.1	Image Appearance Model	61
5.1.2	Per-Pixel Objective Function	64
5.1.3	Aggregating Per-Pixel Objectives	66
5.2	Appearance-Based Focal Stack Focusing	66

5.2.1	Image Appearance Model	67
5.2.2	Per-Pixel Objective Function	69
5.3	Evaluation	70
5.3.1	Methodology	70
5.3.2	Noise Characteristics for Static Scenes	72
5.3.3	Metering for Moving Scenes	72
5.3.4	Empirical Results	77
5.3.5	Effect of Local Edits	77
5.4	Summary and Discussion	78
6	Implementation	82
6.1	Hardware	82
6.2	Camera Pipeline	84
6.2.1	Image Acquisition for HDR Application	84
6.2.2	Image Acquisition for Focal Stack Application	85
6.2.3	Registration and Alignment	85
6.2.4	Image Blending for HDR Application	86
6.2.5	Image Blending for Focal Stack Application	87
6.3	User Interface	88
6.3.1	Edit Modalities	89
6.3.2	Selection Inversion	89
6.4	Rendering Pipeline	90
6.4.1	Rendering for HDR Application	92
6.4.2	Rendering for Focal Stack Application	93
6.5	High-Resolution Capture and Offline Processing	93
6.5.1	Homogeneous Edit Propagation	94
6.5.2	Manifold-Preserving Edit Propagation	95
6.5.3	Focal Stack Compositing	96
7	Results and Discussion	97
7.1	Appearance-based HDR Acquisition	97
7.2	Appearance-based Focal Stack Acquisition	98

7.3	Live Video Editing	98
7.4	Performance	103
7.4.1	Discussion	105
8	Conclusions	106
8.1	Usability Issues	107
8.2	Algorithmic Issues	108
8.3	Extensions	110
8.4	Closing Remarks	111
A	GLSL Shader for HDR Application	113
B	Fast Quantization Source Code	116
	Bibliography	119

List of Tables

2.1	Comparison of Gaussian-filtering schemes	13
2.2	Performance of Gaussian-filtering schemes	14
3.1	Runtime comparison for various lattice lookup methods	38
4.1	Runtime comparison for various edit propagation methods	58
6.1	Table of scale factors for correcting the change in field of view as a function of focus	87
7.1	Runtime on both laptop and tablet implementations for the HDR ap- plication	103
7.2	Breakdown of the runtime on a Tegra4 tablet for viewfinder editing .	104

List of Figures

1.1	Examples of non-WYSIWYG computational photography	3
2.1	Examples of Gaussian filtering	10
2.2	Examples of denoising with Gaussian filtering	11
2.3	High-dynamic-range imaging pipeline	16
2.4	Focal stack compositing	18
2.5	An example of edit propagation	20
3.1	The A_d^* lattice at $d = 2$	25
3.2	Sublattice structure of the permutohedral lattice	26
3.3	Resampling kernel for splatting and slicing in 2D	28
3.4	Illustration of linear-time quantization in A_d	29
3.5	Blurring kernels for the permutohedral lattice in 2D	29
3.6	Overall kernel for Gaussian filtering in the permutohedral lattice in 2D	31
3.7	Loglinear quantization in the permutohedral lattice	32
3.8	Visualization of the linear-time quantization algorithm for the permutohedral lattice	36
3.9	Illustration of the effect of avoiding explicitly finding the anchor point in quantization	38
4.1	Interface for viewfinder editing	46
4.2	Summary of popular descriptors in the literature	48
4.3	Principal components of image patches	49
4.4	Steerable filters	51

4.5	Datasets for descriptor evaluation	53
4.6	Subsampled edit mask	55
4.7	Multi-scale Texture Lookup	57
5.1	Comparison against existing multi-exposure metering methods	62
5.2	Appearance-based metering via per-pixel analysis	65
5.3	The thin-lens model	68
5.4	Appearance-based focusing via per-pixel analysis	70
5.5	Synthetic comparison of appearance-based HDR metering against Hasinoff et al. on church dataset	73
5.6	The distribution of noise in appearance-based metering	74
5.7	Synthetic comparison of appearance-based HDR metering against Hasinoff et al. on flower dataset	75
5.8	Comparison of appearance-based HDR metering against Hasinoff et al. on a moving scene	76
5.9	Empirical comparison of appearance-based HDR metering against generic histogram-based HDR metering on a static scene	79
5.10	Empirical comparison of appearance-based HDR metering against generic histogram-based HDR metering on a dynamic scene	80
5.11	The real-time effect of local tonal edits on appearance-based metering	81
6.1	Camera hardware	83
6.2	Camera pipelines for the HDR application and the focal stack application	84
6.3	The state machine for the viewfinder editing applications	88
6.4	A demonstration of colorization of the scene	90
6.5	The rendering pipeline	91
6.6	Offline processing for improving edit mask quality	95
7.1	Additional results of HDR capture with viewfinder editing	99
7.2	Focus edit for the racer scene	100
7.3	Focus edit for the lego scene	101
7.4	Edit propagation on a live video	102

List of Algorithms

3.1	Splatting in the permutohedral lattice	27
3.2	Slicing in the permutohedral Lattice	30
3.3	Algorithm for quantizing a vector $\vec{x} \in \mathbb{R}^{d+1}$ in A_d^*	35
3.4	A faster algorithm for quantizing a vector $\vec{x} \in \mathbb{R}^{d+1}$ in A_d^*	37
5.1	A dynamic programming routine for solving the global objective function for the optimal exposure in linear time.	67

Chapter 1

Introduction

Taking photos and videos has become an integral part of our daily lives. In addition to the 100 million digital cameras sold annually [1], over a billion camera phones are purchased every year [2], leading to the astounding ubiquity of cameras all around the world. Photography has been extremely successful commercially, owing to the affordability, the ease of use, and interactivity of modern cameras. More importantly, however, photography has enjoyed an immense *cultural* success, as the act and experience of photography, content creation and sharing resonate deeply with today's masses. For instance, more than 300 million digital photographs were uploaded to Facebook[®] every day in 2012 [3].

The rise of the popularity of digital photography coincided with, or perhaps was aided by, the recent advances in the capability of mobile devices. In the research community, the field of computational photography gained prominence: by combining advances in optics, capture strategy, image processing and computer vision, researchers could enhance the photographs produced by cameras, or enable entirely new kinds of photography or photography experience. Many published techniques rely on modifying the capture hardware or capture strategy to acquire coded data, and then computationally decoding it to produce a photograph as an offline process.

However, the adoption and popularization of these new techniques lag behind the progress in academia, and the techniques that have made the jump do not provide experiences that are as compelling as they can be. While many culprits exist, such as

the limited programmability of mobile devices [4], one reason is the loss of interactivity from the aforementioned offline process, which robs photography of the key ingredient of its present success: the WYSIWYG (what-you-see-is-what-you-get) paradigm.

1.1 The WYSIWYG Paradigm

The phrase *what you see is what you get* as it applies to user interfaces was first coined in 1982 by Larry Sinclair, and was popularized in its acronymous form: *WYSIWYG*. It indicates that the user's interaction with his medium in a content-creation process is aided by a close visual approximation of the produced content. The WYSIWYG property enables the user to immediately see the effect of his actions and respond accordingly, leading to a more efficient and more enjoyable process.

Photography has not always been a WYSIWYG experience. In the days of film photography, the final product was influenced by a multitude of factors, ranging from the capture-time parameters like exposure, gain, aperture, focus and focal length, to the variables throughout the development process like the choice of film, paper, and the amount of dodging and burning. The photographer relied on the optical viewfinder and drew from his experience to imagine in his mind how the developed photograph would look. Only with the introduction of the digital cameras with electronic viewfinders, photography became WYSIWYG and thus accessible to the masses. An explosion in the popularity of digital photography ensued, riding the waves of affordable cameras, advances in camera phones, and the rise of social media. With WYSIWYG photography, the photographer has not only the ability to shape the current shot interactively, but also the power to decide how many shots are taken, and whether photos will be taken at all.

Running counter to the notion of WYSIWYG photography is the relatively new field of computational photography, where computation is applied to images before and/or after their capture in order to enhance or extend digital photography. Over time, two particularly effective paradigms have emerged. In the first paradigm, the imaging device is modified to capture *coded data*—it may be optically, spatially or temporally coded or multiplexed—through the use of novel optics or unconventional

capture strategy. This data may in fact not look like a good photograph, or be recognizable as a photograph. See Figure 1.1 for examples of such coded data. Next, the data is decoded computationally in order to yield the output photograph, perhaps under the user’s guidance. The output photograph, compared to a conventional photograph, is enhanced in some dimension: depth of field, dynamic range, signal-to-noise ratio, angular resolution, motion blur, and so forth. In the second paradigm, the photograph is captured normally, and then edited in post-processing by the user in a nonlinear fashion: motion may be magnified or suppressed, or the appearance of objects may be selectively altered.

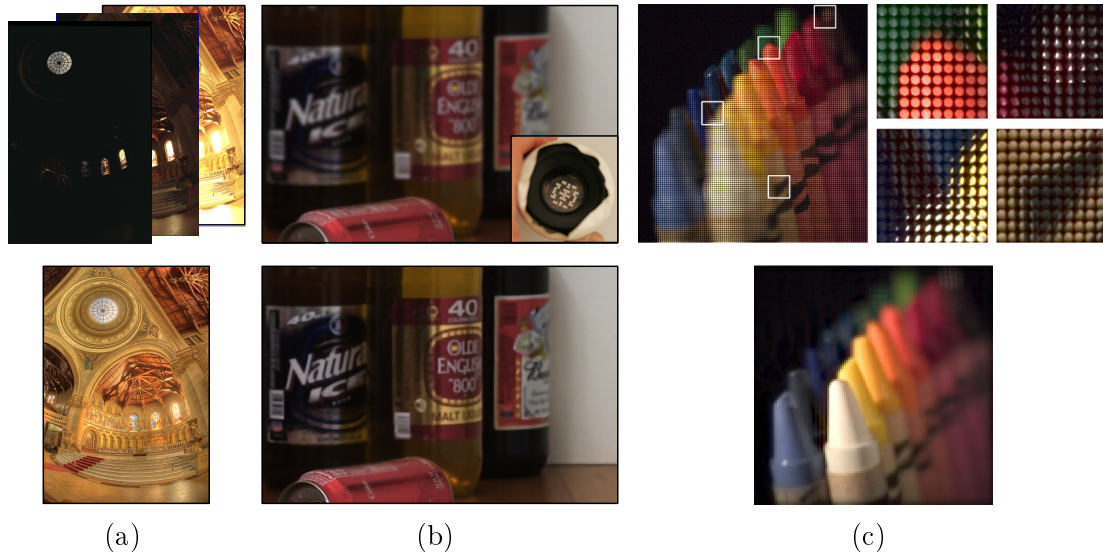


Figure 1.1: Examples of non-WYSIWYG computational photography. Many computational photography techniques first capture a coded-representation of the scene, and decode it later computationally. The coded representation typically does not fully indicate to the user what the output will look like. For each example, the top row shows the data captured by the sensor; the bottom row shows the computationally decoded output. **(a)**: a multi-exposure stack for high-dynamic-range (HDR) imaging [5]. **(b)**: coded aperture imaging for extending the depth of field [6], **(c)**: imaging with a microlens array for capturing a light field [7].

Unfortunately, these paradigms of computational photography contradict the WYSIWYG aspect of digital photography: what the user sees through the viewfinder is often not what he gets at the end, since the user must extract the result and apply the

decoding and/or editing operations offline. In addition to rendering the photographing exercise less interactive, the loss of WYSIWYG property can lead to inadequate or excessive data capture, as the user is left on his own to visualize the final result without any aid.

For some computational photography applications, this problem is solved by providing additional computational resources—more compute, more storage, higher bandwidth—thereby bringing the offline process online. By the virtue of Moore’s Law, this may well happen in near future. However, for other applications that require user guidance, the loss of WYSIWYG property will not be solved simply with additional resources, since the user must still provide his input in a post-process. Fortunately, for a subset of these difficult cases, a well-engineered end-to-end system armed with new class of algorithms can provide a WYSIWYG interface at present time by treating the viewfinder as a canvas, as will be described in this dissertation.

1.2 The Viewfinder as a Canvas

The electronic viewfinder in the camera, since its inception, has been primarily an output device: the scene as captured by the camera sensor is streamed onto the LCD screen, which the photographer uses to frame his shot. With recent proliferation of touch-enabled displays (e.g. smartphones, tablets and many compact point-and-shoot cameras), the screen has taken on a new role as a potential input device: many cameras now sport a touch-based user interface for accessing application menus, selecting settings, or actuating the shutter. Some applications even go further, allowing the user to select the control points for auto-focus, auto-exposure or auto-white balance. While such uses of the screen as an input modality are significant, the progress thus far simply replicates the functionality available with physical controls (dials, knobs, tactile buttons) onto a touch-enabled display.

Once the photograph is taken, the screen can be used as an image-editing interface in a disparate application. Many image-editing software packages and applications exist on mobile devices, ranging from simple global filters like Instagram[®] to more powerful suites like Adobe Photoshop Touch[®] or specialized software that

create cinemagraphs. In these instances, however, the screen is no longer acting as a viewfinder, since it cannot affect the capture process. In summary, the ability to provide freeform input or strokes on a touch-enabled display is not being leveraged on a camera viewfinder.

Using the touch-enabled electronic viewfinder as a live canvas is an intriguing direction for two reasons. For one, further interactivity in photography is a positive trait, as evidenced by the popularity and the diversity of image-manipulation programs: the users want to interact more with their photographs and videos. Secondly, performing operations directly on the viewfinder lets the user review the result immediately and react accordingly—he may adjust the camera parameters or change the framing. For instance, the Flickr[®] smartphone application recently began offering live previews of the Instagram-style filters on the viewfinder. Not only does this help the user create the result he wants, pre-loading the offline editing process to capture time can provide a more engaging photography experience. Sharing the resulting photos also becomes streamlined. Lastly, it is worth noting that many recent image manipulation algorithms already take sparse strokes as inputs, and would translate nicely onto a touch-enabled viewfinder.

It is worthwhile to note that performing live edits directly on the viewfinder requires the situation to be cooperative. For instance, it would need a cooperative subject or scene who should stay relatively still while the edits are being performed. It also requires a bright screen for the viewfinder, or a cooperative viewing environment. These limitations are further explored in Chapter 8.

1.3 Contributions

This dissertation presents the design and the implementation of an end-to-end system for mobile digital photography that enables WYSIWYG interfaces to three computational photography applications, along with the underlying mathematical algorithms and a unified user interface for all applications, called *viewfinder editing*. The three applications are high-dynamic-range (HDR) imaging via exposure stacks, focal stack composition, and stroke-based image and video editing.

Within the presented system, the main contributions are as follows:

- Improvements to a sparse data structure called the *permutohedral lattice*, for use in real-time spatiotemporal edit propagation.
- An algorithm and a stroke-based user interface for performing viewfinder editing on a mobile device. That is, edits propagate spatiotemporally through a live viewfinder, providing a WYSIWYG experience for the first time ever for a large class of operations.
- Camera control algorithms that, based on the user interaction with the above interface, chooses the optimal parameters for an exposure stack or a focal stack, to be used for HDR imaging or focal stack composition, respectively. The proposed algorithms differ fundamentally from the existing ones because of the WYSIWYG property of the system.

While these contributions altogether constitute a coherent system, each of them addresses an existing problem in the field of computer graphics, and has individual applicability beyond the framework proposed in this dissertation.

The rest of the dissertation begins with an overview of the existing state of art on the various subproblems tackled, described in Chapter 2. Chapter 3 introduces the permutohedral lattice, previously described in several publications the author has been involved in [8, 9, 10]. The chapter showcases new developments on the permutohedral lattice, including a crucial algorithmic improvement and others that are tailored towards the use in viewfinder editing.

Chapter 4 lays out the framework for viewfinder editing and the underlying algorithms for spatiotemporal propagation. Several design choices are discussed and explained. Then, Chapter 5 defines the camera control algorithms for HDR imaging and focal stack composition, that account for and make use of viewfinder editing. Chapter 6 brings together the components described up to that point, and describe a system for mobile photography that make use of these contributions. Chapter 7 demonstrates some of the results of the applications, along with performance evaluations and discussions. Chapter 8 concludes the dissertation, discussing the uses and limits of the proposed system, and offering suggestions on future work.

The viewfinder editing paradigm and the proposed system were also discussed in a paper published in 2013 at the ACM SIGGRAPH Asia conference [11]. The work contained in this dissertation can be considered to be an in-depth look at the more recent iterations of the system.

Chapter 2

Prior Work

By assembling an end-to-end system for WYSIWYG computational photography, this dissertation tackles a number of well-studied problems in computer graphics: the first is the problem of fast Gaussian filtering of high-dimensional data, commonly employed in computer vision and computational photography applications. The second is the problem of spatiotemporally propagating edits on images and video sequences, in order to simplify and accelerate editing tasks. The third is the problem of setting camera parameters for stack-based image acquisition, now popular in many computational photography techniques. These three problems will be considered and addressed in the course of this dissertation, respectively in Chapters 3, 4 and 5. This chapter contains a section devoted to the history of each of these three problems.

Fast Gaussian filtering is a generalized framework that can express many well-known image processing operations, such as the bilateral filter [12, 13], joint bilateral filter [14] and non-local means [15]. These filters have gained widespread use because of their edge-aware property and mathematical elegance. Accelerating them and other edge-aware filters has been a key problem in the graphics community recently.

Spatiotemporal edit propagation applies Gaussian filtering or other edge-aware operations in order to convert sparse edits on an image onto dense edits on the same image or onto other image sequences. The advances in this task are closely related to those in edge-aware image processing, and existing work has dealt with both acceleration and propagation quality.

The last section discusses the determination of camera parameters for computational photography applications that capture a stack of photos. Whereas setting the exposure, focus and gain is relatively straightforward for a conventional camera (albeit subjective), applying the same to a stack acquisition is a nontrivial multi-dimensional problem, and owes the recent attention to the rising popularity of HDR photography.

2.1 Accelerating Gaussian Filtering

Gaussian filtering is a general framework for smoothing multi-dimensional signal represented by discrete samples. Let f be an abstract high-dimensional, vector-valued function defined over \mathbb{R}^{d_p} . Formally, $f : \mathbb{R}^{d_p} \rightarrow \mathbb{R}^{d_v}$ for some natural numbers $d_p, d_v \in \mathbb{Z}^+$ indicating the dimensionalities of the underlying spaces. Let (\vec{p}_i, \vec{v}_i) for $i = 1, \dots, n$ be known samples of f , i.e. $f(\vec{p}_i) = \vec{v}_i$. Then, we can compute a smoothed representation $g : \mathbb{R}^{d_p} \rightarrow \mathbb{R}^{d_v}$ defined as a normalized weighted sum of known samples of f :

$$g : \vec{x} \mapsto \frac{\sum_{i=1}^n w_i(\vec{x}) \vec{v}_i}{\sum_{i=1}^n w_i(\vec{x})}, \quad (2.1)$$

where the weights are defined with exponential dropoff with respect to the squared distance from the sample, i.e. a Gaussian kernel:

$$\forall \vec{x} \in \mathbb{R}^{d_p}, \forall i \in \{1, \dots, n\}, \quad w_i(\vec{x}) := \exp \{ -\|\vec{p}_i - \vec{x}\|^2 \}. \quad (2.2)$$

In the scientific computing literature, the operation in Equation (2.1) is also known as the Gauss transform [16]. As can be seen from the formula, g converges to a Gaussian blur of f with an increase in the density of the samples. The formula for the weights can be optionally fitted with a parameter to control the exponential dropoff, but it can also be achieved by simply scaling the space in which the function is defined.

Gaussian filtering is a powerful tool that can express many popular image-processing operations. Given an image I , the well-known Gaussian blur can be formulated by

setting \vec{p}_i to be the (x, y) spatial coordinate and \vec{v}_i be the pixel value at the corresponding location. Simply put, pixels are mixed together with other pixels that are close spatially, with weights that decay exponentially as a function of the squared distance.

Similarly, the edge-preserving bilateral filter [12, 13] for grayscale images is specified by setting $\vec{p}_i = \{x_i, y_i, l_i\}$ and $\vec{v}_i = \{l_i\}$ where (x_i, y_i, l_i) correspond to the two spatial coordinates and the luminance at the said location. In this formulation, pixels are mixed with other pixels that are close by both spatially and radiometrically. Color bilateral filter is obtained by replacing l_i with the RGB value (r_i, g_i, b_i) . A more sophisticated smoothing algorithm is obtained by replacing l_i with a local image descriptor at the given pixel [15]. Joint bilateral filter [14] can also be expressed in this framework by drawing the position vectors p_i from a source different from the image being filtered, e.g. $p_i = \{x_i, y_i, r'_i, g'_i, b'_i\}$. See Figures 2.1 and 2.2 for some examples of these operations.

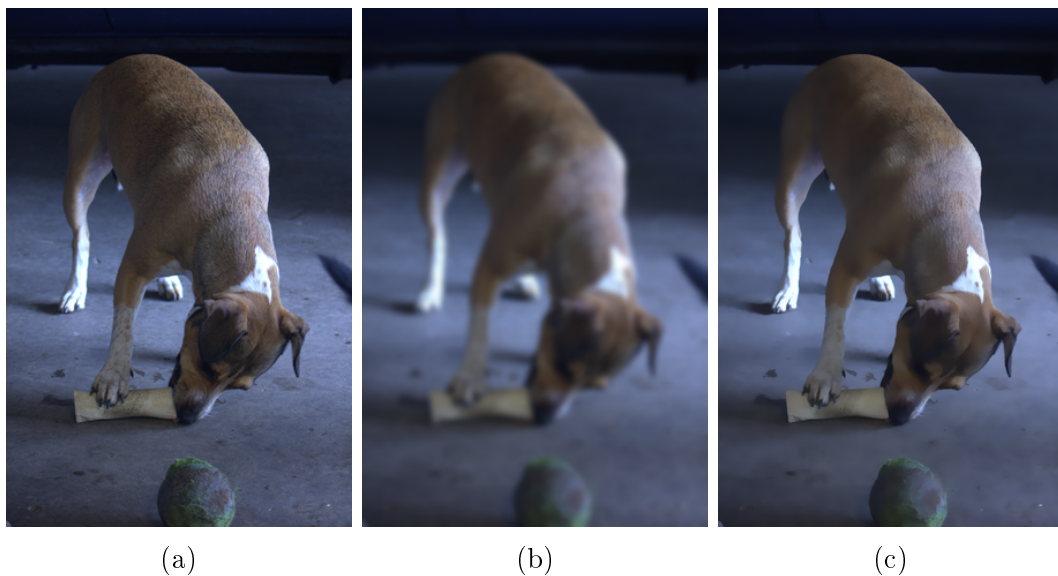


Figure 2.1: Examples of Gaussian filtering. **(a)**: The input. **(b)**: The output of a Gaussian blur ($d_p = 2$). **(c)**: The output of a bilateral filter ($d_p = 5$). Note that surfaces have been smoothed without crossing strong edges. The images are best viewed electronically, to permit zooming.

The direct evaluation of Gaussian filters on a $w \times h$ image incurs a computational

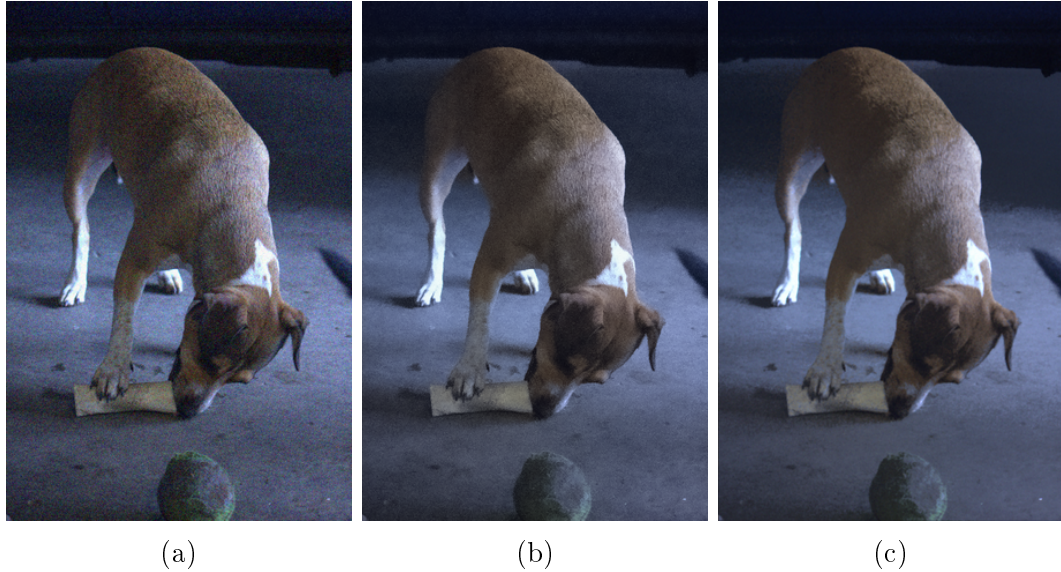


Figure 2.2: Examples of denoising with Gaussian filtering. **(a)**: The input image from Figure 2.1, corrupted with Gaussian noise. **(b)**: The output of a joint-bilateral filter on the noisy input ($d_p = 5$). The noisy image was filtered with respect to the positions of the luma channel. **(c)**: The output of a non-local means filter on the noisy input ($d_p = 8$). By mixing patches of similar appearance, noise has been effectively suppressed. In comparison to (b), it exhibits less noise, but smooth intensity variations in the image have become more piecewise. Either output may be the more appropriate one, depending on the user intention.

cost of $O(w^2h^2)$, resulting from the pairwise interaction of wh pixels with one another. For large images ($w, h \geq 100$, for instance), this can be prohibitively expensive. The need for more efficient algorithms for evaluating Gaussian filters has spurred a number of signal-processing-based approaches that approximate the filter response.

2.1.1 Signal-Processing Approach

While the high-dimensional function $f : \mathbb{R}^{d_p} \rightarrow \mathbb{R}^{d_v}$ is often specified by a set of n samples, the underlying signal can be represented much more compactly, especially since the function is to be blurred. Taking advantage of this, several existing methods follow the same paradigm to accelerate Gaussian filtering:

1. Resample the signal onto a data structure. This data structure is a collection

of samples of f , and the samples may be organized as a grid, a lattice, a tree or a general point-set, for instance. This step is called *splatting*. After this step, each vertex in the data structure roughly corresponds to the aggregate of nearby input points.

2. Blur the signal within the data structure. This step is specific to the particular data structure in question, but in general, the vertices in the data structure are blurred with nearby vertices. This step is called *blurring*.
3. Finally, resample the signal out of the data structure. This step is called *slicing*. Often the slicing step mirrors the splatting step.

This paradigm was first formalized by Paris et al. [17] using a regular d_p -dimensional grid. In this work, the underlying d_p -space is partitioned by a regular grid, and the vertices of the grid are stored as a high-dimensional array. Then, the splatting step spreads each input point onto the vertices of the hypercube enclosing the input point, using multilinear weights. The blurring step consists of applying a separable kernel of finite width to the data structure. The slicing step blends the values stored at the vertices of the same hypercube using the multilinear weights.

Adams et al. [8] uses the d_p -dimensional permutohedral lattice as the data structure. Because the permutohedral lattice tessellates the space with simplices, rather than hypercubes, barycentric interpolation is used in splatting and slicing step. The blurring step is also performed in a separable fashion. See Chapter 3 for details.

The multi-pole method of Greengard and Strain [16] clusters the underlying space by examining the distribution of the input points, and represents the space by a vertex located at the center of each cluster. The splatting stores each input point into the vertex corresponding to the cluster to which it belongs: hence, the vertex is a proxy for all input points in the cluster. Then, to slice, the influence of each cluster on the query point is computed. Yang et al. [18] improves upon this method by organizing the clusters in a tree.

Lastly, Adams et al. [19] employs a d_p -dimensional kd-tree built upon the input dataset, and uses the nodes of the kd-tree to store samples. The splatting, blurring and slicing steps are achieved by a probabilistic query into the kd-tree. The queries

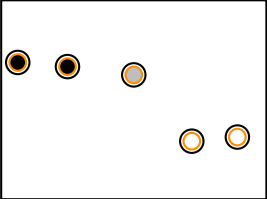
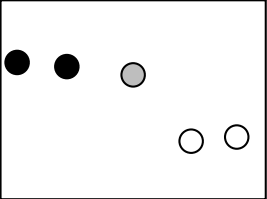
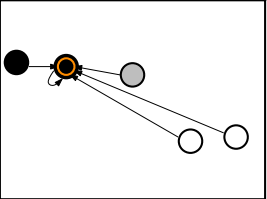
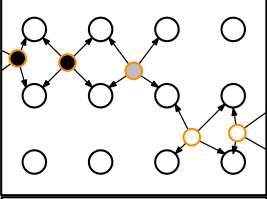
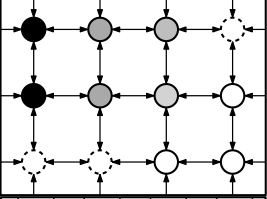
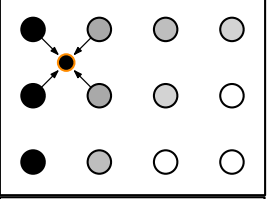
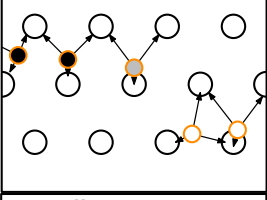
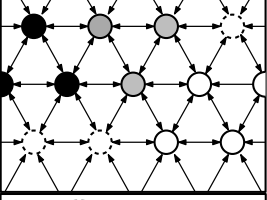
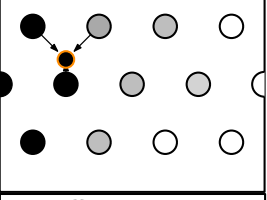
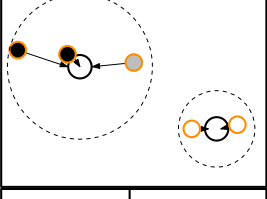
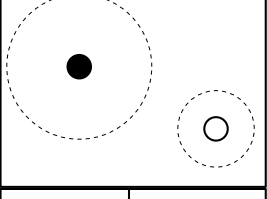
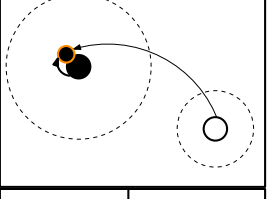
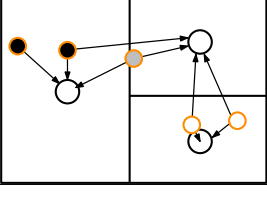
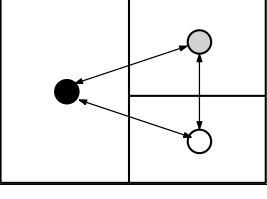
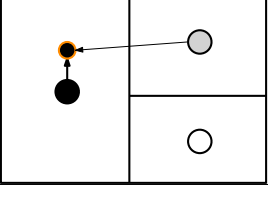
Method	Filtering stages (Splatting / Blurring / Slicing)		
Pointset			
Regular grid [17]			
Lattice [8]			
Multi-pole [18]			
kd-tree [19]			

Table 2.1: Comparison of Gaussian-filtering schemes. For each method, the splatting, blurring, and slicing steps are illustrated from left to right. The orange-rimmed circles in the first column represent the input point. The larger circles represent vertices in the data structure. The grayscale color of each circle indicate its value. In the splatting step, the input points are resampled onto the data structure. In the blurring step, the vertices are filtered. In the slicing step, for each query point, nearby vertices are sampled in some fashion. For instance, a naive approach would be to use the positions of the input points as the vertices, and compute the pairwise weights in the slicing step. See Section 2.1.1 for descriptions of other methods.

Method	Time complexity	Space complexity
Pointset	$O(d \cdot n^2)$	$O(d \cdot n)$
Regular grid [17]	$O(2^d \cdot n)$	$O(2^d \cdot n)$
Lattice [8]	$O(d^2 \cdot n)$	$O(d^2 \cdot n)$
Multi-pole [18]	$O(d^c \cdot n \log k)$	$O(d \cdot n + d^c \cdot k)$
kd-tree [19]	$O(d \cdot n \log n)$	$O(d \cdot n)$

Table 2.2: Comparison of the theoretical time and space complexity of Gaussian-filtering schemes. Here n is the cardinality of the input dataset, and d is the dimensionality of the position vectors. For the multi-pole-based fast Gauss transform [18], c and k are tunable parameters.

proceed in parallel, branching when necessary probabilistically such that the chance of reaching a certain vertex is proportional to the Gaussian weight.

See Figure 2.1 for a visual summary of the aforementioned methods. Table 2.1 tabulates the space and time complexity of these methods.

2.2 Camera Control for Computational Photography

Camera control for the conventional digital photography consists of the choice of exposure, gain, focus, zoom and aperture [20]. In addition to these technical parameters, the photographer can exercise his judgment on the physical distance to the subject, the perspective of the view frustum, the composition, and the timing of the shutter press. Picking the ideal capture parameters in case of a single photograph is fairly straightforward: automatic focus, exposure, gain, and aperture settings have been available on consumer-grade digital cameras for decades [21, 22], and their principles are well-understood, even though their implementations may only be documented by patents written in very obscure terms. In contrast, many computational photography algorithms make use of stack photography. This compounds the difficulty of automatic parameter selection, since each photograph in the stack must have its parameters determined. In addition, these algorithms are not WYSIWYG, so the user

is left to visualize the post-processed output and to guess whether the stack acquired is sufficient. The same problem exists in case of nonlinear image editing.

2.2.1 High Dynamic Range Imaging

One of the most popular computational photography technique is high-dynamic-range (HDR) imaging [23], which typically combines multiple photographs at varying exposures to synthetically increase the dynamic range of the output [5, 24]. As such, HDR imaging requires the determination of the exposure levels for the individual frames, with the assumption that each region of the scene should be well-imaged by at least one of the frames.

While HDR imaging can produce high-quality result for static scenery, existing methods typically do not handle scene and camera motion gracefully. Handheld HDR imaging requires registration and alignment of the frames, which can be done based on the image content [25] or an inertial measurement unit [26], but this only accounts for the camera motion and does not solve parallax issues. With respect to scene motion, motion analysis can help reduce ghosting artifact at the cost of discarding data [27]. Otherwise, expensive non-rigid registration is required [28], which still may have issues with occlusion and disocclusion. Therefore, reducing the number of and the duration of the photographs is desirable, as opposed to a full exposure stack—for instance, the iconic Memorial Church dataset was based on 16 images [5] and required a tripod.

The simplest method for multiple-exposure metering is exposure bracketing [23], in which the scene is captured once using a standard auto-exposure algorithm, and additional photos are acquired with a fixed number of stops offset from the first (e.g. ± 2 stops.) The state-of-the-art methods for multiple-exposure metering attempt to maximize the signal-to-noise ratio (SNR) of the resulting HDR composite. Hasinoff et al. [29] derives the mathematical formula for SNR of each pixel as a function of the exposure values, and solves for the set of exposures that maximizes the worst SNR among all the pixels, given a time budget. Conversely, the minimal time budget that produces the SNRs with the desired lower bound can also be found. Gallo et

al. [30] similarly optimizes for the average SNR over the pixels. In both cases, the optimization problem can be reformulated as a problem on the HDR histogram of the scene, which allows quicker evaluation of the objective function.

It is important to note that the dynamic range of display devices, whether they be electronic screens or printers, is also limited. Consequently, HDR composites typically undergo tone-mapping process [31] which compresses the dynamic range of the data to fit the range that the display is capable of reproducing. Figure 2.3 illustrates this pipeline. Ironically, this stage discards information, contrary to the ideals of the acquisition algorithms. See Čadík et al. [32] for a comparison and evaluation of common tone-mapping operators.

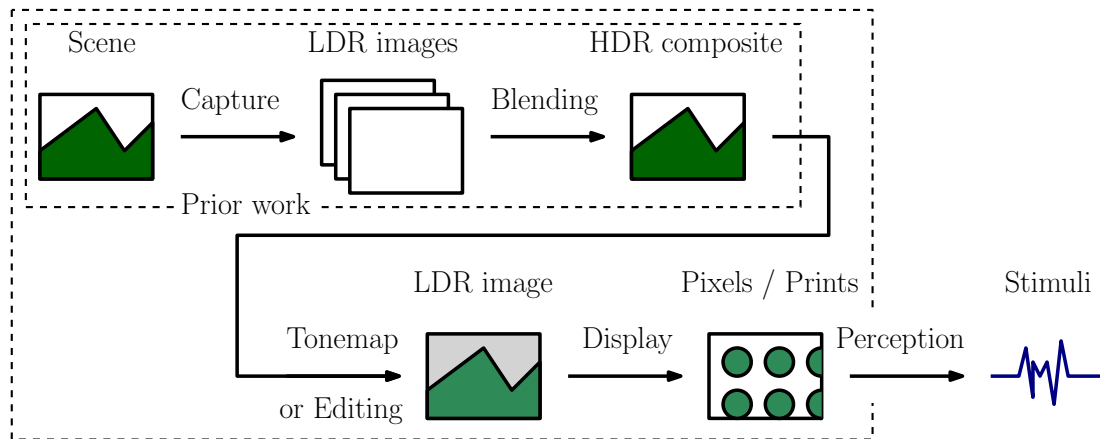


Figure 2.3: A typical high-dynamic-range (HDR) imaging pipeline. Multiple photos of the scene are taken at varying exposure levels, and are combined to form the HDR composite. Existing work on multiple-exposure metering considers these stages (marked with the inner dashed box) only in determining the appropriate exposures. In this dissertation, tonemapping and editing post-capture are assumed known, via the WYSIWYG property being pursued, and are accounted for in the metering process.

In most traditional workflows, the dynamic range of the target device and the tonemapping operator to be used are not known a priori, and as such, choosing exposure parameters in a manner that most faithfully record the raw data is the rational approach. However, in a camera interface that strives to deliver stack-based computational photography and image editing in a WYSIWYG manner, the rest of the pipeline must be known by definition, in order to produce the appropriate

visualization in the viewfinder. This fact can be exploited to close the loop, by accounting for the tonemapping and other transforms in the metering process.

2.2.2 Focal Stack Photography

In focal stack photography, multiple photographs are captured with the lens focused at different depths. These photographs can be stitched together to create an all-focus composite: the in-focus regions of each photograph were combined together [33] in case a single photograph is unable to image the entirety of the scene sharply because of the limited depth of field (DOF) [34]. For instance, macro photography typically suffers from a very narrow depth of field, and all-focus imaging is commonly used to overcome this problem and synthetically extend the depth of field of the output. While simply stopping down the aperture can extend the depth of field as well, the image quality and SNR suffer [35]. Other methods for extending the depth of field exist, such as focus sweep [36], coded aperture [6, 37] and custom optics [38, 39], but unlike others, focal stack photography does not require modification of the camera hardware or expensive and artifact-prone deblurring.

Focal stacks suffer from the same problem as exposure-stack photography; that is, registration of the individual frames is required to handle camera motion and scene motion. Non-rigid registration for images with varying defocus blur appears to be an unattempted problem. There is currently no literature on separating motion blur from defocus blur, as existing motion-deblurring research assumes that the unknown scene content has high contrast and contains sharp edges [40].

Hasinoff and Kutulakos [41] give a detailed analysis of the use of focal stack for extending the depth of field, including the computation of the optimal (minimal) set of focus distances given the target aperture size to simulate. However, this method is scene-independent. In contrast, Vaquero et al. [42] first analyzes the scene content by preliminarily sweeping the scene. Once the depths at which objects appear are identified, a minimal focal stack is acquired at these depths.

The compositing process for all-focus imaging can be generalized in order to reduce the depth of field, simulate tilt-shift lens, or even visualize non-physical depth

of field [43]. For instance, Jacobs et al. [43] simulates a wider aperture by using the slice focused on the foreground to compute the foreground pixels, and using the slice focused *closer* to compute the background pixels. See Figure 2.4 for an illustration. This method of composition produces better results compared to synthesizing defocus blur in the image space with a single image [44], since the latter cannot properly reproduce certain photographic effects, such as highlights or contrast inversion [45]. Jacobs et al. also provides a stroke-based user interface for manipulating the composition after acquisition. For this application, a full focal stack of 16 or 32 slices were captured on a tripod.

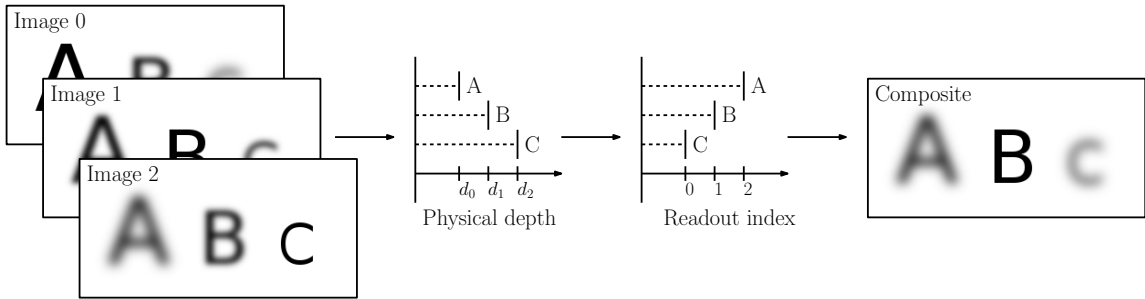


Figure 2.4: An example of focal stack compositing: reduced depth of field. Multiple photos of the scene are taken at varying depths. **Left:** In this toy example, the scene consists of three letters A , B and C , with A being the closest. Three images are taken at the respective depth. **Middle:** A depthmap can be constructed from the stack. To simulate a lens focused at the middle letter with a wide aperture, pixels are read out at the depth *on the opposite side* of the reference plane. For instance, the letter A is read out from Image 2, whereas it is actually the sharpest in Image 0. **Right:** The final composite. Note that by flipping the readout depth, a considerably greater defocus is achieved in comparison to the input image focused at the same depth (Image 1).

2.2.3 Composition

One capture parameter that is difficult to algorithmically optimize is composition, as it is highly subjective and personal. There exists a general guide such as the rule of the third [20], but the precise choice of perspective, distance and timing are up to the photographer. Bae et al. [46] offers a specialized tool for helping users recreate

the composition of a historical photograph, but no general algorithm exists for aiding composition at capture time.

To some extent, the composition can be altered after the photograph is taken. For instance, they can be cropped manually, or even automatically to obey standard composition rules [47]. However, cropping discards data and is inherently hamstrung by the composition of the original image. Even the viewpoint can be changed slightly [48], but such techniques require inpainting or other texture synthesis methods, which can create objectionable artifacts.

Lucky imaging [49] can be considered a form of composition assist: a burst of images can be captured, and the user may choose the one that captures the moment the best, e.g. a fleeting smile, or even combine the burst of images into a composite [33]. However, these methods work only off-line. Some commercial point-and-shoot cameras employ face detection [50] and alert the user when the subject smiles.

2.3 Edit Propagation

Edit propagation algorithms allow the user to specify a sparse set of edits or selections and automatically propagate them onto the rest of the image or other frames. They are useful for reducing the amount of user interventions in an editing or selecting task, and are essential for input modalities such as smartphones or tablets for which pixel-perfect specification of edits is infeasible. Edit propagation is applicable to many other image processing tasks, with examples ranging from colorization [51], image composition [33] to tone management [52]. See Figure 2.5 as an example application.

Alpha matting [53, 54] can be considered to be the predecessor to edit propagation. Many matting algorithms take as input a trimap, a mask specifying known foreground and known background regions. Then the matting algorithm classifies the remaining regions to either foreground ($\alpha = 1$) or background ($\alpha = 0$) with possible fractional alpha values between 0 and 1 [55, 56].

There are two large categories of edit propagation algorithms. One transmits edit information from each pixel to its neighbors in an edge-aware fashion, similar to

anisotropic diffusion [57]. The other is based on detecting regions that have appearance similar to that of the user-edited area.



Figure 2.5: An example of edit propagation. **Left:** The input is a grayscale image, along with sparse edits specified as strokes. **Right:** The edit propagation propagates the edits to regions of similar appearance. Typically each pixel is considered to be a mixture of the known classes of exemplars, and corresponding edits are applied with the same coefficients in the mixture. This particular result is with the algorithm of Levin et al. [51].

2.3.1 Edge-Aware Smoothing

Propagating edits or selection can be formulated explicitly as a diffusion process. A cost function on the image gradient is used to build a linear system that diffuses edits [51, 52], which can be solved either directly, iteratively or in a multi-scale fashion. Other approaches exist, such as to use the gradients to drive graph cut [33], or to perform wavelet decomposition with coefficients based on the image gradient [58].

Since processing each pixel requires access to only its immediate neighbors, these algorithms can be extremely fast. For instance, edge-avoiding wavelets [58] or domain transform [59] can process VGA-resolution images in the order of milliseconds. As a consequence, however, these methods cannot propagate information onto spatially discontinuous regions, and are not robust against occlusions or camera shakes, which can break temporal contiguity.

2.3.2 Appearance-based Smoothing

Several competing algorithms enforce the notion that regions of similar appearance should have similar edit or selection values, by modeling edits as a function over the space of patch appearance. The function can be recovered in various ways. For instance, An and Pellacini [60] and Farbman et al. [61] consider the pairwise interaction between all pixels and computes the diffusion amongst pixels. Chen et al. [62] models each pixel as a linear combination of nearby pixels, and applies the same weights onto the edited values to form a linear system to solve. Several papers take a machine-learning approach and treat the problem explicitly as a clustering of all pixels in the image [63, 64]. Adams et al. [8] samples the space of the local descriptor of each image patch. An explicit clustering of the patches can be done as well [65, 66].

So far, no existing edit propagation algorithms has tackled the problem of propagating edits directly on the viewfinder. Besides the technical challenge involved, one caveat is that propagating edits on the viewfinder in real time is distinct from the problem of propagating edits on an image sequence as an offline process. Methods discussed thus far for the latter explicitly assume that the input sequence is given in its entirety, and rely on a preprocessing step that statically analyses the whole sequence and builds expensive classifiers, whose costs are amortized over the multiple frames to be processed or the multiple edits to be applied [60, 61, 63, 64, 65] for acceleration. For viewfinder editing, such approaches are problematic because the user edits and the viewfinder content evolve over time, and would require continuously rebuilding the classifier.

2.3.3 Note on User Interface

It is worthwhile to note that most of the modern stroke-based edit propagation algorithms require the user to specify at least two types of edits: positive edits to mark exemplars that the user wants edited, and negative edits to mark exemplars that the user does not want edited. While useful for specifying the edit precisely, this requirement can complicate and protract the user interaction necessary.

When the edit propagation is fast enough to run real-time on a live viewfinder,

however, user feedback and correction become possible. Therefore, this dissertation relies only on positive exemplars and propagate conservatively: if the user finds that the edits do not propagate enough, he can add additional positive exemplars easily.

In fact, this can be observed in existing touch-based image-editing algorithms on mobile devices, which are fast enough to run in real time on a still image. Liang et al. [67] computes an edit mask based on the bilateral distance to the pixel the user touches—hence only a single positive exemplar is used. The extent of propagation is controlled by the parameter to the underlying bilateral filter, which is controlled by a user swipe away from the point of the initial touch. A commercial software [68] available on smartphones operates on the same principle.

2.3.4 Other Alternatives based on Tracking

There may be other alternatives for spatiotemporal propagation based on explicit tracking. There is extensive literature available on both two-dimensional tracking in image space, such as Kanade-Lucas-Tomasi (KLT) feature tracker [69, 70], and three-dimensional tracking in world space, such as Simultaneous Localization and Mapping (SLAM) [71]. Two-dimensional tracking is significantly faster, but cannot handle nonplanar distortions like parallax or occlusion. Three-dimensional tracking has been widely used for augmented-reality (AR) applications, it is computationally expensive. Neither of these methods handles scene motion, and assume that the viewfinder content is static. Lastly, implementing an editing framework on top of an explicit tracking algorithm is not a straightforward task, evidenced by the lack of research literature in the area.

Chapter 3

The Permutohedral Lattice

On one hand, the permutohedral lattice is a mathematical construct that has been known since the mid-twentieth century [72], describing a set of points obeying a particular arrangement. On the other, its recent success for representing and smoothing high-dimensional functions is noteworthy, especially for applications in computer graphics. The use of the lattice is central to this dissertation, and inspired much of this work. This chapter describes the permutohedral lattice, and the original work on implementing high-dimensional Gaussian filtering on top of the permutohedral lattice, conducted by Andrew Adams [10] in collaboration with the author. New contributions on the permutohedral lattice are discussed in Section 3.3 and beyond.

In the context of viewfinder editing, the permutohedral lattice will be used in the later chapters as the underlying data structure for performing high-dimensional queries. That is, the edits performed by the user will be modeled as a function over the space of high-dimensional texture descriptors, and this function will be stored in the permutohedral lattice. The lattice will then be queried for each image patch in the subsequent viewfinder frames, in order to compute the value of this function. As such, these new contributions will be crucial for accelerating this potentially time-consuming process.

The permutohedral lattice is best recognized for its relevance to the *covering problem* [73], which seeks to place unit spheres in an Euclidean space so that every point belongs to at least one unit sphere. The optimal placement of spheres that

minimizes its density is not known for arbitrary dimensions, but much literature exists on the optimal lattice covering, as lattices are easier to deal with than arbitrary pointsets. It is provably the most efficient lattice up to dimension 5 [74, 75, 76], and the most efficient known lattice up to dimension 22 [77]. As such, it is well-suited for sampling high-dimensional spaces: sampling a band-limited function effectively tiles the space with its support, a lattice with low covering density would require fewer samples to avoid aliasing [78].

3.1 Definition

The d -dimensional permutohedral lattice is denoted in mathematics as A_d^* and it is dual to the root lattice A_d . Both of these lattices lie in a d -dimensional hyperplane within \mathbb{R}^{d+1} .

Definition 3.1. *The lattices A_d and A_d^* are defined as follows:*

$$A_d = \{ \vec{x} \mid \vec{x} \in H_d \cap \mathbb{Z}^{d+1}, x_0 \equiv \dots \equiv x_d \equiv 0 \pmod{d+1} \}, \quad (3.1)$$

$$A_d^* = \{ \vec{x} \mid \vec{x} \in H_d \cap \mathbb{Z}^{d+1}, x_0 \equiv \dots \equiv x_d \equiv k \pmod{d+1} \text{ for some } k \in \mathbb{Z} \}, \quad (3.2)$$

where H_d is the hyperplane $\{ \vec{x} \in \mathbb{R}^{d+1} \mid \sum_{i=0}^d x_i = 0 \}$.

Note that this parametrization uses $d+1$ coordinates to express a d -dimensional space corresponding to the hyperplane H_d . This parametrization is standard for the permutohedral lattice and is useful for illustrating its properties. There exists a standard orthonormal rotation matrix for converting coordinates between \mathbb{R}^d and this subspace of \mathbb{R}^{d+1} [8].

In summary, A_d is the set of integer coordinates that sum to zero and have remainder 0 when divided by $d+1$. For A_d^* , the condition on the remainder is relaxed so that each component has the same remainder k for some k . It is easy to see that A_d^* is the union of $d+1$ cosets of A_d . The readers should be cautioned that the definitions given in Equations (3.1) and (3.2) have been scaled by $d+1$ to ensure that all

coordinates are integers, for ease of manipulation, in contrast to the standard definitions [77] which use fractional coordinates. For the remainder of the chapter, vertices are classified by the remainder of their components modulo $d + 1$ as *remainder- k* points. Thus, each of the $d + 1$ cosets has a consistent remainder modulo $d + 1$.

The two- and three-dimensional analogues of A_d^* are the well-known hexagonal lattice and the body-centered cubic lattice (BCC). The two-dimensional case ($d = 2$) is illustrated in Figure 3.1.



Figure 3.1: Voronoi and Delaunay tessellations of the regular grid \mathbb{Z}^2 versus A_2^* . **Left:** the Voronoi cells of \mathbb{Z}^2 and A_2^* are shown, along with lattice points. The two lattices have equal density. **Right:** the Delaunay cells induced by the Voronoi cells on the left. All cells of \mathbb{Z}^2 are squares, whereas A_2^* has triangular Delaunay cells.

The permutohedral lattice possesses a number of properties that are useful for implementing Gaussian filtering, which have been illustrated in existing literature. A few such properties are introduced below. For the mathematical background on the subject of lattices, See Conway and Sloane [77]. For a more detailed and self-contained treatise on the permutohedral lattice in the context of high-dimensional filtering, see Baek et al. [9].

Proposition 3.2. A_d^* has a uniform Delaunay cell, which is a simplex, containing the nearest remainder- k point for each $k = 0, \dots, d$.

A simplex is the general analogue to a triangle in higher dimensions.

Proof. See [8, 9, 10, 77]. The key consequence of this result is that manipulating data on the lattice is possible with a uniform algorithm that is translation-invariant. Also, the fact that the lattice yields a simplicial tessellation of the space makes the lattice useful for filtering high-dimensional signals, since the complexity of the cell grows only linearly with the dimensionality d . □

Proposition 3.3. *The Delaunay cell of A_d^* is a simplex, containing a remainder- k point for each $k = 0, \dots, d$.*

Proof. See [77]. Figure 3.2 visualizes this for A_2^* . □

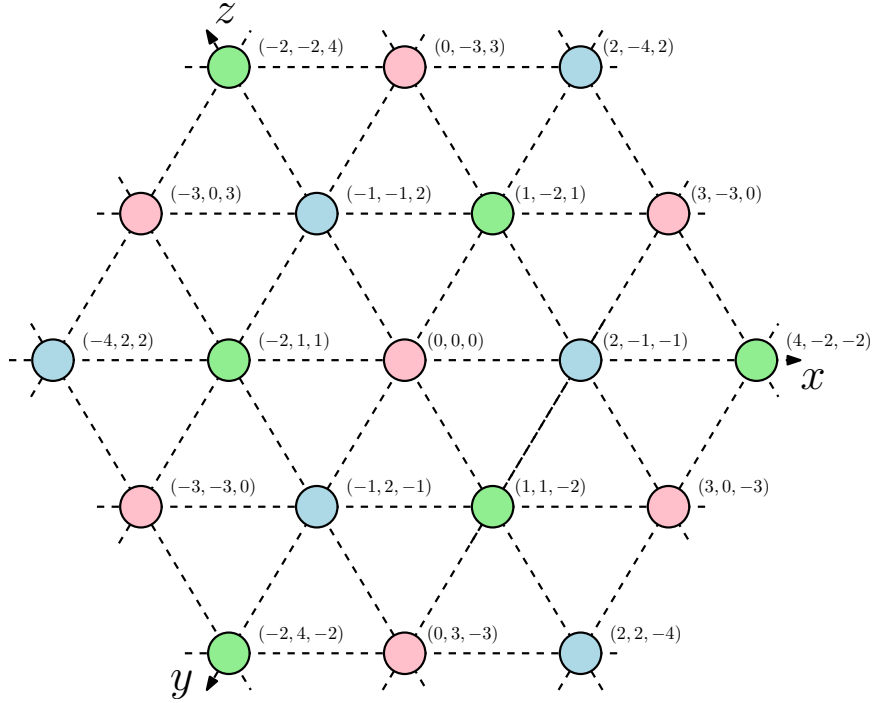


Figure 3.2: Visualization of the sublattice structure of A_2^* . Each remainder- k point has been color-coded consistently. Note that each simplex, a triangle in this case, contains one vertex of each color.

In fact, Proposition 3.3 can be strengthened further.

Proposition 3.4. *Given a point, the Delaunay cell containing it is composed of the nearest remainder- k point for each $k = 0, \dots, d$.*

Proof. See [8, 9]. This gives rise to a natural algorithm for finding the Delaunay cell: find the nearest vertex in each of the $d + 1$ cosets. Conway et al. provides an $O(d)$ algorithm for finding the nearest vertex in a designated coset [79]. □

3.2 Prior Work on Gaussian Filtering

Performing Gaussian filtering with the permutohedral lattice is a three-step process: first, the high-dimensional data is stored in the lattice; second, the lattice undergoes a filtering operation; third, the smoothed high-dimensional data is resampled from the lattice. The three steps are respectively called *splatting*, *blurring*, and *slicing*, as mentioned in Section 2.1.1.

3.2.1 Splatting

The lattice can store samples only at predetermined vertex locations. Therefore, given a high dimensional function $f : \mathbb{R}^{d_p} \rightarrow \mathbb{R}^{d_v}$ represented by a point set, the function must be resampled at the vertices in order to be stored in the lattice. The resampling kernel ought to be simple to compute, and should be isotropic if possible. Adams et al. [8] conducts the splatting in A_d^* based on barycentric interpolation. That is, given a point $\vec{p} \in H_d$, the Delaunay cell containing it is identified, and since it is a simplex, there naturally exists a set of barycentric weights with which the data can be distributed to the vertices. See Algorithm 3.1 and Figure 3.3 for the process and a visualization of the resulting resampling kernel.

Algorithm 3.1 The splatting algorithm

Compute the simplex containing the query point $\vec{p} \in H_d$. Namely, compute the nearest remainder- k point for each $k = 0, \dots, d$.

for $k \leftarrow 0, \dots, d$ **do**

Let \vec{c}_k be the remainder- k vertex in the simplex.

Compute the barycentric weight w_k of this vertex.

$f(\vec{c}_k) \leftarrow f(\vec{c}_k) + w_k \cdot \vec{v}$.

The execution of the splatting step involves finding the nearest remainder- k point in each of the $d + 1$ sublattices. Each sublattice is congruent to A_d , and finding the nearest vertex in A_d is $O(d)$, as shown in Theorem 3.5, due to Conway and Sloane [80]. Be mindful that A_d is distinct from the permutohedral lattice A_d^* . (See Definition 3.1.)

Theorem 3.5. *Finding the nearest vertex in A_d is $O(d)$.*

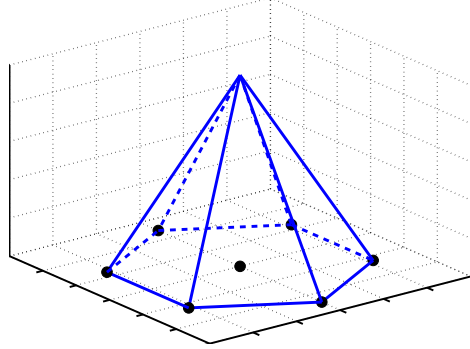


Figure 3.3: The splatting and slicing kernel for A_2^* . The kernel shows the barycentric weight assigned to the vertex at the center as a function of the query point being splatted.

Proof. Let $\vec{x} = \{x_0, \dots, x_d\} \in H_d$ be the vector to quantize. Obtain $\vec{y} = \{y_0, \dots, y_n\} \in \mathbb{Z}^{n+1}$ by rounding each component of \vec{x} to the nearest multiple of $d+1$. Let $\vec{\delta} = \vec{x} - \vec{y}$.

Now compute $\Delta = \sum_{i=0}^d \delta_i$. If $\Delta = 0$, then \vec{y} belongs to A_d and is the desired quantization. If $\Delta < 0$, we take the $(-\Delta)$ -smallest components of $\vec{\delta}$ and subtract $d+1$ from the corresponding components of \vec{y} . If $\Delta > 0$, we take the Δ -largest components of $\vec{\delta}$ and add $d+1$ from the corresponding components of \vec{y} . This process ensures that the resulting \vec{y} will belong to A_n .

In the above step, determining the $|\Delta|$ -smallest or -largest components can be accomplished in $O(d)$, by using the generalized selection algorithm [81] to find the cutoff. Hence the entire process can run in $O(d)$. \square

Figure 3.4 works through an example to illustrate how Theorem 3.5 functions. This will be important later in understanding Section 3.3, which introduces modifications to the existing algorithms.

It follows immediately that the runtime of the splatting process is $O(d^2)$.

Corollary 3.6. *The runtime of the splatting step shown in Algorithm 3.1 is $O(d^2)$.*

3.2.2 Blurring

The blurring step consists of locally propagating data within the lattice, and is analogous to implementing a Gaussian blur in a multidimensional regular grid via separable

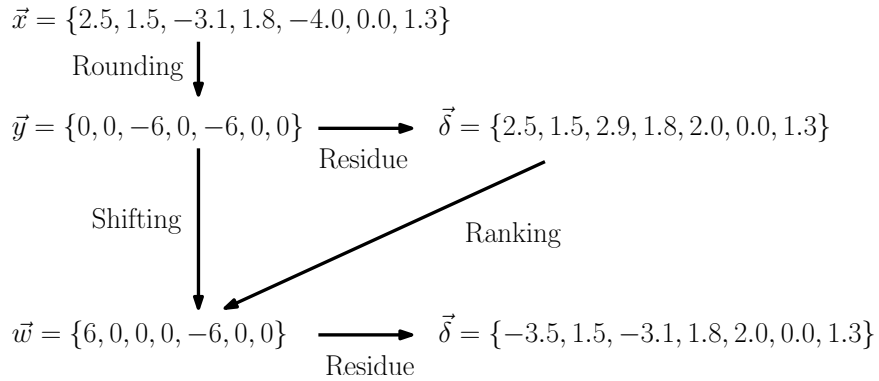


Figure 3.4: Illustration of linear-time quantization in A_d for $d = 6$. The input vector \vec{x} is rounded componentwise to the nearest integral multiple of $d + 1$, forming \vec{y} . The residue $\vec{\delta}$ is then defined as $\vec{x} - \vec{y}$. However, in order to ensure that $\vec{y} \in A_d$, its components must add up to zero. As it stands, the sum is at -12. Hence, it is necessary to increment two of the components by 6. This is done by choosing the two components with the largest corresponding residue, namely δ_0 and δ_2 . The resulting vector \vec{w} is then the vertex in A_d nearest to the query point \vec{x} .

kernels. It can be mathematically shown that performing one-dimensional Gaussian blur along each of the $d + 1$ linearly dependent axes of A_d^* does correctly implement a d -dimensional Gaussian [9]. These axes correspond to the translational vectors relating the $d + 1$ sublattices. See Figure 3.2 for the examples of these axes. Then, keeping in line with the prior work on Gaussian filtering on a regular grid [82], a discrete, finite-extent kernel is applied in each of these axes, as shown in Figure 3.5.

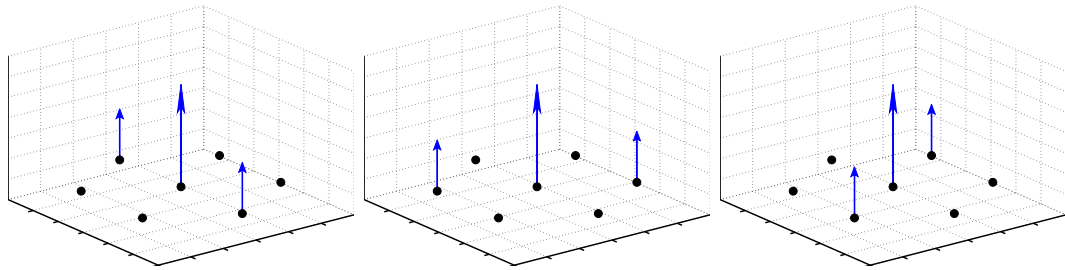


Figure 3.5: The blurring kernels for A_2^* . In general, there are $d + 1$ blurring kernels for A_d^* , each consisting of a directional blur along one of the $d + 1$ linearly dependent axes in the d -dimensional subspace.

3.2.3 Slicing

The slicing step is exactly analogous to the splatting step. The same resampling kernel is used. In fact, if splatting and slicing are guaranteed to occur in the same locations, the barycentric weights computed in the splatting step can be cached to accelerate the slicing step. As before, the runtime of the slicing step is $O(d^2)$.

Algorithm 3.2 The original slicing algorithm

```

Compute the simplex containing the query point  $\vec{p} \in H_d$ .
 $r \leftarrow \vec{0}$ .
for  $k \leftarrow 0, \dots, d$  do
    Let  $\vec{c}_k$  be the remainder- $k$  vertex in the simplex.
    Compute the barycentric weight  $w_k$  of this vertex.
     $r \leftarrow r + w_k \cdot f(\vec{c}_k)$ .
return  $r$ 

```

3.2.4 Summary

Figure 3.6 shows the resulting kernel for Gaussian filtering in A_2^* , after splatting, blurring and slicing are accounted for. In summary, the permutohedral lattice is a data structure that can perform Gaussian-like smoothing of high-dimensional data quickly, with $O(d^2)$ runtime for each splatting and slicing operation. The subsequent sections of this chapter discusses novel modifications to the lattice that enable viewfinder editing.

3.3 Fast Quantization in the Permutohedral Lattice

The slicing step in the permutohedral lattice is akin to a high-dimensional lookup operation with built-in smoothing via barycentric interpolation. A faster alternative would be a simple quantization. This section explores how to perform quantization in the permutohedral lattice quickly and efficiently, as this will be useful for viewfinder editing later. The problem of vector quantization in both structured and general lattices is a well-studied fundamental problem, dating back to the seminal work of Conway and Sloane [79, 80].

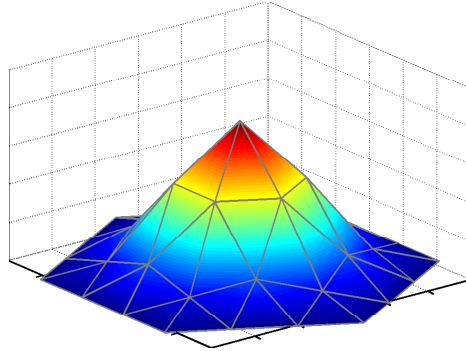


Figure 3.6: The overall kernel for Gaussian filtering in A_2^* . It is the convolution of the splatting kernel, the $d + 1$ blurring kernels, and the slicing kernel, and approximates a truncated Gaussian kernel.

Given the d -dimensional lattice A_d^* , the naïve approach for quantization would be to invoke Theorem 3.5 for each of the $d + 1$ sublattices. Once the nearest remainder- k vertex is found for each $k = 0, \dots, d$, the distance to the query point can be computed, and the vertex that minimizes this distance can be chosen. However, the asymptotic runtime of this approach is still $O(d^2)$ as noted by Conway and Sloane [80], since simply enumerating the coordinates of the nearest remainder- k vertices would require a quadratic-time procedure.

This runtime can be improved, however. Figure 3.7 provides an intuition as to how subquadratic quantization may be possible: once the $O(d)$ algorithm is executed to find the nearest remainder- k point for a particular k , it restricts the choice of possible nearest vertex in all other sublattices. Theorem 3.7 shows that a subquadratic time is indeed possible:

Theorem 3.7. *Vector quantization in A_d^* is $O(d \log d)$.*

Proof. Instead of running the generalized selection algorithm in $O(d)$ for each coset, in the proof of Theorem 3.5, we can sort the components of $\vec{\delta}$ once in $O(d \log d)$ instead and reuse the resulting ranking of the components. Then, to find the nearest vertex in the other sublattices (translated copies of A_d), one can simply translate $\vec{\delta}$ by the same amount, while keeping it sorted in $O(1)$. The constant time for update is possible because the translation vector does not affect the sorted order, except for a single element: the translation will always be a permutation of $(1, 1, \dots, 1, -d)$. The

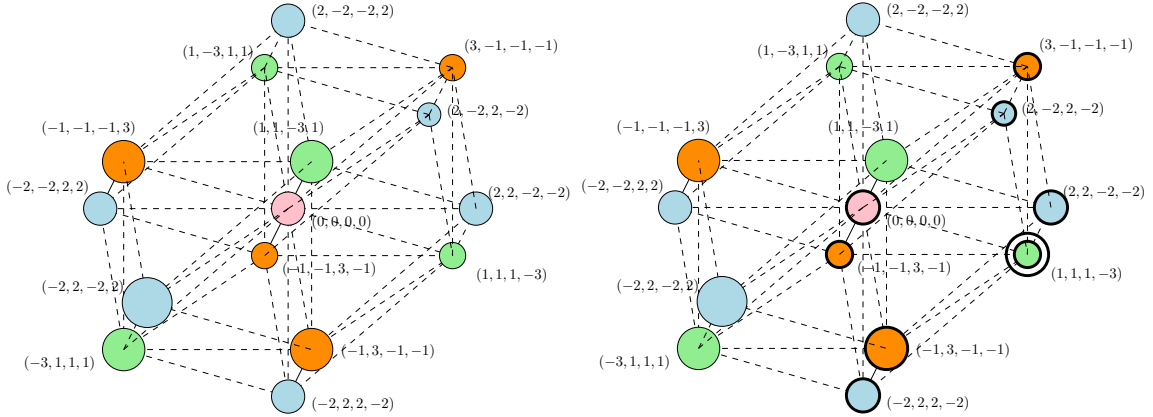


Figure 3.7: Illustration of subquadratic quantization algorithm on A_3^* . **Left:** all vertices sharing a Delaunay cell with a particular vertex, namely $(0, 0, 0, 0) \in A_3^*$ are shown. **Right:** Once Theorem 3.5 is invoked to locate the nearest remainder-1 point, say $(1, 1, 1, -3)$, it narrows down the possible candidates for the nearest remainder- k points for $k > 1$. All possible vertices are shown with bolder outline. This indicates that a linear-time algorithm for finding the nearest vertex in the remaining sublattice may be unnecessary.

computation of the distance to the query point can be similarly amortized to be $O(1)$ per vertex, resulting on $O(d \log d)$ total runtime. \square

One can improve the algorithm further to a linear-time process, a key result in this dissertation.

Theorem 3.8. *Vector quantization in A_d^* is $\Theta(d)$.*

The major insight is that a partial sorting of $\{\delta_0, \dots, \delta_d\}$ is sufficient.

We begin with $\vec{x} = \{x_0, x_1, \dots, x_d\} \in H_d$. Find the nearest lattice point in A_d per Theorem 3.5 in $O(d)$, and subtract it from \vec{x} to obtain $\vec{\delta}$. We will now quantize $\vec{\delta}$. The quantization of \vec{x} can then be obtained by undoing the subtraction.

As before, denote by v^k the nearest remainder- k lattice point to $\vec{\delta}$. The goal is then to find v^k over $k = 0, \dots, d$ that minimizes the distance to δ . By construction, $v^0 = \{0, \dots, 0\}$. From [8, 77], we know that v^k is a permutation of the following:

$$\left\{ \underbrace{k, \dots, k}_{d+1-k}, \underbrace{d+1-k, \dots, d+1-k}_k \right\},$$

where the components of value k are stored in indices corresponding $(d + 1) - k$ largest components of $\vec{\delta}$. Let i_0, i_1, \dots, i_d be the indices sorted by their respective components, e.g. $\delta_{i_a} \leq \delta_{i_b}$ whenever $a \leq b$. For brevity, let $I \downarrow = \{i_0, \dots, i_{k-1}\}$ and $I \uparrow = \{i_k, \dots, i_d\}$.

The task is then equivalent to solving the following:

$$\begin{aligned}
\operatorname{argmin}_k \|\vec{\delta} - v^k\|^2 &= \operatorname{argmin}_k \sum_{i=0}^d (\delta_i - v_i^k)^2 \\
&= \operatorname{argmin}_k \sum_{i \in I \uparrow} (\delta_i - k)^2 + \sum_{i \in I \downarrow} (\delta_i - k + (d + 1))^2 \\
&= \operatorname{argmin}_k \sum_{i=0}^n (\delta_i - k)^2 + \sum_{i \in I \downarrow} [(d + 1)^2 - 2(\delta_i - k)(d + 1)] \\
&= \operatorname{argmin}_k \left[\left(\sum_{i=0}^d \delta_i^2 \right) - \left(2k \sum_{i=0}^d \delta_i \right) + k^2(d + 1) \right] \\
&\quad + \left[(d + 1)^2 k - 2k^2(d + 1) + \left(2(d + 1) \sum_{i \in I \downarrow} \delta_i \right) \right] \\
&= \operatorname{argmin}_k (d + 1)^2 k - k^2(d + 1) + \left(2(d + 1) \sum_{i \in I \downarrow} \delta_i \right) \\
&\quad \text{by dropping the first summation (constant)} \\
&\quad \text{and the second summation (zero)} \\
&= \operatorname{argmin}_k (d + 1)k - k^2 + \left(2 \sum_{i \in I \downarrow} \delta_i \right).
\end{aligned}$$

For brevity, we shall denote by $f(k)$ the objective function in the last line.

Lemma 3.9. *Suppose λ minimizes $f(k)$, and $\lambda \in \{1, \dots, n - 1\}$. Then,*

$$\delta_{i_\lambda} - \delta_{i_{\lambda-1}} \geq 1.$$

Proof of Lemma. By optimality, it must be that

$$f(\lambda) \leq f(\lambda - 1) \text{ and } f(\lambda) \leq f(\lambda + 1).$$

The restriction on λ ensures that all terms above are well-defined. Substituting in the definition of $f(\cdot)$, we obtain,

$$\begin{aligned}
f(\lambda) &\leq f(\lambda - 1) \\
\implies \left[(d+1)\lambda - \lambda^2 + 2 \sum_{i \in I \downarrow} \delta_i \right] &\leq \left[(d+1)(\lambda - 1) - (\lambda - 1)^2 + 2 \sum_{i \in I \downarrow \setminus \{i_{\lambda-1}\}} \delta_i \right] \\
&\text{where } I \downarrow := \{i_0, i_1, \dots, i_{\lambda-1}\}, \\
\implies 2 \left(\sum_{i \in I \downarrow} \delta_i - \sum_{i \in I \downarrow \setminus \{i_{\lambda-1}\}} \delta_i \right) &\leq (d+1)(\lambda - 1) - (\lambda - 1)^2 - ((d+1)\lambda - \lambda^2) \\
\implies 2\delta_{i_{\lambda-1}} &\leq 2(\lambda - 1) - d \\
\implies \delta_{i_{\lambda-1}} &\leq (\lambda - 1) - \frac{d}{2}
\end{aligned}$$

Similarly, we obtain $\delta_{i_\lambda} \geq \lambda - \frac{d}{2}$ from $f(\lambda) \leq f(\lambda + 1)$.

Combining the two transitively produces

$$\delta_{i_{\lambda-1}} \leq (\lambda - 1) - \frac{d}{2} \leq \lambda - \frac{d}{2} \leq \delta_{i_\lambda}.$$

The lemma follows immediately from the above inequality. \square

Lemma 3.9 gives us an important precondition on optimality of $\lambda \in \{1, \dots, d-1\}$. While this does leave out the possibility of $\lambda = 0$ or $\lambda = d$, these can be checked in $O(d)$ respectively.

Proof of Theorem 3.8. The range of δ_i is contained in $[-d-1, d+1]$. Hence, we can divide this interval into $2(d+1)$ bins of unit length each, and place $\{\delta_0, \dots, \delta_d\}$ into the bins, forming a histogram. By the previous lemma, $\delta_{i_{\lambda-1}}$ and δ_{i_λ} cannot be in the same bin. In fact, the former must be the maximum of a bin, and the latter must be the minimum of the next nonempty bin. Equivalently, k must equal the number of items in the first m bins for some m , so that $\delta_{i_{k-1}}$ (i.e. the k -th smallest component) will be the maximum of the m -th bin. In other words, we can iterate through $m \in \{0, 1, \dots, 2d+1\}$ and test the appropriate $f(k)$. Incidentally,

this procedure ends up evaluating $f(0)$ and $f(d)$.

The running time of this procedure is $O(d)$: it is true that the computation of $f(\cdot)$ can be invoked up to $O(d)$ times, but the only term potentially expensive to compute is the expression $2 \sum_{i \in I_{\downarrow}} \delta_i$, which is the sum of all items in the preceding bins. This expression can be evaluated in amortized $O(1)$ time by keeping a running sum, as shown in Algorithm 3.3.

Algorithm 3.3 Algorithm for quantizing a vector $\vec{x} \in \mathbb{R}^{d+1}$ in A_d^*

Compute the nearest lattice point $\vec{w} \in A_d$ per Theorem 3.5.

Compute the differential $\vec{\delta} := \vec{x} - \vec{w}$.

Initialize arrays $B_{count}[\cdot] = 0$, $B_{sum}[\cdot] = 0$.

for $i \leftarrow 0, d$ **do**

$j \leftarrow \text{floor}(\delta_i + (d + 1))$.

$B_{count}[j] \leftarrow B_{count}[j] + 1$.

$B_{sum}[j] \leftarrow B_{sum}[j] + \delta_i$.

Initialize $k \leftarrow 0$, $S_{sum} \leftarrow 0$.

for $m \leftarrow 0, 2d + 1$ **do**

Compute $f(k) = k - \frac{k^2}{d+1} + 2 \cdot S_{sum}$.

$k \leftarrow k + B_{count}[m]$.

$S_{sum} \leftarrow S_{sum} + B_{sum}[m]$.

$\lambda \leftarrow \text{argmin}_k f(k)$ among all computed $f(\cdot)$.

Compute $v^{\vec{\lambda}}$, the nearest lattice point from the λ -th coset of $A_d \subset A_d^*$

return $v^{\vec{\lambda}} + \vec{w}$.

In terms of the pseudocode, the major insight is that computing the sum of the items in the bins does not require sorting the contents of each bin, *because Lemma 3.9 ensures that a partial sum of a bin content is never necessary*. For the algorithm below, it is easy to verify that each step outside the loops is $O(d)$ and each step inside the loops is $O(1)$, yielding an asymptotic runtime of $O(d)$ overall. Because each of the $d + 1$ components must be accessed at least once, a sublinear algorithm is impossible, leading to the conclusion that vector quantization in A_d^* is $\Theta(d)$, as desired. \square

Figure 3.8 visualizes the inequality in Lemma 3.9 for A_2^* . As the figure illustrates,

the linear time algorithm relies on the fact that checking these inequality can be amortized to $O(1)$, and by checking these inequalities, one obviates the need to perform a full sort of the coordinates of the query point.

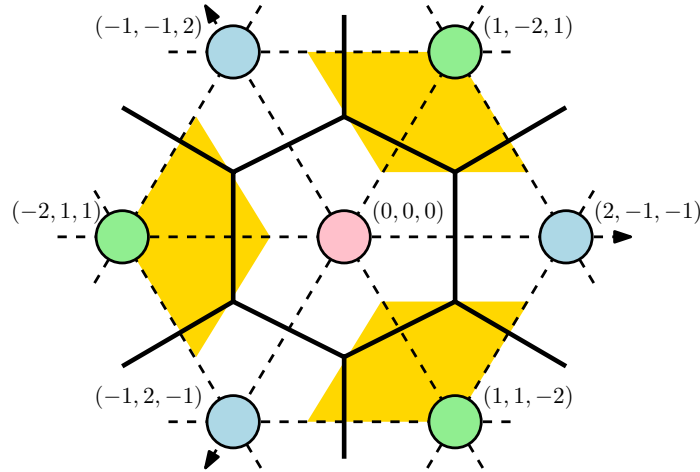


Figure 3.8: One of the inequalities given in Lemma 3.9 is visualized as yellow shaded area. The Voronoi cell of each vertex is also shown, and is a hexagon in A_2^* . Note that failing to satisfy the inequality immediately rules out remainder-1 vertices (marked in green) from being the nearest vertex.

3.3.1 Further Improvement

While Theorem 3.8 shows that the quantization algorithm in Algorithm 3.3 cannot be further improved asymptotically, it is indeed possible to optimize it for faster execution. Recall that Algorithm 3.3 invokes Theorem 3.5 to find the nearest remainder-0 vertex in A_d and then again to find the nearest vertex from the closest sublattice, making use of the generalized select algorithm each time. However, branching operations can be expensive, especially on mobile CPUs that may not be equipped with sophisticated branch-predicting hardware. By modifying the algorithm appropriately, one can achieve linear time quantization without having to rely so heavily on conditional statements. Algorithm 3.4 describes the modified algorithm. A full C++ implementation is available in Appendix B, which can be dropped into the original lattice code [8].

Algorithm 3.4 A faster algorithm for quantizing a vector $\vec{x} \in \mathbb{R}^{d+1}$ in A_d^*

Round each coordinate of \vec{x} to the nearest multiple of $d + 1$. Assign this to \vec{y} .

$E \leftarrow (\sum_i y_i)/(d + 1)$. This may be nonzero.

Compute the differential $\vec{\delta} := \vec{x} - \vec{y}$.

Initialize arrays $B_{count}[\cdot] = 0$, $B_{sum}[\cdot] = 0$.

for $i \leftarrow 0, d$ **do**

$j \leftarrow \text{floor}(\delta_i + (d + 1))$.

$B_{count}[j] \leftarrow B_{count}[j] + 1$.

$B_{sum}[j] \leftarrow B_{sum}[j] + \delta_i$.

Initialize $k \leftarrow 0$, $S_{sum} \leftarrow 0$.

for $m \leftarrow 0, 2d + 1$ **do**

 Compute $f(k) = (k - E) - \frac{(k - E)^2}{d + 1} + 2 \cdot S_{sum}$.

$k \leftarrow k + B_{count}[m]$.

$S_{sum} \leftarrow S_{sum} + B_{sum}[m]$.

$\lambda \leftarrow \text{argmin}_k f(k)$ among all computed $f(\cdot)$.

$m' \leftarrow$ the value of m that corresponded to this λ in the above loop.

$\lambda \leftarrow \text{mod}(\lambda - E, d + 1)$.

for $i \leftarrow 0, d$ **do**

$\vec{v}_i \leftarrow y_i + \lambda + (d + 1) \cdot \text{sign}(E - \lambda)$.

if $\text{floor}(\delta_i + (d + 1)) \leq m'$ **then**

$\vec{v}_i \leftarrow \vec{v}_i - (d + 1)$.

return \vec{v} .

The key departure in the modified algorithm from the original linear-time algorithm is that the “anchor” vertex in A_d (denoted by \vec{w} in Algorithm 3.3) is never explicitly computed. This vertex is originally obtained by first rounding the components of the query point \vec{x} to the nearest integral multiples of $d + 1$ (denoted by \vec{y} in Algorithm 3.4) and then adjusting the components by $d + 1$ or $-(d + 1)$ until their sum becomes zero, as described in the proof of Theorem 3.5. Referring to Figure 3.4, one observes that not adjusting the components simply will cycle the residues within the interval $[-(d + 1), (d + 1)]$ as shown in Figure 3.9, and does little to change the distance between the consecutive residues, which are of importance to Lemma 3.9. Therefore, with a bit of tweak in the bookkeeping of coefficients, one can obtain the same quantization vector, as shown in Algorithm 3.4.

In practice, the speed-up that results from eliminating unnecessary branching is

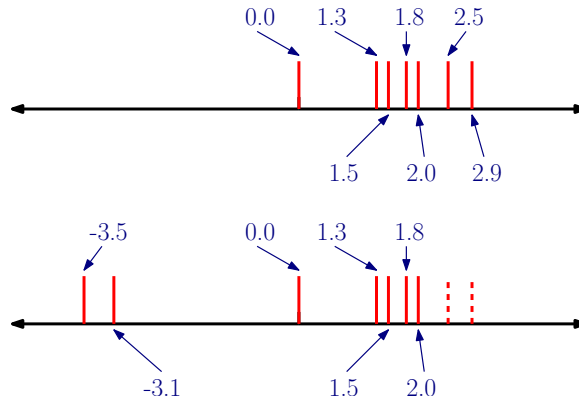


Figure 3.9: Illustration of the effect of avoiding explicitly finding the anchor point in quantization. The same input vector is used as in Figure 3.4. **Top:** The distribution of the residues $\delta_0, \dots, \delta_d$ obtained from $\vec{x} - \vec{y}$. **Bottom:** The distribution of the residues once \vec{y} is corrected to have a zero-sum, yielding the anchor point \vec{w} . Note that the residues at the fringe were shifted to the other end. Ultimately, the set of distances between consecutive residues (when sorted) remains stable, to the extent that applying Lemma 3.9 is straightforward.

significant. Table 3.1 displays the runtime of lattice lookup in the 16-dimensional permutohedral lattice, for the prior work of Adams et al. [8] using barycentric interpolation, the linear-time algorithm described in Algorithm 3.3, and the further optimized code described in Algorithm 3.4. The linear-time algorithm is about three times faster than the prior work, whereas the further optimization reduces the runtime by an additional 19%.

Method	Runtime
Barycentric interpolation	13.24 ms
Linear-time quantization	4.41 ms
Improved linear-time quantization	3.71 ms

Table 3.1: Runtime comparison for various lattice lookup methods. The runtime was tested as a part of a complete viewfinder-editing application that runs on the ARM cores of a NVIDIA Tegra4 tablet. The total time taken to perform lattice lookup for 16-dimensional descriptors in each viewfinder frame at VGA resolution is reported.

3.4 Adaptations for Streaming Texture Retrieval

Looking ahead, the viewfinder editing system in Chapter 4 employs the permutohedral lattice as the underlying data structure for texture lookup. In particular, the patches that the user wishes to edit will be stored in the permutohedral lattice, and the patches in each viewfinder frame will be looked up in the lattice to see if they had been previously stored. This kind of usage departs in two key ways from the general image-processing tasks in which the permutohedral lattice had previously been used.

The first is that the input data is a stream, potentially unbounded in length. To address this, the permutohedral lattice can be extended to handle streaming data in order to model a time-varying or time-weighted high-dimensional function. A potential example with the bilateral grid is given by Paris et al. [83] for the application of video filtering, in which the homogeneous weight associated with the value in every vertex is decayed by a constant factor α at each time step, and a new set of values are additively splatted. However, this per-vertex update can be expensive to perform at every frame. As will be shown in Chapter 4, the viewfinder editing framework explicitly avoids doing $O(1)$ work per vertex per frame.

The second departure from previous uses of the permutohedral lattice is that there is an imbalance between splatting and slicing needs: splatting is a rare event that occurs only when and where the user generates a command, whereas slicing must be fast enough to process viewfinder frames in real time. Therefore, slicing step can be replaced by a simple vector quantization shown in Section 3.3, rather than barycentric interpolation. It is true that replacing the slicing step with a simple quantization makes the overall kernel less isotropic (farther away from a true Gaussian), but in practice the high-dimensional filtering works well enough, as shown in the results in Chapter 7.

3.4.1 Importance Measure

In the modified version of the permutohedral lattice, each vertex \vec{p} has an associated importance measure $M(\vec{p})$ ¹. This importance measure is lazily updated upon access. To allow computing the correct factor for decay, the vertex \vec{p} also records the timestamp $t(\vec{p})$ (typically the associated frame count) of the last update. In summary, when the vertex happens to be accessed at time t_{now} , the vertex data updates itself as follows:

$$\begin{aligned} M(\vec{p}) &\leftarrow M(\vec{p}) \cdot \alpha^{t_{now}-t(\vec{p})}, \\ t(\vec{p}) &\leftarrow t_{now}. \end{aligned}$$

Whenever the vertex receives new data via splatting or is accessed via slicing, the importance measure is augmented by the corresponding barycentric weight. Therefore, the importance measure is proportional to the frequency of read or write access, and decays over time. Note that the aggregate total of the importance measure over the entire lattice can be easily tracked: each splatting operation will increment it by 1, and each slicing operation will increment it by 1 or less, depending on whether the vertices of the enclosing simplex exist. Between every iteration, it decays by α .

3.4.2 Deleting Vertices

In order for the permutohedral lattice to function with a finite, bounded amount of memory, it must be able to delete vertices once the load factor on the underlying hash table is sufficiently large. The time-decaying importance measure provides a useful metric for this purpose: given its use in the viewfinder editing framework, any image patches the user wishes to edit will have its importance boosted via splatting. The importance decays slowly, unless the image patches remain within the field of view of the camera: in this case, the image patch will be found during the slicing step, and will keep its importance high. Hence, deleting nodes with the lowest importance is a

¹Note that this measure is unrelated to the possible homogeneous coordinate in the associated value of the vertex.

well-grounded approach.

Unfortunately, searching the lattice for the nodes with the lowest importance is impractically expensive. As an approximation, the following scheme is used: when splatting a vertex that induces a hash collision, the lowest importance is tracked. If the number of hash collision exceeds a threshold (between 6 and 12 in the implementation), the entry corresponding to the lowest importance is removed, and the new vertex is inserted in its place.

3.5 Summary

The permutohedral lattice is a mathematical structure that has been successfully adapted to the task of high-dimensional Gaussian filtering, in collaboration with Andrew Adams and others. This chapter introduced its definition and its use in high-dimensional filtering, and unveiled a fast quantization scheme for high-dimensional lookup as a new contribution, among others. Theorem 3.8 and Algorithm 3.4 embody the key results. The resulting quantization scheme is considerably faster than the original work that relies on barycentric interpolation, and will be used in viewfinder editing in Chapter 4.

Chapter 4

Viewfinder Editing

Viewfinder editing is a new concept proposed in this dissertation, but it is perhaps a natural extension to the existing state of art in edit propagation. Viewfinder editing allows the user to specify local and global edits directly on the viewfinder of a camera. Hence, editing photographs becomes an online process with immediate feedback, as opposed to the traditional workflow in which one captures the photos first and then downloads them to one's desktop computer to process them in a photo-editing software that may deploy the traditional edit propagation algorithms.

The obvious advantage of this approach is that the user can immediately see the result of his edits and obtain useful feedback, essentially making the experience WYSIWYG. This helps the user reconsider his composition, or may even factor into the decision to take the particular photograph in the first place. It further accelerates the content-publishing pipeline, as the user can immediately upload or share his creations at the time of shutter press. Furthermore, as will be explored in Chapter 5, new kinds of camera control algorithms are enabled by the WYSIWYG property of the proposed framework.

The main challenges of viewfinder editing lies in designing a robust algorithm for propagating edits spatiotemporally that can operate within the limits of the computational capability and programmability of mobile devices. The subsequent sections describe the design and implementation of such an algorithm, using the permutohedral lattice discussed in Chapter 3 as a foundation.

4.1 Affinity-Based Edit Propagation

Image editing on a viewfinder stream of a hand-held camera must accommodate temporally persistent selection of objects through sparse user input. While explicit tracking of keypoints through SLAM (Simultaneous Localization and Mapping) or other variants [84] is possible, such an approach remains computationally expensive and does not account for scene motion.

The viewfinder editing framework forgoes explicit tracking, and relies instead on affinity-based edit propagation [60, 64], in which edits are modeled as functions residing in the space of local patch descriptors. Consider a d -dimensional descriptor function D that operates on an $m \times m$ patch:

$$D : \mathbb{R}^{m^2} \rightarrow \mathbb{R}^d. \quad (4.1)$$

Note that spatial information is discarded in the above descriptor, in departure from the existing work: some spatial locality will be regained later through image-space operations. Then, the edits are defined as follows:

$$S_i : \mathbb{R}^d \rightarrow [-1, 1], \quad i = 1, 2, \dots \quad (4.2)$$

where each of S_1, S_2, \dots corresponds to a particular type of edit, such as tone, color, saturation or blurriness. The sign of the function corresponds to the direction of the edit—for instance, for tonal edits, positive edits correspond to brightening whereas negative edits correspond to darkening; for focus edits, positive edits correspond to sharpening whereas negative edits correspond to blurring, and so on. Finally, $S_0 : \mathbb{R}^d \rightarrow [0, 1]$ is reserved for representing the soft selection mask. The above formulation implicitly assumes that the local context around each pixel is sufficient to describe the set of regions that the user wants to associate with the given edit; in other words, pixels with similar neighborhoods should be subjected to similar edits. To compute the edit mask M_i for an entire viewfinder frame, one iterates through the $m \times m$ -sized patches sampled from the frame and computes the composition $(S_i \circ D)$

for each $i \geq 0$ to obtain the edit masks:

$$M_i : \mathbb{Z}^2 \rightarrow [-1, 1], \text{ where}$$

$$M_i : (x, y) \mapsto S_i \circ D(\{\text{the } m \times m \text{ patch centered at } (x, y)\}). \quad (4.3)$$

Once the edit maps are computed for each type of edit, they can be applied to the viewfinder content, which is displayed to the user. Note that there is no explicit continuity constraint on the mask, but in practice the mask will eventually undergo spatial smoothing as described in Section 4.4.3.

There are three main hurdles to accomplishing the above task on the viewfinder stream. First, a user interface must be designed so that the function S_i can be implicitly constructed by the user’s interaction with the viewfinder. Second, the composition of the vector-valued function $\vec{S} = (S_0, S_1, S_2, \dots)$ and D , namely $\vec{M} = (M_0, M_1, M_2, \dots)$ must be computed quickly. Third, an adequate descriptor D must be designed, which is sufficiently discriminative yet robust and fast to compute. These three subproblems are presented in the following sections in the provided order, as the later subproblems are informed by the earlier ones.

4.2 Representing and Specifying Edits

Existing affinity-based methods attempt to globally optimize or interpolate \vec{S} based on the user-provided samples. The cost of global optimization or interpolation is often mitigated by preprocessing the dataset to learn the topology of the patch descriptors or training a classifier. While this approach works well for offline processing of video or image sequences, it is impractical for the online processing the viewfinder frames in a streaming fashion.

In order to address this issue, \vec{S} is represented by the permutohedral lattice described in Chapter 3. The lattice is suitable since it can store high-dimensional vector-valued function (e.g., \vec{S}) with $O(1)$ cost for incremental update. Note that the lattice internally performs resampling in the patch descriptor space upon splatting, which serves to locally propagate the data.

Because we forgo an explicit optimization or interpolation unlike previous work, edits do not propagate as aggressively, but this issue is mitigated in three ways: first, we apply edge-aware smoothing on S_i with respect to the scene image whenever a viewfinder frame is produced. Second, because the user receives feedback interactively as the strokes are made, it is easy and intuitive to control propagation—the user essentially interactively paints S_i . Third, once the user captures a stack, we rely on existing edit-propagation algorithms in the literature for high-quality offline processing.

Instead of initializing the lattice with all patches present in a given image, we take a streaming approach: as the user strokes over the screen and selects patches, we locate only nodes corresponding to these patches and update their values. Note that unselected patches are never written into the lattice; if a patch lookup fails at any point, a default value is assumed for \vec{S} .

As customary in Gaussian filtering, we use 2D homogeneous coordinates to represent each of S_0, S_1, \dots in the permutohedral lattice. The actual value of S_0, S_1, \dots is obtained by dehomogenizing the 2D vector. We will denote the homogeneous form as $\tilde{S}_i : \mathbb{R}^d \rightarrow \mathbb{R}^2$ for each i . The homogeneous coordinates are used to express the relative weight of the edit.

Edits are specified in three phases, as illustrated in Figure 4.1: first, the user strokes over the region of interest, and confirms the selection by tapping on the selected area. Second, the user is shown a widget listing the various types of edits supported, and the user taps on his choice. Third, the user horizontally swipes left or right, in order to specify how much the edited value should be decreased or increased. All of the phases are interactive; as the user moves a finger on the screen, the updated edits are reflected on the viewfinder.

Selection. While the user stroke is being registered, image patches whose centers are within a small fixed distance from the touch event are converted to descriptors and are looked up from the lattice. New nodes are created if they are not found. We

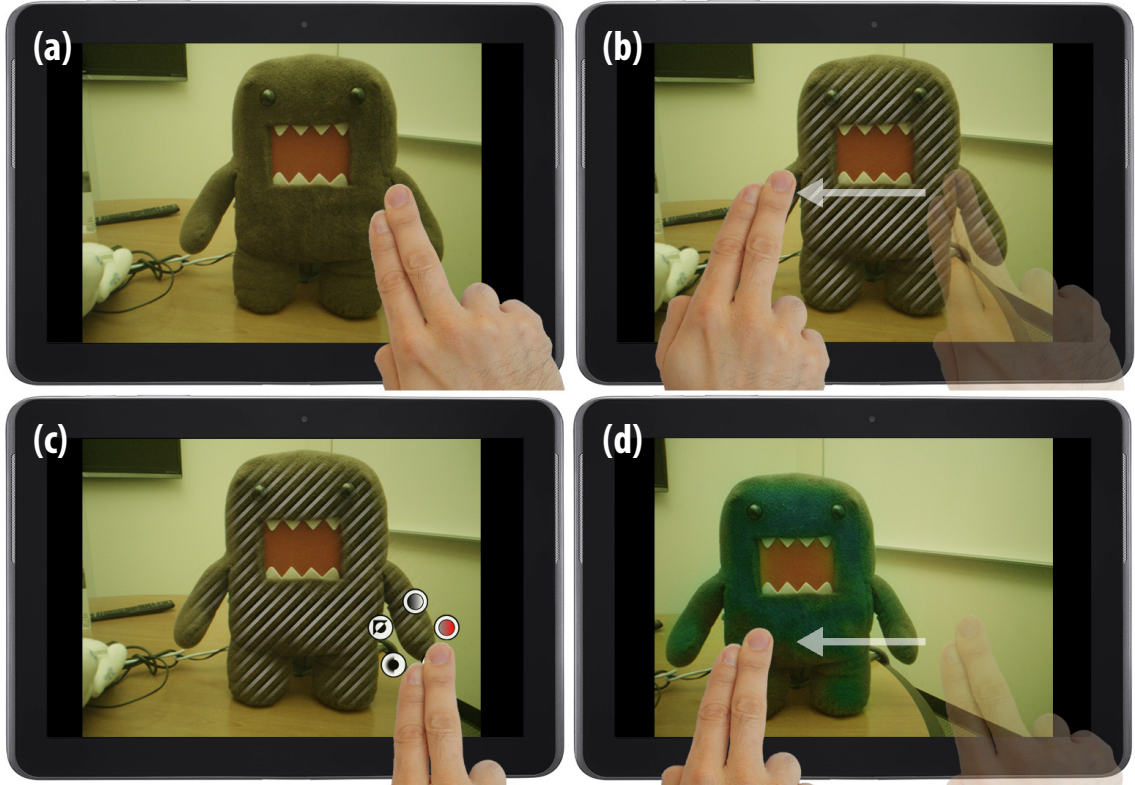


Figure 4.1: Interface for viewfinder editing. **(a)**: The user begins by stroking over the region of interest. **(b)**: As the user swipes his finger, the selection updates on the screen. In this example, the user has selected the doll via a single stroke. **(c)**: The user confirms the selection by tapping within the selected region, which invokes a UI widget offering various edit operations the user can choose from. The user chooses to edit the hue of the selected region. **(d)**: As the user swipes his finger horizontally to indicate the sign and the magnitude of the edit, the viewfinder updates. The images are actual screen grabs from an editing session on a tablet. The tablet border and the hand are simulated images.

increment the value of S_0 for these nodes to signify that they are now selected

$$\tilde{S}_0(\vec{p}) := \tilde{S}_0(\vec{p}) + \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \quad (4.4)$$

In practice, a vertex lookup (via splatting) in the permutohedral lattice will return several nearby vertices for barycentric interpolation. The above equation is applied to

all these, scaled by the appropriate barycentric coefficient for each vertex. If the user passes over the same texture \vec{p} multiple times in a single stroke, the dehomogenized value $\vec{S}_0(\vec{p})$ will still be 1, but the weight of the selection will be larger, so \vec{p} will be more influential in the interpolation performed in the lattice.

The cost of specifying selection is $O(1)$ for each event, independent of viewfinder dimensions and the edit history.

Editing. If the user is in the act of specifying an edit $k \in [-1, 1]$ of type j , then for each selected descriptor \vec{p} , we adjust $\tilde{S}_j(\vec{p})$ before applying it to the image.

$$\tilde{S}_j(\vec{p}) := \tilde{S}_j(\vec{p}) + k \cdot \begin{pmatrix} S_0(\vec{p}) \\ 1 \end{pmatrix}. \quad (4.5)$$

As shown in Equation (4.5), the extent of the edit is further scaled by the soft selection mask S_0 .

Note that this adjustment is not yet written into the lattice. Therefore, the cost of visualizing each viewfinder frame grows linearly with the viewfinder dimensions, and is independent of the number of nodes in the lattice. Once the user finalizes the edit, we can fold it into the lattice by applying Equation (4.5) to every selected patch \vec{p} in the lattice, and reset $\tilde{S}_0(\vec{p})$ to zero. Hence, the cost of finalizing an edit is proportional to the size of the lattice, and independent of the viewfinder dimensions. While this step is slightly more expensive, it is only performed when the user signals the end of a discrete edit.

4.3 Descriptor Design

Because the robustness of the appearance-based edit propagation scheme depends strongly on the texture descriptor, the descriptor must be carefully selected. Figure 4.2 shows a number of well-known texture descriptors in the graphics and vision literature. On one hand, the descriptor must be sufficiently discriminative. On the other hand, increasing the dimensionality of the descriptor quickly overwhelms the

edit propagation algorithm or can lead to overfitting [63]. Descriptors at several dimensionalities were explored. The smallest descriptor tested was a simple color value at each pixel ($d = 3$), and was quickly deemed insufficient. Generally, anything under $d = 8$ was not sufficiently discriminative. At the same time, understanding of the use of the permutohedral lattice and its runtime discouraged dimensionalities at $d = 20$ or higher. Between these thresholds, several descriptors at either $d = 8$ and $d = 16$ were implemented and evaluated, as described below. Note that dimensionalities of powers of 2 are preferred for ease of 4-wide vectorization.

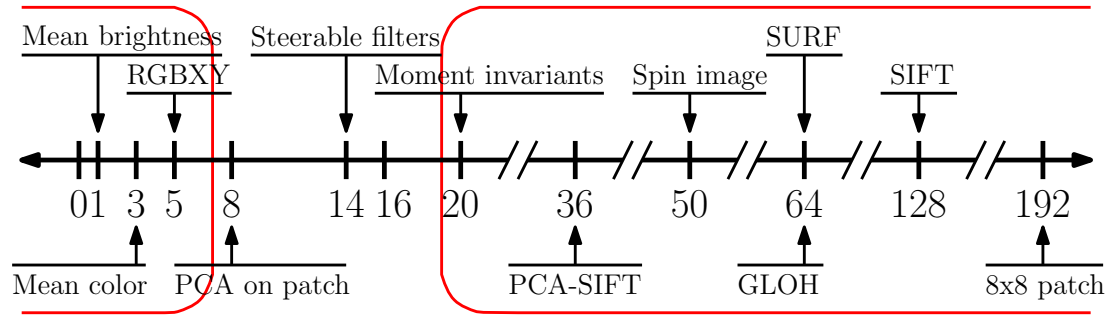


Figure 4.2: A summary of popular descriptors in the computer graphics and vision literature, sorted in the order of increasing complexity from left to right. The number on the scale indicates the dimensionality of the descriptors. On one hand, descriptors that are less than 8 dimensions (marked by a red outline on the left) were not sufficiently discriminative. On the other hand, descriptors that are 20 dimensions or higher (marked by a similar red outline on the right) were too expensive. As described in Section 4.3, various descriptors of dimensionalities between 8 and 16, inclusive, were tested. Eventually, a descriptor of 16 dimensions was chosen.

All the descriptors tested contain three components corresponding to the mean pixel value in the underlying 3-dimensional color space (RGB, YUV, etc.), and the remaining components are values computed from the luminance channel. Each descriptor required manual tuning of the relative strength of the individual channels. In the implementations, however, equal weights were assigned to the auxiliary channels—the ones other than the mean pixel value. The weights for the mean chrominance channel were set higher by a factor of 4 or 8 than that of the mean luminance channel, as the numerical variation in chrominance channels is typically smaller.

Pixel-based Descriptors

Using the pixel values in an image patch directly as a descriptor has been highly successful in denoising tasks [85]. However, as the dimensionality is too large for real-time applications, even at 8-by-8 grayscale patches ($d_p = 64$), dimensionality-reduction techniques such as principal component analysis (PCA) [86] are essential for other tasks. Figure 4.3 shows the first 12 principal components from a set of 30 million image patches sampled from viewfinder sequences. It was observed that the principal components resemble the zeroth-, first- and second-order gradients on the image patch, in addition to some components corresponding to color information. Based on this information, the following 8-dimensional descriptor was developed and tuned: mean pixel values (3 components), the first-order derivatives (2 components) of intensity, the second-order derivatives (3 components) of intensity. This particular descriptor with patch size of 8×8 was used for generating all figures from [11].

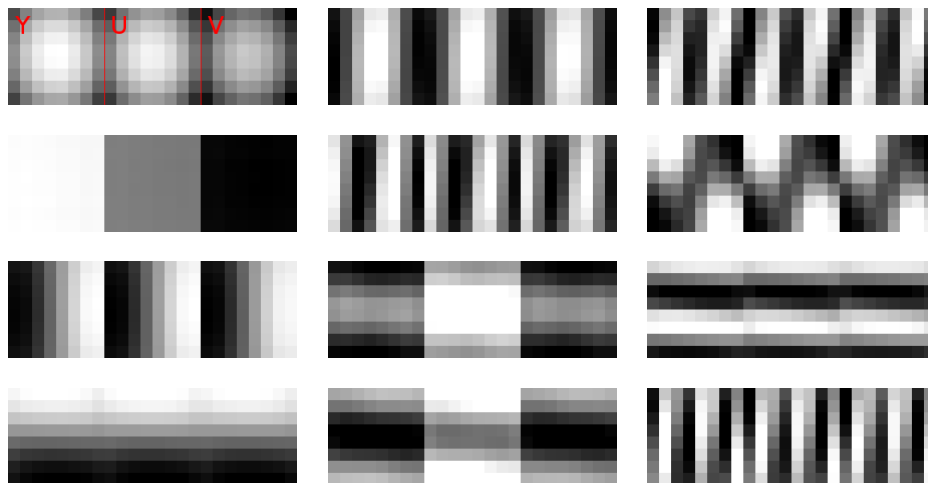


Figure 4.3: The first 12 principal components of 8-by-8 image patches in the YUV color space. The top-left image corresponds to the first component, and the rest of the images are numbered in order, traveling down in a column first then across rows. Each image visualizes the coefficient of each of the 8-by-8-by-3 pixels: the first third corresponding to the 8-by-8 patch in the luminance channel, and the next two to the two chrominance channels. The maximum and minimum in each image are normalized to white and black, respectively.

Steerable Filters

Steerable filters, first described by Freeman and Adelson [87], refer to the family of kernels whose rotations can be described by a linear combination of a set of basis kernels. The derivatives of a given order of a Gaussian kernel are primary examples. In the steerable version, the dominant orientation of the patch is first computed by analyzing the distribution of the gradients [88], and the filters are rotated accordingly in order to gain rotational invariance.

Figure 4.4 visualizes the derivatives of a Gaussian kernel up to fourth order. The first- and second-order derivatives can be appended to the mean patch values in order to form an 8-dimensional descriptor. Similarly, the first-, second-, third- and fourth-order derivatives can be used along with the mean patch values to form a 16-dimensional descriptor, after dropping one of the fourth-order derivatives. The standard deviation of the Gaussian was set to $\frac{5m}{32}$ where m equals the width and height of the underlying patch. Patch sizes of 8×8 and 16×16 were tested.

Gradient-based Descriptors

Gradient-based descriptors have been successful in keypoint localization and recall [88, 89, 90]. However, these descriptors are typically very large, ranging from 64 to 128 dimensions. Among these, SURF (speeded-up robust features) [90] is fast to compute. In the original SURF descriptor, the patch is partitioned into 4-by-4 subsquares, and in each square, four gradient statistics are computed: $\sum |dx|$, $\sum |dy|$, $|\sum dx|$, and $|\sum dy|$, leading to a 64-dimensional descriptor. To create a more compact descriptor, this is reduced into a 12-dimensional descriptor by partitioning the patch into 2-by-2 subsquares and only computing the first three gradient statistics. The resulting descriptor is augmented by the mean patch value, and yet another component is given by the standard deviation of the luminance channel, resulting in a 16-dimensional descriptor overall. Patch size of 16×16 was tested.

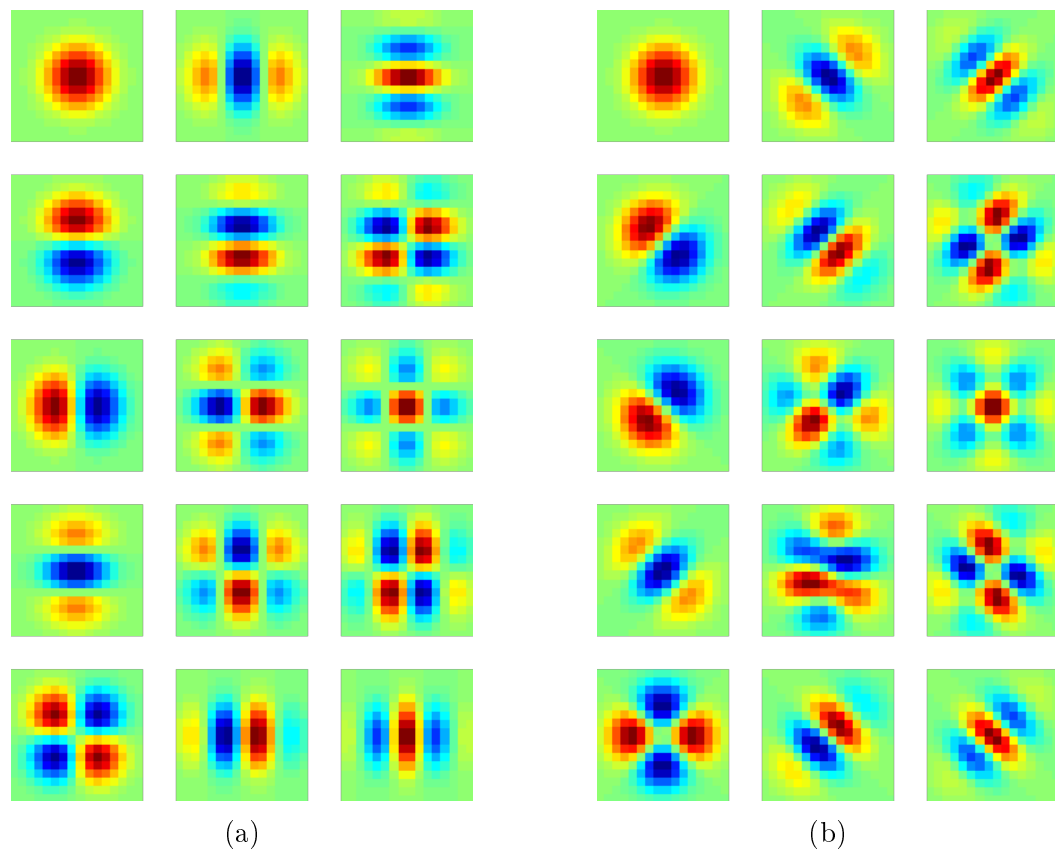


Figure 4.4: Steerable derivatives of the Gaussian kernel over a 16-by-16 patch, up to the fourth order, in addition to the zeroth order Gaussian kernel. **(a)**: The 15 kernels are shown in order, from left to right, and from top to bottom. **(b)**: The rotated versions (by 45 degrees) are shown. Each filter has been independently rescaled for visualization.

4.3.1 Evaluation

While evaluating the performance of a keypoint descriptor for retrieval tasks can be done in a straightforward manner by plotting the receiver operating characteristic (ROC) curves of the given descriptor [89], evaluating texture descriptors for the purpose of viewfinder editing is not straightforward. In existing spatial edit propagation tasks, methods are typically compared by first providing a detailed user input—strokes of multiple classes—and visually inspecting if distinct objects are segmented properly [62]; quantitative comparisons have been intractable.

There are two additional factors complicating the evaluation of texture descriptors for viewfinder editing: first, the spatial edit propagation framework in this dissertation relies on the fact that immediate user feedback is possible, so it can be and should be conservative. Second, only a single class is used in this framework, rather than multiple classes, in contrast to existing edit propagation work.

Hence, the evaluation was performed in the following manner: a camera application was implemented on a tablet to record hypothetical user interactions. It records up to 30 seconds of incoming viewfinder frames at 30 fps and also records the stream of touch events. The user was instructed to first stroke over objects or regions he would want to edit on the viewfinder, and then to introduce camera or object motion as desired. No visual feedback was given beyond the location of the touch events and the incoming viewfinder stream. 20 such sequences were captured and stored, shown in Figure 4.5. Then, the viewfinder editing was tested offline by replaying the sequences and the accompanying touch events, with two instances of the algorithm running side by side, using two different descriptors. The quality of the segmentation was noted visually and compared qualitatively, as done in the evaluation of existing spatial edit propagation papers.

4.3.2 Discussion

There are several observations made from the comparison of various texture descriptors. First, rotational invariance of a descriptor does not enhance its performance. The oriented and non-oriented versions of steerable filters exhibited equivalent performance. Two explanations are hypothesized: first, the descriptor is not being used for one-to-one matching, so the change in orientation is not problematic. Second, when the user strokes over a texture, several equivalent patches at varying orientation may be stroked over. In that case, the rotation of the scene or the object may cause each of them to generate a different descriptor, but taken as a set, the resulting descriptors should register a match against the original set of descriptors, albeit with a permutation within the set.

Second, using physical units for the underlying color space, such as LogLUV,



Figure 4.5: Datasets for descriptor evaluation. 20 sequences of still frames at 30 fps, each up to 30 seconds, along with touch events generated by the user, were acquired.

considerably outperformed using gamma-mapped YUV or RGB space. For instance, the FCam [49] implementation on the tablets used or the FlyCapture SDK [91] for the PointGrey Flea3 camera can return Bayer-mosaicked raw frames along with the frame metadata, which can then be reliably converted to LogLUV space. In contrast, the Android camera framework [92], while significantly more portable, is a black box and returns auto-exposed gamma-mapped YUV or RGB values, which cannot be converted to LogLUV space without knowing the camera response curve. (Even if the camera response were known, the YUV or RGB values provided are already quantized to 8 bits, so the resulting LogLUV values would have quantization artifacts.) Given that tuning the coefficients tied to the individual components of the descriptor requires knowing the noise characteristic of the incoming data, descriptors based on the latter camera API had difficulty handling a wide range of scenes.

Third, using high-dynamic-range (HDR) viewfinder frames also outperformed using low-dynamic-range (LDR) viewfinder frames. This difference is simply due to the fact that HDR imagery will contain less noisy image patches at either extreme of

pixel intensity. On the other hand, for LDR viewfinder frames, pixels may saturate, or be below the noise floor, in which case the descriptor is no longer reliable.

Fourth, a descriptor computed over a larger neighborhood outperformed the same type of descriptor computed over a smaller neighborhood. For each class of descriptor, computing it on a 16-by-16 patch invariably showed better performance qualitatively over computing it on a 8-by-8 patch, for instance. At 32-by-32, computing the descriptor became too expensive: each doubling of the width and height of the descriptor will quadruple the runtime of descriptor computation, unless the descriptor allows caching computation locally.

Fifth, in general, higher-dimensional descriptors generated better-looking selection masks than the lower-dimensional counterpart in terms of mask quality, at the cost of additional computation time. This should not be surprising, as the best keypoint descriptors have dimensionalities ranging from 64 to 128. It was determined that lookup of descriptors at $d = 16$ could be implemented to be fast enough to maintain interactive rates on a mobile device.

Sixth, the gradient-based descriptor outperformed the others, being able to handle complex textures better. All in all, the 16-dimensional SURF-like descriptor over 16-by-16 local context was found to be the best candidate.

4.4 Computing Edits Quickly and Robustly

The design of the pipeline outlined in this chapter must be sufficiently efficient in order to accommodate real-time viewfinder editing on a mobile device with limited computational resources. At the same time, the quality of the edit propagation must be adequate. Several modifications to the pipeline were motivated and engineered, in order to meet these requirements.

4.4.1 Subsampling Edit Masks

In order to accelerate the computation of edit masks, the edit masks are initially evaluated at samples placed in regular intervals. The resulting subsampled edit masks can

be upsampled with respect to the edges of the viewfinder image using an edge-aware joint-upsampling algorithm. The runtime of the mask generation is then effectively reduced by the square of the subsampling period. Although the cost of a joint-upsampling operation is incurred additionally, this cost is relatively small—as small as several milliseconds—compared to the cost of the lattice lookup, and a spatial smoothing operation is nevertheless helpful even when upsampling is not necessary, since it suppresses noise and unevenness in the edit masks.

Several subsampling periods (1, 2, 4, 8, 16) were tested. Among these, a subsampling factor of 4 at least was necessary to obtain reasonable frame rate on mobile platforms. Once the factor exceeds 8, the performance degraded considerably. For instance, Figure 4.6 shows a frame from a test dataset, processed with subsampling period of 8.

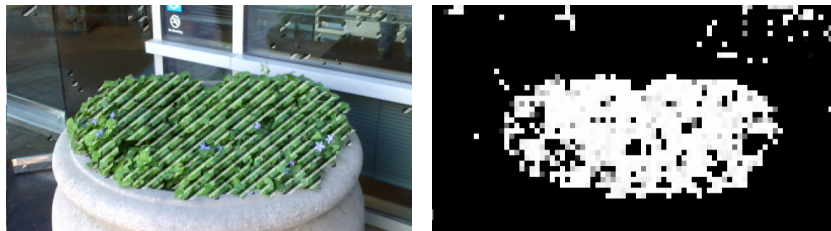


Figure 4.6: Subsampled edit mask. **(a)**: A still frame from one of the test datasets. **(b)**: The corresponding selection mask with subsampling period of 8, without any spatial smoothing applied. The mask has been resized with a nearest-neighbor up-sampling routine.

The joint-bilateral filter [14], edge-avoiding wavelets [58] and domain transform [59] were implemented as a potential joint-upsampling algorithm. It was empirically found that edge-avoiding wavelets were the fastest, with domain transform being essentially as fast. The joint bilateral filter was implemented in two flavors: first, a full joint bilateral filter was implemented, which was found to be too slow for this purpose; later, it was replaced by a simpler upsampling scheme that accesses only the four known samples at the corners of each enclosing square in the subsampling grid. Among these, domain transform was found to present a good balance between runtime efficiency and smoothing performance.

4.4.2 Multi-scale Lookup

Appearance-based edit propagation methods are often prone to false positives or false negatives in classifying image patches. Operating on a complex texture exacerbates this problem, as it becomes very likely that the user does not stroke over all possible exemplars, leaving out some patches that do belong to the same object. To address this issue, we rely on a multi-scale approach, in which we store the image patches the user strokes over at multiple resolutions, and combine the lookup response across scale to generate a single mask for each viewfinder frame. For this dissertation, three scales were used, with consecutive scales being apart by a factor of 2.

Figure 4.7 illustrates the usefulness of such multi-scale approach. As shown in the figure, regions with a diverse set of texture cannot be matched to the user selection simply via examining the neighborhoods at the finest scale. However, at a coarser scale, the distance between the patch and the ones in the lattice will be smaller, allowing the patch to be recognized as having been selected.

4.4.3 Spatiotemporal Smoothing

As discussed above, only a subsampled set of image patches are converted to descriptors and then looked up from the permutohedral lattice, saving a significant amount of time, but yielding edit selection masks at a resolution lower than that of the viewfinder. The resulting edit masks undergo edge-aware upsampling using domain transform with respect to the edges of the viewfinder content. Generalizing this, we also perform the recursive variant of the domain transform filter across frames temporally, in order to reduce temporal artifacts.

In essence, the filter blends the masks for the previous and the current frames together where the pixel difference is small to reduce the temporal noise, but preserves the current frame content where the difference is large—usually because the edges do not align due to camera motion; because of this non-linearity, it is not necessary to register the masks. This helps to suppress flickering in noisy regions, where the result of the lattice lookup is inconsistent because of spurious texture introduced by the camera noise.

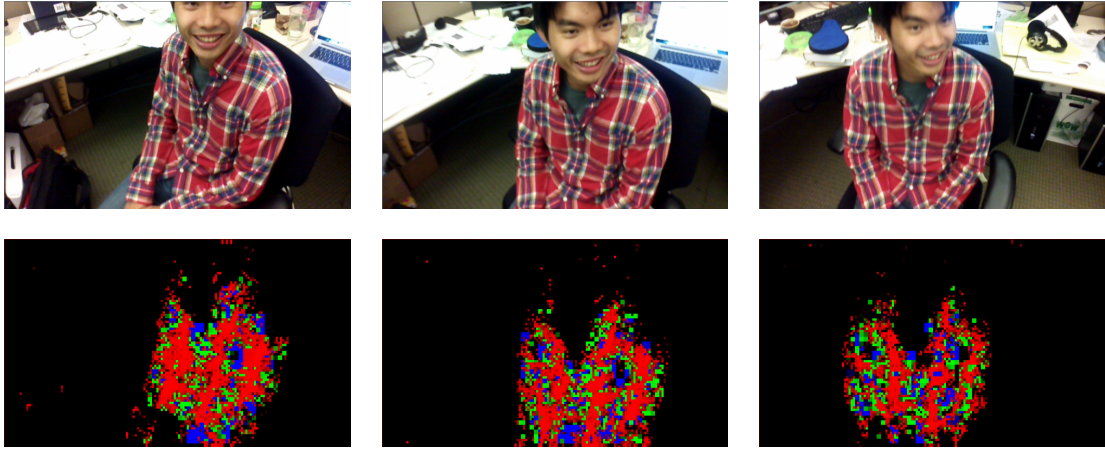


Figure 4.7: Demonstration of multi-scale texture lookup. **Top:** Three viewfinder frames sampled from a sequence captured on the tablet. In the sequence, the user strokes over the checked shirt of the subject. **Bottom:** Corresponding selection mask generated from multi-scale lookup, prior to any spatiotemporal smoothing. Red corresponds to pixels whose neighborhoods match the ones stored in the permutohedral lattice at the finest scale. Green corresponds to pixels whose neighborhoods return positive response from the lattice at a coarser scale (by a factor of 2), excluding the ones already marked red. Blue corresponds to an even coarser scale.

4.4.4 Offline Processing

Once the user is satisfied with the edits and presses the shutter, the necessary frames can be captured at full resolution of the sensor, and they can undergo the same processing as the viewfinder content, respecting the WYSIWYG property. This computation can happen offline, and as such, additional computation can be performed in order to generate edit masks with much higher quality. The image patches are processed at full resolution without the subsampling discussed in Section 4.4.1, generating a full-size edit mask. The full-size edit mask is still smoothed spatially with domain transform.

The resulting edit masks could still be improved by leveraging the existing body of work in offline edit propagation. Among the available algorithms, we choose manifold-preserving edit propagation [62], which exhibits higher mask quality over competing methods, albeit slow. To apply this algorithm, the full-resolution edit mask is first

thresholded and eroded to create a trimap. The trimap is then passed to manifold-preserving edit propagation in order to re-populate low-confidence areas of the edit mask, where confidence is given by the homogeneous coordinates in the lattice. This additional step was found empirically to help produce cleaner and more homogeneous masks. Figure 6.6 demonstrates an example of this process.

4.4.5 Discussion

Method	Runtime		
	Pre-computation	Per-frame cost	Total
Chen et al., 2012 [62]	2000.0 ms	—	2000.0 ms
An and Pellacini, 2008 [60]	1900.0 ms	—	1900.0 ms
Li et al., 2008 [63]	1500.0 ms	—	1500.0 ms
Farbman et al., 2010 [61]	580.0 ms	—	580.0 ms
Xu et al., 2009 [64]	433.0 ms	30.0 ms	463.0 ms
Bie et al., 2011 [65]	163.0 ms	15.0 ms	178.0 ms
Li et al. 2010 [66]	22.7 ms	20.0 ms	42.7 ms
Proposed method	—	13.6 ms	13.6 ms

Table 4.1: Runtime comparison for various edit propagation methods. Self-reported runtimes in previously published papers are used. The reported time for the resolution closest to VGA was taken, and was interpolated linearly to VGA resolution if necessary. No adjustment for the difference in the testing rigs was done. However, each of the above methods reported measuring its runtime on a multi-core Intel CPU.

The appearance-based edit propagation algorithm described thus far is considerably faster than other appearance-based methods in the literature. Table 4.1 lists the runtimes of recent work and also that of the proposed algorithm, decomposing the runtime into pre-computation cost and per-frame cost whenever such a division is applicable; as discussed in Chapter 2, many existing algorithms rely on performing a static analysis of the input dataset in a non-causal manner in order to build a classifier or a data structure, and then amortizing this cost over the multiple frames to be processed.

The proposed method represents a speed-up of an order of magnitude over all existing methods, with the exception of Li et al. [66]. However, Li et al. [66] relies on

the RGB pixel values (3 channels) to describe the local texture, while the proposed method employs a 16-dimensional descriptor. The proposed method is therefore considerably more robust, while still being three times faster. On the other hand, if the speed is less important—for instance, in offline processing—methods based on static analysis will outperform the proposed work in terms of mask quality.

4.5 Summary

This chapter defines the notion of viewfinder editing, and describes the underlying algorithms and the design process through which the algorithms were determined and developed. Viewfinder editing realizes spatiotemporally coherent edits on a live viewfinder via appearance-based edit propagation, and is based on the permutohedral lattice (Chapter 3). The performance necessary to run on the live viewfinder of a mobile platform is achieved by combining a number of approaches, including subsampling, multi-scale processing, spatiotemporal smoothing, and gradient-based texture descriptor. All in all, the performance of viewfinder editing represents an order-of-magnitude speed-up over that of competing methods.

Chapter 5

Appearance-Based Camera Control

Viewfinder editing, as designed in the previous chapter, enables a real-time interface to locally manipulating the viewfinder content. Interestingly enough, viewfinder editing can be used to bring the WYSIWYG aspect to stack-based computational photography, in which a set of frames at varying parameters are acquired and then composited together to yield the final output. Two prime examples are high-dynamic-range (HDR) image tonemapping and focal stack compositing. Instead of running the composition as an offline process, the camera application can provide the said composition interactively, directed by the user's interaction.

In fact, knowing the user's intention ahead of the capture time has an auxiliary benefit for stack-based computational photography that goes beyond regaining the WYSIWYG aspect: the camera application can rely on the user edits to determine the optimal capture parameters for the exposure or focal stack, capturing exactly and only the data necessary to accommodate the transform the user wishes to perform on the scene, ensuring that the captured data exhibits the sufficient dynamic range, depth of field, signal-to-noise ratio in each region of the scene to the extent necessary.

This approach diverges from the existing philosophy on data acquisition for stack-based computational photography. The present algorithms for determining camera parameters are aimed at faithfully acquiring the scene, attempting to maximize some measure of signal-to-noise ratio [29, 30]. This philosophy is rational when the post-processing to be performed on the acquired image is unknown, and when there is no

additional information on the relative saliency of different scene elements. In contrast, with viewfinder editing it is possible to fold the entire post-processing pipeline into the camera application, including tonemapping and local edits, and one can leverage this knowledge to optimize the capture parameters. For instance, if the user sees the tonemapped HDR image on the viewfinder, and proceeds to brighten a darkened region, a longer exposure may be necessary; if the user darkens clipped regions to reveal the details, a shorter exposure may be necessary.

This chapter discusses how to mathematically take advantage of the WYSIWYG property that arises from viewfinder editing—the knowledge of the local edits and the rest of the post-processing pipeline. A new multi-exposure metering and a new stack focusing algorithms are proposed. The new multi-exposure metering algorithm is evaluated against prior methods.

5.1 Appearance-Based HDR Metering

Appearance-based HDR metering extends the state-of-the-art HDR metering by incorporating the knowledge of the final post-processed output. Because the WYSIWYG interface by definition requires the application to deliver the final result in real time, the system must be computing the display value of each pixel. This value can then be used to derive the metering algorithm, allowing the algorithm to optimize for the quality of the tonemapped, edited output, instead of the quality of the scene luminance estimate as done in existing work. Figure 5.1 illustrates this distinction.

5.1.1 Image Appearance Model

We assume a c -bit linear camera whose sensor captures the scene and records values $p(x, y)$, which are perturbed by additive Gaussian noise as shown in Equation (5.1):

$$p(x, y) = \min(2^c - 1, L(x, y) \cdot t \cdot K + N(0; \sigma_r)), \quad (5.1)$$

where $L(x, y)$ is the physical scene luminance; t is the exposure time; K is a calibration constant; $N(0; \sigma_r)$ is the Gaussian read noise; and the measurement has been clamped

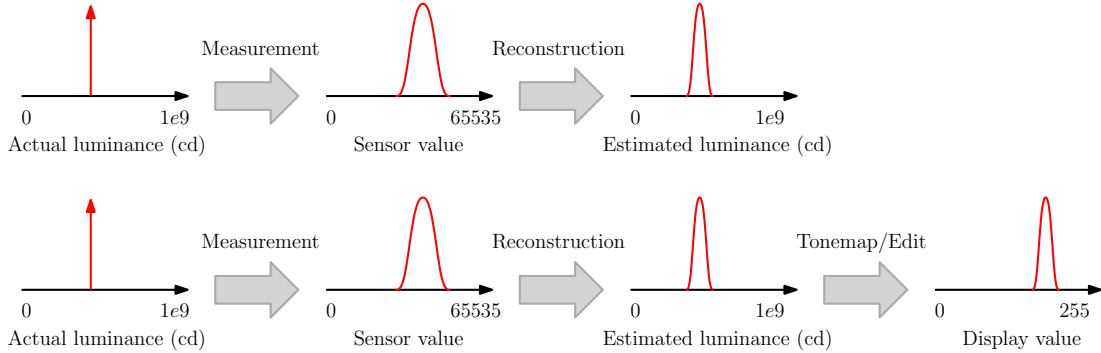


Figure 5.1: Comparison against existing multi-exposure metering methods. **Top:** The standard imaging pipeline assumed by existing multi-exposure metering methods. Going from the left, the actual luminance present in the scene is observed by the sensor with some amount of uncertainty, becoming a probability distribution of sensor values. From this, the algorithm estimates the actual luminance, which is used as a basis for the subsequent optimization process. **Bottom:** The augmented pipeline with a WYSIWYG interface. The tonemapping operator and its parameters, along with any edits, are available to the system, since the final result must be provided to the user in real time. As such, the displayed pixel value (rightmost) are known and can be used as a basis for the subsequent optimization process instead.

to properly reflect the possibility of saturation. From this model, one can reconstruct an unbiased estimate of the scene luminance with Equation (5.2), barring saturation— for high-dynamic-range scenes in which saturation is a problem, multiple photos may be taken to improve the fidelity of the estimate:

$$\tilde{L}(x, y) = \frac{p(x, y)}{t \cdot K}. \quad (5.2)$$

The estimate for the linear scene luminance finally undergoes a monotonic tonemapping operator T to be shown on a k -bit display, resulting in the final pixel value $\tilde{I}(x, y)$. In addition, $\tilde{I}(x, y)$ includes a possible per-pixel manipulation of the scene luminance with an edit mask $M_E : \{(x, y)\} \rightarrow \mathbb{R}^+$ prior to tonemapping:

$$\tilde{I}(x, y) = \min \left(2^k - 1, T \left(\tilde{L}(x, y) \cdot M_E(x, y) \right) \right). \quad (5.3)$$

In Equation (5.3) above, M_E is equivalent to the power of two raised to the appropriate number of stops for either brightening or darkening the pixel. Note that this model can fully handle nonlinear cameras by folding the camera response function into T .

The computed pixel intensity $\tilde{I}(x, y)$ still suffers from the uncertainty inherited from the sensor measurement. In the context of an image patch of a particular brightness, this per-pixel uncertainty in $\tilde{I}(x, y)$ manifests itself as spatial noise in $I(x, y)$. Hence, to curb the appearance of spatial noise on display, the uncertainty in $\tilde{I}(x, y)$ should be quantified and bounded.

To quantify the uncertainty in $\tilde{I}(x, y)$, recall that the estimate $\tilde{L}(x, y)$ in fact represents a Gaussian probability distribution. Provided that T is smooth and the standard deviation of $\tilde{L}(x, y)$ is relatively small, one can apply first-order approximation to model the probability distribution corresponding to $I(x, y)$ as a Gaussian, with its standard deviation $\sigma_I(x, y)$ computed as follows:

$$\begin{aligned} \frac{\sigma_I(x, y)}{\Delta\tilde{I}(x, y)} &\approx \frac{\sigma_r}{\Delta p(x, y)} \\ \implies \sigma_I(x, y) &\approx \sigma_r \cdot \frac{dI(x, y)}{dp(x, y)} \\ \implies \sigma_I(x, y) &\approx \frac{\sigma_r \cdot M_E(x, y)}{t \cdot K} \cdot T' \left(\tilde{L}(x, y) \cdot M_E(x, y) \right), \end{aligned} \quad (5.4)$$

where $T'(\cdot)$ is the first-order derivative of T with respect to its parameter. The relative error for this first-order approximation is below 5% for all but 3% of possible pixel values for a gamma tonemapping operator at $\gamma = 2.2$ with $\sigma_d = 0.01(2^k - 1)$, proving that the linear approximation holds well.

The acceptable per-pixel uncertainty is a function of the displayed imagery, as well as the viewing condition, including the viewer's visual adaptation level. However, for the purpose of computational photography applications, a bright display and photopic vision for the viewer are assumed, leading to a fixed constant (denoted by σ_d hereafter)

for the threshold on per-pixel uncertainty. Then, requiring that the estimated per-pixel uncertainty is bounded above by σ_d , we derive,

$$\frac{\sigma_r}{K} \cdot \frac{M_E(x, y) \cdot T' \left(\tilde{L}(x, y) \cdot M_E(x, y) \right)}{\sigma_d} \leq t. \quad (5.5)$$

Because Equation (5.4) is applicable only if the pixel is not saturated in the sensor, we enforce an upper bound as to avoid sensor saturation:

$$t < \frac{2^c - 1}{K \cdot \tilde{L}(x, y)}. \quad (5.6)$$

If a pixel saturates on display after editing and tonemapping, it may be permissible for the sensor to saturate. However, sensor saturation causes the scene luminance estimate to be incorrect. This is allowable, provided that the resulting incorrect estimate will still cause the pixel to saturate on display:

$$t \leq \frac{(2^c - 1)M_E(x, y)}{K \cdot T^{-1}(2^k - 1)}. \quad (5.7)$$

In summary, by placing a threshold on the visual quality of the displayed pixel values, one can obtain lower and upper bound on the exposure that will guarantee meeting the said threshold. Each pixel in the image yields a lower and upper bound on the exposure.

5.1.2 Per-Pixel Objective Function

Before the lower and upper bound on the exposure for each pixel can be aggregated, it must first be converted into a meaningful objective function. This objective function should penalize exposure times outside the lower and upper bounds. Denote by $B_*(x, y)$ and $B^*(x, y)$ the bounds derived at each pixel using Equations (5.5–5.7). For exposure values outside the interval $[B_*(x, y), B^*(x, y)]$, the objective function has a value of 0.

Otherwise, for the region within the interval $[B_*(x, y), B^*(x, y)]$, the objective

function should have a positive score. There are the following considerations for the values within this region: on one hand, the objective function should favor shorter exposures in order to reduce the total exposure time. On the other hand, longer exposures do yield increase in SNR. These two concerns can be balanced by providing a linear dependence on the logarithm of exposure value, determined empirically. The logarithm of the exposure value is used, in lieu of the exposure value itself, because the SNR rises linearly with the former, not the latter.

By combining these guidelines, the following objective function was born:

$$J(x, y, t) = \begin{cases} 0, & \text{if } t \notin [B_*(x, y), B^*(x, y)], \\ 1 + \alpha(x, y) \log_2 \frac{t}{B_*(x, y)}, & \text{otherwise,} \end{cases} \quad (5.8)$$

illustrated in Figure 5.2 (a) and (b) on a logarithmic time axis, with $\alpha(x, y) = -0.3$ if the pixel is saturated on display, and 0.0 otherwise. The linear coefficient $\alpha(x, y)$ is negative for pixels saturated on display, because increasing exposure does not increase the SNR of the displayed pixel in this case.

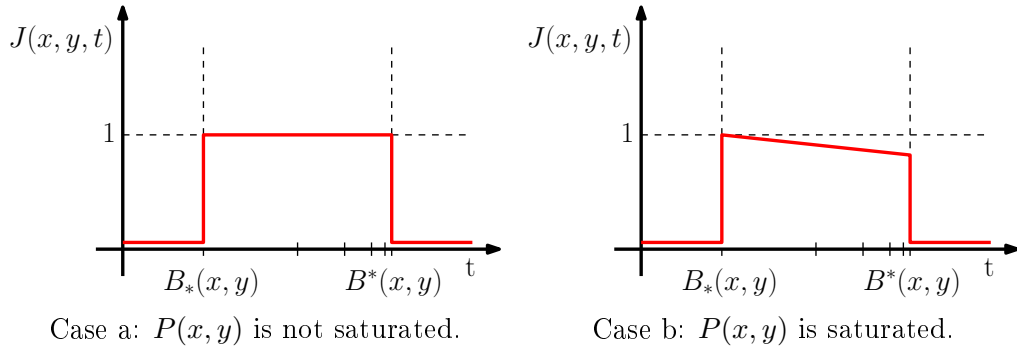


Figure 5.2: Appearance-based metering via per-pixel analysis. For each pixel on the screen, we compute the minimal and maximal permissible exposure values, accounting for the local and global transforms raw sensor values undergo. **(a, b)**: For metering, each pixel yields an objective function $J(x, y, t)$ based on the minimum and maximum per-pixel exposure values $B_*(x, y)$ and $B^*(x, y)$. For instance, **(a)** signifies that any exposure value between $B_*(x, y)$ and $B^*(x, y)$ is equally desirable, whereas the plot in **(b)** suggests that while any exposure value in the range is acceptable, shorter exposures are preferred.

5.1.3 Aggregating Per-Pixel Objectives

The per-pixel objective function in Equation (5.8) can now be aggregated across the entire image to form a single objective function. Multi-exposure metering then can be performed by analyzing this function. Gallo et al. [30] note that most scenes are handled by three exposures or fewer, and that most cameras offer only a limited number of possible exposure values. In practice, the domain of this function can be discretized to a few values on a geometric scale—256 in the implementation in Chapter 6—so that evaluating the function now can be done on a discrete set.

For multi-exposure metering that captures more than one shot, the choice of the multiple exposures affect one another. As such, they should be optimized simultaneously in order to guarantee global optimality. However, for simplicity and ease of implementation, a greedy approach that seeks to iteratively maximize the aggregate objective function $\sum_{x,y} J(x, y, t)$ with respect to the exposure time t was implemented instead and is described herein.

When t is mapped to logarithmic domain, the objective in Equation (5.8) becomes a sum of piecewise-linear functions, which can be maximized in linear time using dynamic programming, by pre-computing and caching the first- and second-order derivatives of the objective. The algorithm, shown in Algorithm 5.1, greedily finds exposures that maximize the objective. The algorithm is run iteratively, each time removing the pixels from consideration whenever their requirements are met. The procedure terminates upon reaching the maximum stack size or satisfying a certain percentage of per-pixel requirements.

5.2 Appearance-Based Focal Stack Focusing

Another popular target for manipulation is the depth of field. A focal stack, a set of images focused at different depths, can be combined to simulate extended depth of field [41]; reduced depth of field can be obtained likewise [43]. See Figure 2.4 for example. In focal stack composition, an *indexing map* that specifies the depth from which each pixel should be drawn is first created, and the final composition is obtained

Algorithm 5.1 A dynamic programming routine for solving the global objective function for the optimal exposure in linear time.

Require: $T = \{t_0, t_1, \dots, t_{n-1}\}$, a discretized set of possible exposure values in an increasing geometric sequence with ratio r .

Require: Per-pixel lower and upper bound on exposures, $B_*, B^* : \{(x, y)\} \rightarrow T$.

Require: Arrays $J, A, B[0, \dots, n-1]$, initialized to zero. They respectively model $\sum_{(x,y)} J(x, y, t)$ and its first- and second-order differentials with respect to t .

- 1: **for each** (x, y) in the image **do**
- 2: $k_* \leftarrow$ the index such that $t_{k_*} = B_*(x, y)$
- 3: $k^* \leftarrow$ the index such that $t_{k^*} = B^*(x, y)$
- 4: $B[k_*] \leftarrow B[k_*] + \alpha(x, y) \log_2 r$ $\triangleright B[\cdot]$ keeps track of the aggregate slope.
- 5: $B[k^*] \leftarrow B[k^*] - \alpha(x, y) \log_2 r$
- 6: $A[k_*] \leftarrow A[k_*] + 1$ $\triangleright A[\cdot]$ keeps track of the aggregate step.
- 7: $A[k^*] \leftarrow A[k^*] - (1 - (k^* - k_*)\alpha(x, y) \log_2 r)$
- 8: $d \leftarrow 0$
- 9: **for** $i \in \{0, \dots, n-1\}$ **do**
- 10: $J[i] \leftarrow J[i-1] + A[i] + d$ \triangleright Here $J[-1]$ is defined as zero.
- 11: $d \leftarrow d + B[i]$
- 12: $k \leftarrow \operatorname{argmax}_i J[i]$ where $i \in \{0, \dots, n-1\}$
- 13: **return** t_k

by indexing into the focal stack, which is effectively a three-dimensional volume.

If the user can interactively specify the desired manipulation prior to capture and verify it via visualization in the viewfinder, the autofocus routine can deduce the minimal focal stack needed, instead of capturing the full focal stack. In other words, the system acquires only the parts of the three-dimensional volume that are necessary.

5.2.1 Image Appearance Model

Recall the well-known Gaussian lens formula [34], which relates the physical depth of the object to the distance between the lens and the sensors in the thin-lens model shown in Figure 5.3:

$$\frac{1}{f} = \frac{1}{d_o} + \frac{1}{d_i}, \quad (5.9)$$

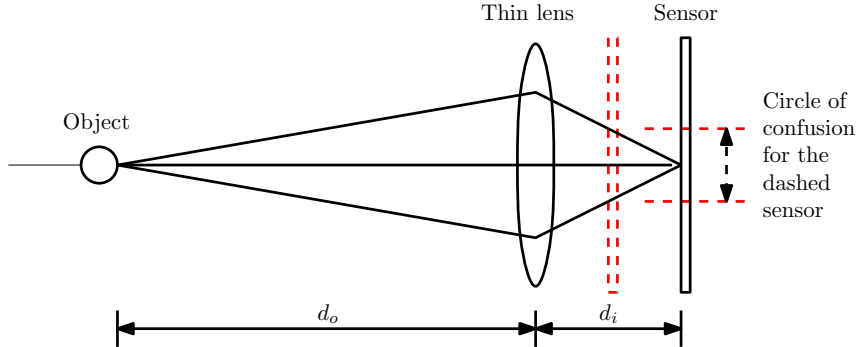


Figure 5.3: The thin-lens model. In the thin lens model, the distances from the lens to the object and to the image, denoted by d_o and d_i respectively, follow a mathematical formula known as the Gaussian lens formula (Equation (5.9)), provided that the object is in focus. If the formula is not met, as in the case of the misplaced sensor in dashed lines, a defocus blur known as the circle of confusion will result.

where f is the focal length of the lens, and d_o and d_i are the distances from the lens to the object and to the image on the sensor planes, respectively, provided that the object is currently in focus.

It is common to quantify the state of a camera lens module in diopters, which corresponds to inverse of the distance to the object in focus. For instance, focusing at infinity is equivalent to 0 diopter, and focusing at $1m$ away is equivalent to 1 diopter. Using the Gaussian lens formula in Equation (5.9), one can show that the diopter measure is mathematically equal to $1/f - 1/d_i$. In turn, any offset in diopters from this value results in defocus.

Let $[z_{min}, z_{max}]$ be the range of diopters supported by the lens module. The user interaction begins with a reference photograph focused at a particular depth, corresponding to $z_0 \in [z_{min}, z_{max}]$ diopters. The user then paints a mask $F : \{(x, y)\} \rightarrow [-1, 1]$ via viewfinder editing, specifying which regions should be sharper or blurrier in a reference photograph focused at depth. Meanwhile, the viewfinder stream cycles through a number of focus settings to continuously acquire the scene at various depths and builds a rough depthmap $z_* : \{(x, y)\} \rightarrow [z_{min}, z_{max}]$ (in diopters) based on a local contrast measure.

Then an indexing map $\hat{z} : \{(x, y)\} \rightarrow [z_{min}, z_{max}]$ can be created as follows:

$$\hat{z} : (x, y) \mapsto \begin{cases} z_0, & \text{if } F(x, y) = 0, \\ z_*(x, y), & \text{if } F(x, y) = 1, \\ z_{min}, & \text{if } F(x, y) = -1 \text{ and } z_0 < z_*(x, y), \\ z_{max}, & \text{if } F(x, y) = -1 \text{ and } z_0 \geq z_*(x, y), \\ \text{linearly interpolate,} & \text{otherwise.} \end{cases} \quad (5.10)$$

The rationale behind the construction of the indexing map in Equation (5.10) is that when the user performs no edit ($F(x, y) = 0$), the reference depth is specified; when the user maximally sharpens the region ($F(x, y) = 1$), the pixels should be drawn from the depth corresponding to the region; on the other hand, when the user maximally blurs the region ($F(x, y) = -1$), the pixels should be drawn from the depth that would maximally defocus the region, away from the actual depth. For any other values of $F(x, y)$, linear interpolation is used for the indexing map.

Once the indexing map \hat{z} is computed, the final composition can be constructed as described earlier. Since \hat{z} contains a continuous range of diopter values, it is possible that a slice focused at the exact depth is unavailable in the focal stack. In such cases, the two nearest slices are accessed, and the looked-up values are linearly interpolated.

5.2.2 Per-Pixel Objective Function

The indexing map \hat{z} described in Equation (5.10) can serve as a basis for a per-pixel objective function. The per-pixel objective function penalizes deviation from the desired diopter value given by the indexing map, as visualized in Figure 5.4 and detailed mathematically in Equation (5.11).

$$J(x, y, z) = \max \left(0, \frac{\epsilon - \|z - \hat{z}(x, y)\|}{\epsilon} \right). \quad (5.11)$$

The per-pixel objective is 1 at the desired depth $\hat{z}(x, y)$, linearly reducing to zero at depth error ϵ . In the implementation in Chapter 6, the value of $\epsilon = 1.0$ is

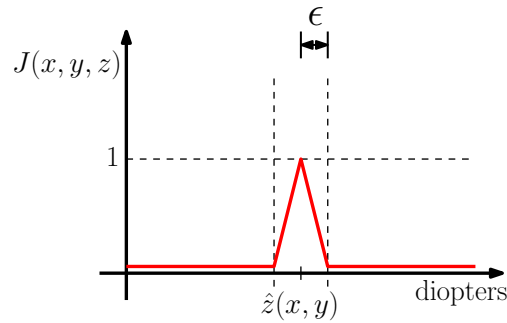


Figure 5.4: Appearance-based focusing via per-pixel analysis. For each pixel on the screen, the desired depth at which the lens should be focused is first computed in form of an indexing map. This gives rise to an objective function that penalizes deviation from the indexing map.

used for a lens with $z_{min} = 0.0$ and $z_{max} = 10.0$ diopters. Because this per-pixel objective is linear as a function of diopters, it can be aggregated over all pixels and be optimized quickly as in Section 5.1.3: the optimal focus distance is iteratively selected by maximizing the aggregate objective, while holding out pixels whose per-pixel objective functions achieve nonzero values at previously chosen focus distances.

5.3 Evaluation

In this section, the proposed appearance-based HDR metering method is evaluated against the existing state of the art. On the other hand, the prior work on determining focus values for a focal stack is sparse: there exists an algorithm designed for obtaining the minimal focal stack required for an *all-focus* composition [42], but existing work on general focal stack composition, such as Jacobs et al. [43], assumes that the entire focal stack is captured. Thus for focal stack composition, see the empirical results shown in Chapter 7.

5.3.1 Methodology

The appearance-based HDR metering proposed in Section 5.1 was evaluated against the HDR metering method of Hasinoff et al. [29] in an offline comparison, and against

a generic histogram-based HDR metering—akin to Gallo et al. [30]—in an online comparison. Hasinoff et al. solves for the set of exposures that maximizes the worst per-pixel SNR across the image. A MATLAB implementation was written to perform an exhaustive global search over the parameter space in order to solve this maximization. Histogram-based HDR metering computes the luminance histogram of the scene, and greedily picks the exposures in an iterative fashion, holding out histogram bins that are well-covered by previously chosen exposures. Once the metering method to be evaluated computes the metering parameters for the given scene, we capture the corresponding exposure stack, register and blend them, tonemap the resulting HDR image, and perform any local edits specified prior to capture. The tonemapping function, its parameters, and any local edits are known prior to capture, following the assumptions in appearance-based camera control. The appearance-based HDR metering makes use of this knowledge.

For the offline comparisons, the scene to be captured was simulated in one of two setups. In the first setup, publicly available high-dynamic range radiance maps of static scenes were used to generate the requested exposure stack. Additive read noise reported by Granados et al. [24] for a point-and-shoot camera was included, and a 12-bit sensor was assumed.

The second setup was designed to objectively study the effect of exposure duration on moving scenes. A high-frame-rate image sequence was captured at 180 fps. From this image sequence, any photograph whose exposure duration is an integer multiple of $1/180$ " could be synthesized by simply accruing these frames. Photographs of other exposure duration could be synthesized by integrating the appropriate frames with fractional weights. Hence, the exactly same camera and scene motion can be provided to multiple metering methods, and the results can be compared against one another.

Lastly, an online comparison was performed on a static scene. A ground truth radiance map was first obtained via a redundant exposure stack on a tripod, and then the candidate metering methods were run serially.

5.3.2 Noise Characteristics for Static Scenes

In this set of comparisons, the appearance-based HDR metering proposed in Section 5.1 was evaluated against the HDR metering method of Hasinoff et al. under fixed time budget, using publicly available high-dynamic range radiance maps of static scenes. That is, each metering method was allowed to freely allocate a fixed amount of time across multiple shots. The final tonemapped results are shown in Figures 5.5 through 5.7. Figure 5.5 and Figure 5.7 highlight the improvement in the image quality for the proposed method over the prior work.

Studying Figure 5.6 reveals an interesting intuition that explains the reason behind the performance of the proposed method. Because both metering methods have the same time budget, the metering process can be seen as “distributing” a set amount of noise across the scene histogram. Because the appearance-based metering outperforms the prior work in the insets provided in Figure 5.5, it stands to reason that the prior work may actually be better in some other areas. Figure 5.6 demonstrates precisely this point. However, appearance-based metering puts most of the noise in regions in which the noise would not be perceptually conspicuous, by design. On the other hand, Hasinoff et al. tries to improve the SNR of the underexposed region at the cost of the SNR of other areas, which are perceptually more important in the tonemapped image.

5.3.3 Metering for Moving Scenes

The aforementioned high-frame-rate datasets containing scene and camera motion were used to compare the proposed method against that of Hasinoff et al. We allow each metering method to freely minimize the total time budget on its own, but control for the final image quality by fixing one of the exposures. No noise was synthetically added to the exposure stacks. The results are shown in Figure 5.8. In general, appearance-based HDR metering is able to capture the scene satisfactorily in less time than the prior work, which mitigates the halo and ghosting artifact that result from scene motion shown in Figure 5.8c.

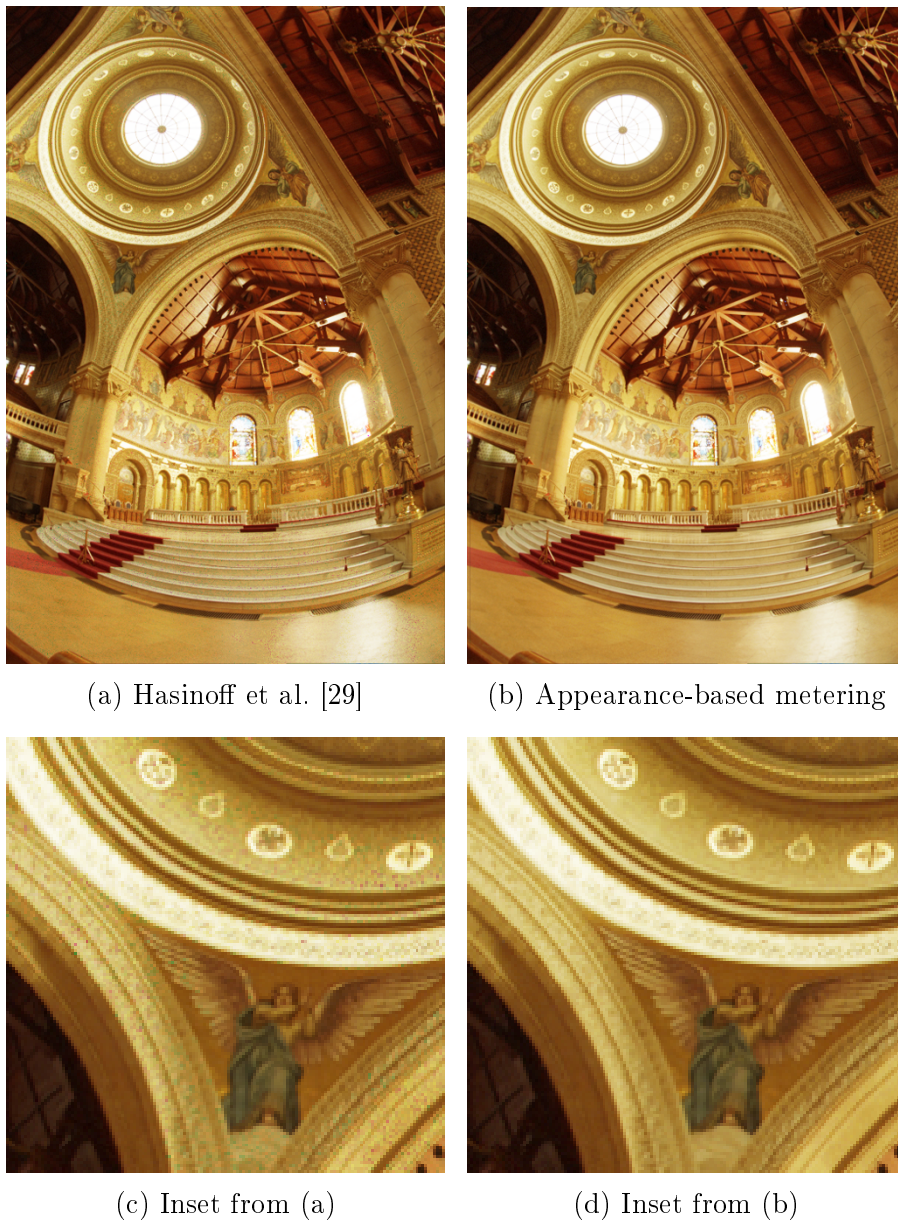


Figure 5.5: Synthetic comparison of appearance-based HDR metering against Hasinoff et al. [29] on church dataset. In each case, a total time budget of 103 ms was allowed. **(a)**: The HDR composite resulting from the metering of Hasinoff et al. (0.027 ms, 1.70 ms, 101.30 ms). **(b)**: The HDR composite resulting from the appearance-based metering proposed in Section 5.1 (0.50 ms, 9.96 ms, 92.18 ms). **(c, d)**: Insets from (a) and (b). Reduction in noise is clearly visible for appearance-based metering. The input dataset is courtesy of Paul Debevec [5].

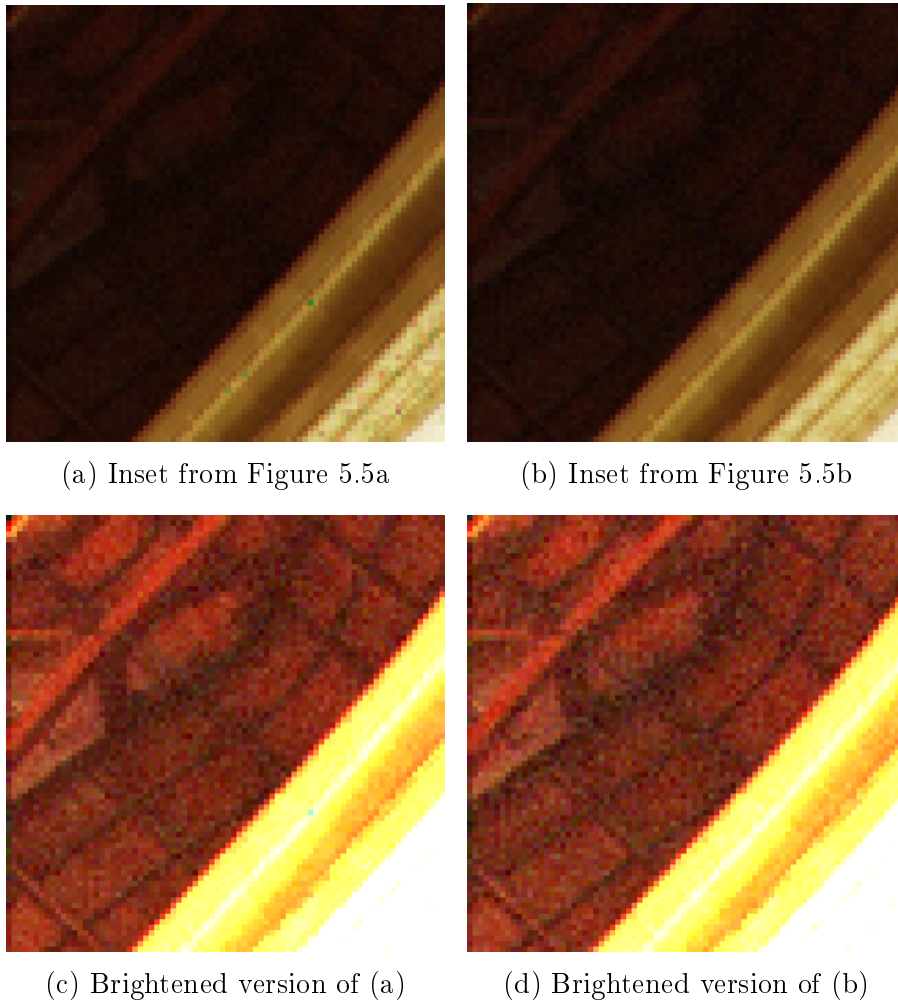


Figure 5.6: The distribution of noise in appearance-based metering. **(a, b)**: Insets from the top-left corners of the images from Figures 5.5a and 5.5b. The wooden structure in the shadow exhibits some noise in both cases, but are qualitatively similar in terms of perceivable noise. **(c, d)**: The insets have been artificially brightened. Appearance-based metering in (d) exhibits marginally more noise near the wooden buttress in this case, but only when the image has been artificially brightened. In summary, when working with a fixed budget, a metering method must “distribute” noise across the histogram, and appearance-based HDR metering is designed to put most of this noise in regions in which noise would not be perceptually conspicuous.

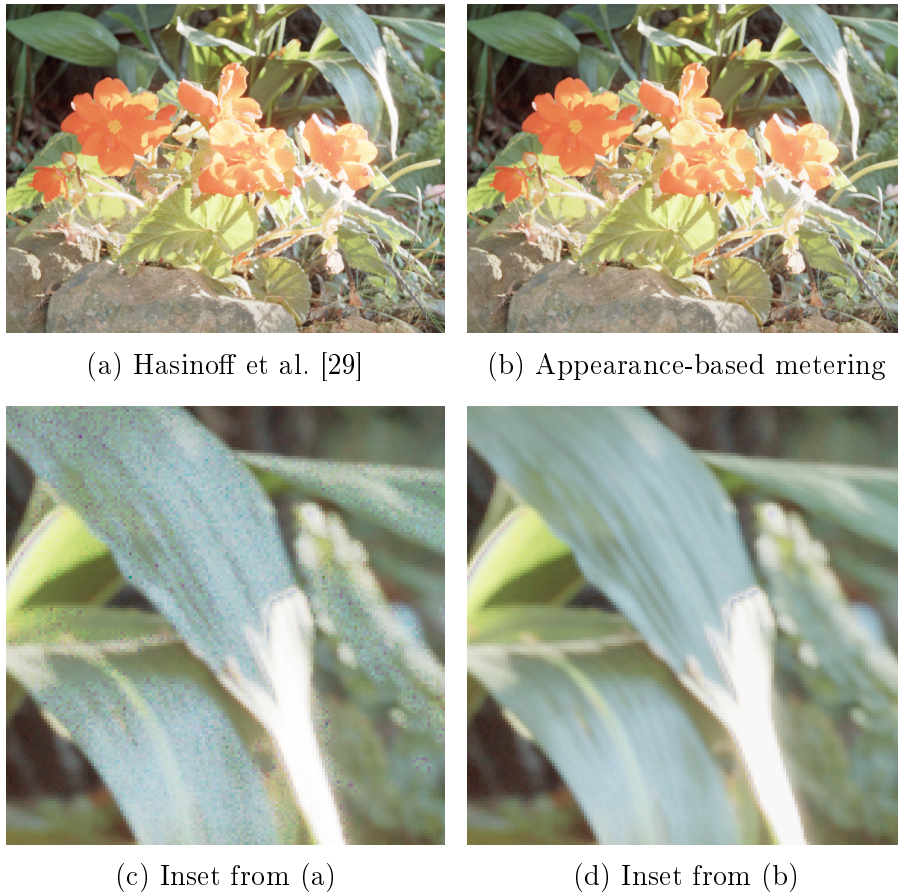


Figure 5.7: Synthetic comparison of appearance-based HDR metering against Hasinoff et al. [29] on `flower` dataset. In each case, a total time budget of 103 ms was allowed. **(a)**: The HDR composite resulting from the metering of Hasinoff et al. (0.027 ms, 1.70 ms, 101.30 ms). **(b)**: The HDR composite resulting from the appearance-based metering proposed in Section 5.1 (0.50 ms, 9.96 ms, 92.18 ms). **(c, d)**: Insets from (a) and (b). Reduction in noise is clearly visible for appearance-based metering, especially in the bottom-right quadrant of the inset. The input dataset is courtesy of Greg Ward.



Figure 5.8: Comparison of appearance-based HDR metering against Hasinoff et al. on a moving scene. The scene was simulated from a high-frame-rate video, so that the two metering methods receive the exposure stacks with the identical scene and object motion. We fixed one of the exposures and allowed the two metering methods to independently minimize the total time budget. The flower was locally brightened during viewfinding. **(a)**: The HDR composite resulting from the metering of Hasinoff et al. (5.0 ms, 80.0 ms). **(b)**: The HDR composite resulting from the appearance-based metering proposed in Section 5.1 (5.0 ms, 22.0 ms). **(c, d)**: Insets from (a) and (b). Longer total exposure time for the prior work results in a more pronounced ghosting artifact in the resulting tonemapped output.

5.3.4 Empirical Results

The simulated results thus far lend credibility to the hypothesis that appearance-based HDR metering can obtain better image quality and also reduce motion artifacts. This was verified by online testing on both static and dynamic scenes.

The first online test was conducted by setting up a static scene, and capturing an HDR radiance map via four methods: an exhaustive multi-exposure bracket (13 exposures in total with consecutive shots being a stop apart), appearance-based HDR metering, histogram-based HDR metering, and single-exposure metering. The data acquired via the first method served as the ground truth. All four acquisitions were conducted serially on a tripod. Figure 5.9 shows the result obtained with the latter three methods, and visualizes the relative error compared to the groundtruth. As seen in the figure, the final tonemapped and edited image exhibits a considerably higher SNR in the edited region when the stack was captured with appearance-based HDR metering, compared to histogram-based HDR metering or single-exposure metering.

We also performed an online comparison on a dynamic scene with a histogram-based HDR metering method, which is also fast enough to run in real time. The results shown in Figure 5.10 corroborate the findings in the previous experiments that appearance-based metering can help mitigate ghosting artifacts by minimizing the total exposure time better than prior work. Surprisingly, despite having shorter total exposure time than its competitor, it exhibits considerably cleaner midtone areas in Figure 5.10d, demonstrating that being aware of the post-processing pipeline is beneficial to the capture process.

5.3.5 Effect of Local Edits

In order to demonstrate the effect of local edits on metering, the appearance-based metering algorithm was used to generate two HDR composites in Figure 5.11, once without any local edits and once with a local edit that brightens a backlit black statue. As shown in the figure, different exposure stacks were produced: the former case produced a 2-image stack with 16 ms total exposure, whereas the latter case produced a 3-image stack with 96 ms total exposure, tacking on an extra shot in

order to accommodate the edit the user specified.

5.4 Summary and Discussion

This chapter augments the existing state-of-art methods on multi-exposure camera control by further taking into account the WYSIWYG property that arises from viewfinder editing (Chapter 4.) Whereas existing methods analyze how camera settings affect the quality of the raw radiance map, the proposed method goes further down in the pipeline and analyzes how camera settings affect the quality of the tonemapped, edited output displayed to the user.

Both offline and online comparisons with the existing state-of-the-art methods for HDR metering suggest that the common wisdom of capturing the entire scene radiance faithfully may not be the best solution in all cases. Today's displays have considerably less bitdepth than the HDR composites (or even a single RAW image produced from a commodity sensor), necessitating a compressive tonemapping operation. In addition, some regions of the image may be brightened or darkened by the user. In the end, the signal-to-noise ratio is not equally important across all the pixels, and appearance-based HDR metering takes advantage of this fact. Existing HDR metering methods would be more suitable over the proposed appearance-based HDR metering for work that requires true offline post-processing in softwares like Adobe Photoshop or Lightroom, since the offline post-processing cannot be predicted during capture. However, in those crucial situations, photographers will rely on tripods and capture an overcomplete stack.

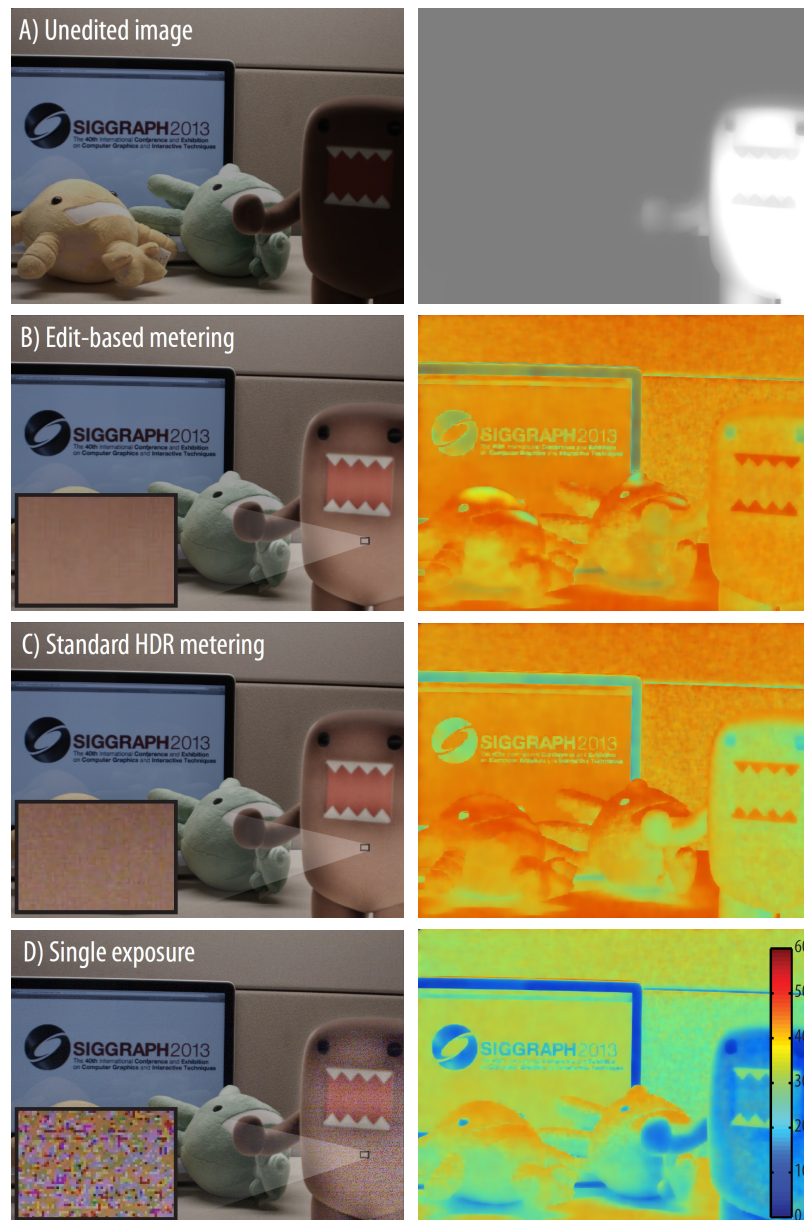


Figure 5.9: An empirical comparison of appearance-based HDR metering against generic histogram-based HDR metering on a static scene. (a): The unedited tonemapped output, along with the edit mask to be applied (computed during viewfinder editing). (b, c, d): Descending from the top, the edited tonemapped output using appearance-based HDR metering, histogram-based HDR metering, and a single-shot metering. On the right hand side is the visualization of the output SNR, using the groundtruth obtained from an exhaustive multi-exposure bracket, is shown.

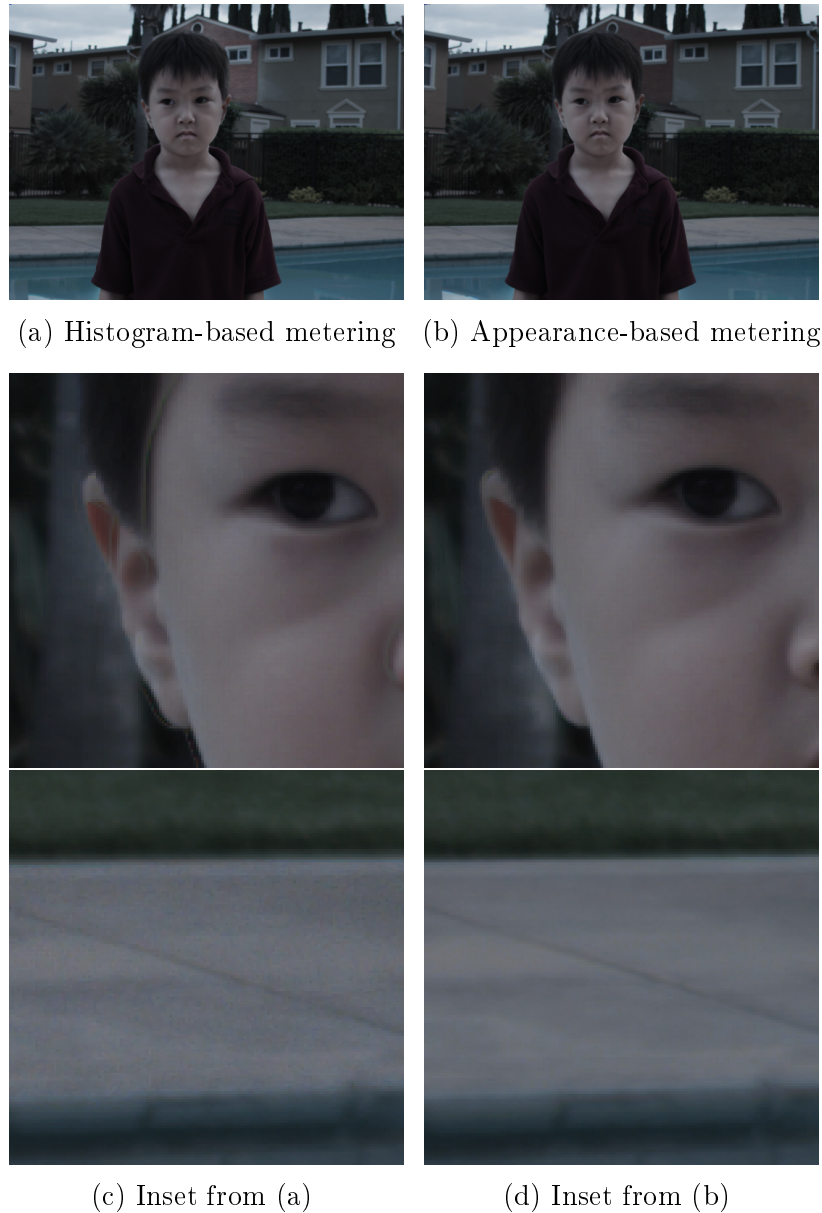


Figure 5.10: An empirical comparison of appearance-based HDR metering against generic histogram-based HDR metering on a dynamic scene. In this example, the backlit face of the child was brightened prior to capture. **(a)**: The composite resulting from the histogram-based HDR metering (0.579 ms, 9.958 ms, 23.879 ms). **(b)**: The composite resulting from the appearance-based metering proposed in Section 5.1 (0.645 ms, 5.555 ms, 11.101 ms). **(c, d)**: Insets from (a) and (b). The proposed method mitigates ghosting artifact, but is also able to reduce noise despite having a shorter total exposure time.

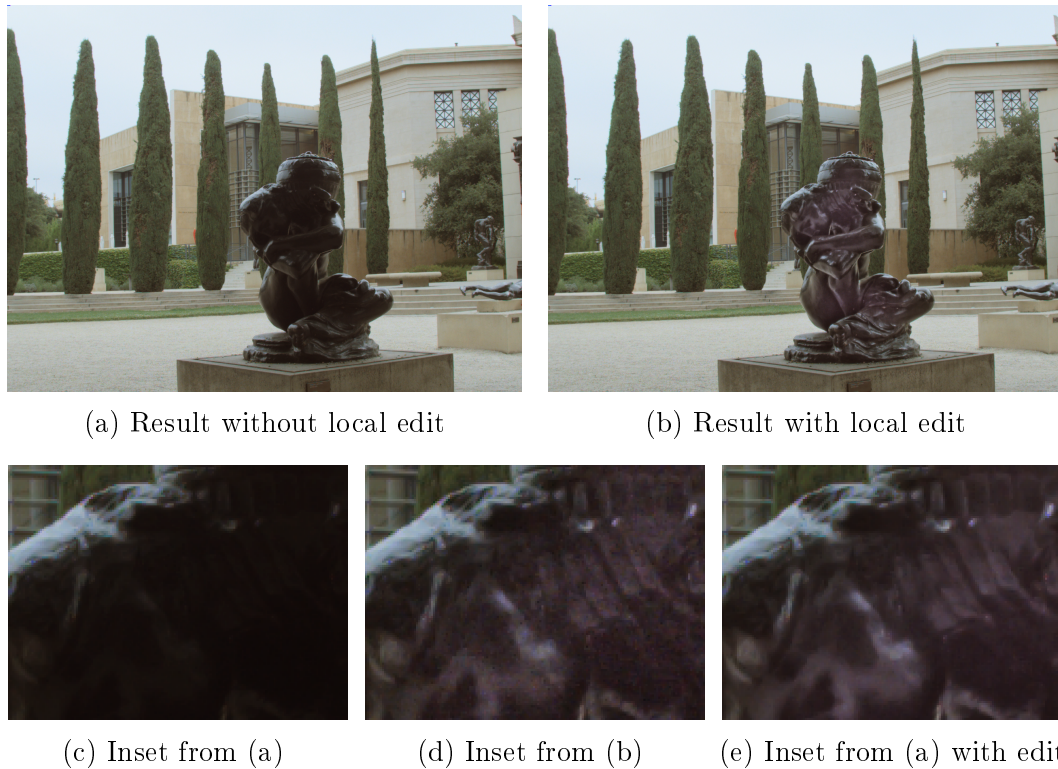


Figure 5.11: The real-time effect of local tonal edits on appearance-based metering. **(a)**: This tonemapped image was produced from a 2-image stack at (1.607 ms, 14.874 ms), as determined by the appearance-based metering. **(b)**: The user brightened the dark statue on the viewfinder and retook the photo. Our algorithm automatically adjusted to the new appearance on the viewfinder and appended an extra shot (79.613 ms) to the stack. **(c, d)**: Insets are shown from the center of the two images. Both insets are relatively free of noise. **(e)**: Just applying the local edit to the statue after capture without taking it into consideration during metering yields much more noticeable noise in the output, compared to (d) where the local edit is accounted for during the metering process. The insets are best viewed on a screen while zoomed in.

Chapter 6

Implementation

In the previous chapters we discussed the algorithmic details of viewfinder editing and appearance-based camera control routines. Combining them together, we now detail the end-to-end implementation of a complete camera framework that realizes WYSIWYG computational photography on two distinct platforms: one is a traditional x86 environment hosted on a laptop with a USB camera. User input is provided with a mouse on a windowed application. The second platform is the ARM-powered tablet with an embedded lens module and a touch-enabled screen.

For ease of demonstration, two separate executables were built in order to showcase appearance-based HDR metering and appearance-based focal stack compositing. Because the lens used with the desktop platform does not support electronic focusing, the focal stack compositing application is implemented only on the tablet platform.

6.1 Hardware

The implementations on the two platforms described in this chapter can in theory cover a large range of hardware. The only requirement is sufficient computational power, a camera with a known response curve, and an input device. For reference, the two platforms used in the development of the system are described in this section.

Much of the real-time result and figures for the desktop platform in this dissertation were gathered from a laptop with Intel Core i7-2760QM CPU (2.4 GHz) and



Figure 6.1: Camera hardware used for the two platforms. **Left:** the NVIDIA Tegra3 developer tablet with an embedded camera module. The screen is simulated. **Right:** a Fujinon varifocal lens attached to the PointGrey USB camera. The assembly connects to a laptop via USB 3.0 protocol.

NVIDIA Quadro 1000M GPU, using a PointGrey Flea3 FL3-U3-32S2C-CS USB camera [93] with a Fujinon varifocal YV4.3X2.8A-2 lens on a CS mount, shown on the right half of Figure 6.1. The camera itself is capable of producing Bayer-mosaicked 1280x960, 12-bpp (bits-per-pixel) images at 60 fps. PointGrey provides drivers and an SDK [91] for the Windows OS, with which the system interfaces. Because the SDK does not support deterministic per-frame alternation of camera parameters, such as exposure and gain, the system pipelines frames in order to obtain this property, at the cost of cutting the frame rate by a third. Therefore, for the HDR application, the system obtains frames at 20 fps. The Fujinon lens has dials for manually adjusting focus, zoom and aperture, but the dials are not motorized and cannot be automated.

On the other hand, the mobile implementation was realised on the NVIDIA Tegra3 developer tablet, shown on the left half of Figure 6.1, featuring a quad-core ARM Cortex-A9 CPU and ULP GeForce GPU with OpenGL ES 2.0 support. It provides two rear-facing cameras, one of which is available via an implementation of the FCam API [49], and can stream Bayer-mosaicked 1296x972, 10-bpp images at 30 fps. While the particular FCam driver implemented on the tablet does nominally support deterministic alternation of capture parameters, it internally throttles the frames to achieve it, and the effective frame rate is 10 fps. The camera also contains a voice-coil-driven lens module that can change focus on demand, useful for the focal stack application. Additional benchmarking was done on a Tegra4 developer tablet.

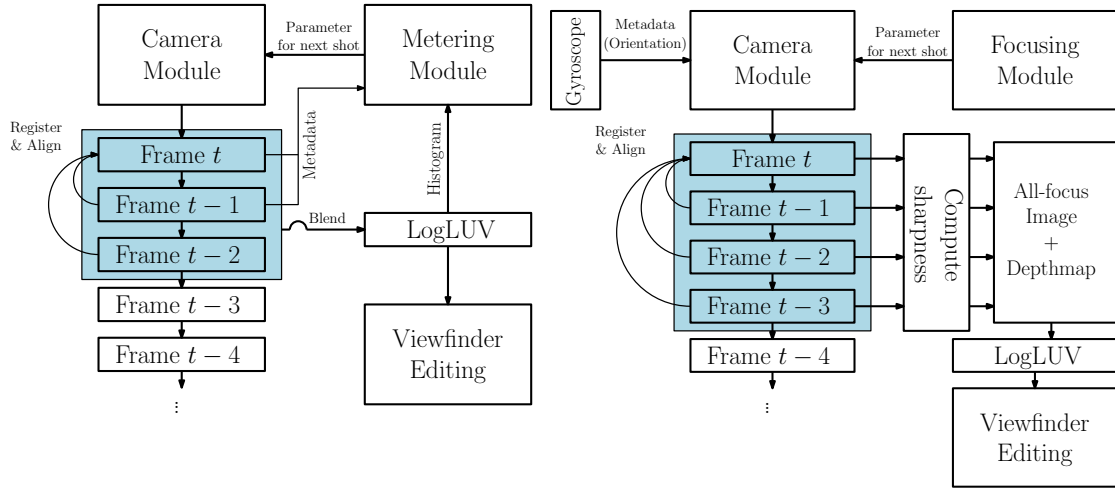


Figure 6.2: A diagram of the camera pipelines for the HDR application and the focal stack application, respectively on the left and the right. See Sections 6.2.1 through 6.2.5 for details. The blue rectangles indicate a moving window for combining the most recent frames.

6.2 Camera Pipeline

Figure 6.2 summarizes the camera pipelines for the HDR application and the focal stack application, respectively. The camera pipeline is responsible for fetching the sensor data and preparing inputs to the viewfinder editing module.

6.2.1 Image Acquisition for HDR Application

The sensor streams raw image data into a stack that caches the most recent N frames, where $N = 3$ for the desktop platform and $N = 2$ for the tablet platform. Both the PointGrey camera and the built-in camera module of the tablet are capable of producing Bayer-mosaic data at bitdepths of 12 and 10, respectively, and at resolutions of 1280x640 and 1292x972, respectively. In both cases, the Bayer-mosaic data is cropped at center to be 1280x640, and then demosaicked by a simple downsampling scheme to produce a three-channel 16-bit RGB images at VGA resolution (640x480).

The metering for the viewfinder stream should ensure that the patches the user operate on are of high fidelity—the edit propagation mechanism works better on a

faithful HDR image than one with saturated, blurry, or underexposed regions. As such, the streaming frames are metered using a greedy, histogram-based HDR metering, rather than applying the appearance-based metering in Chapter 5. The exposure for the next frame is chosen by evaluating how well it covers the log-luminance histogram of the scene, after discounting bins in the histogram that are covered by the latest $N - 1$ frames. These are the frames that are expected to be blended with the target frame, as described in Section 6.2.4. For a relatively static scene and stable camera, this scheme converges quickly to a reasonable solution.

6.2.2 Image Acquisition for Focal Stack Application

The built-in lens module of the tablet is equipped with a voice coil motor that can move the lens in order to focus. The software stack for the tablet implements the FCam API[49], and using this API, a command for moving the lens is issued after receiving each frame. Empirically, the real-time lens position returned by the API was not accurate when the lens is moving, as internally it operates by assuming that the lens travels at a constant speed, whereas the lens typically overshoots its target position and converges to it over time. As such, a delay of 3 frames was necessary after issuing the command to guarantee that the lens position returned was accurate. As such, we issue the command to move the lens at 1/3 of the frame rate. The target focus value is cycled from the set $\{0.0, 4.0, 7.0, 10.0\}$ in diopters. The returned raw image data is demosaicked and cached separately for each target focus value.

6.2.3 Registration and Alignment

In both applications, a moving window over the stream of incoming frames is used to define the set of images to be combined together. As such, rudimentary registration and alignment of the frames are required to avoid ghosting artifacts. On the x86 platform, feature points are found from the source frame in a stratified manner, and they are searched for in the target frame. For the matching keypoints, sparse flow vectors are recovered by running a pyramidal implementation of the Lucas-Kanade feature tracker [69] in the OpenCV library after histogram equalization, and

a homography is computed from the flow vectors. On the tablet platform, each camera frame is tagged with metadata from the inertial measurement unit using the FCam API [49], recording the quaternion of the orientation. A homography is then recovered under the assumption that the center of the rotation coincides with the center of the tablet. Then, the frames in the moving window are warped to align with the most recent frame. The homography between each pair of frames consecutive in time is cached, so that only one registration operation is needed per each incoming frame; the homography between the latest frame and any other previous frames can be computed by accruing the pairwise homographies.

For the focal stack applications, the frames at different focus values may have slightly different levels of zoom [43]. To address this discrepancy, photographs of a calibration setup is obtained at various focus distances with the tablet, and scalars relating the relative zoom levels is computed by extracting SIFT features [88] and solving for a scale transform about the center of the frame. The resulting scale factors are applied to each incoming frame to ensure that the field of view stays constant. Note that this calibration procedure is needed to be performed only once per device. Table 6.1 lists the scale factors recovered for the tablet used. For intermediate focus distances, the scale factor was linearly interpolated in diopter space from the nearby known values.

6.2.4 Image Blending for HDR Application

Once the N frames in the moving window are registered and aligned, the frames are then merged into an HDR radiance map in LogLUV format [23], which accounts for the nonlinearity of human visual system. The blending of the individual frames follows a formula based on the work of Robertson et al. [94]:

$$I_{\text{out}}(x, y) \propto \frac{\sum_{i=1}^N w_i(I_i(x, y)) \cdot \frac{I_i(x, y)}{E_i}}{\sum_{i=1}^N w_i(I_i(x, y))}, \text{ where}$$

$$w_i(p) = E_i^2 \left(\exp \left\{ -16 \cdot \left(p - \frac{1}{2} \right)^2 \right\} - \exp \left\{ -16 \cdot \left(-\frac{1}{2} \right)^2 \right\} \right). \quad (6.1)$$

Diopters	Focus Distance	Scale Factor
0.0	∞	1.00000
1.0	1.00m	0.99664
2.0	50.0cm	0.99343
3.0	33.3cm	0.98996
4.0	25.0cm	0.98676
5.0	20.0cm	0.98348
6.0	16.7cm	0.98079
7.0	14.3cm	0.97739
8.0	12.5cm	0.97442
9.0	11.1cm	0.97117
10.0	10.0cm	0.96845

Table 6.1: Table of scale factors for correcting the change in field of view as a function of focus, for the focal stack application on the NVIDIA Tegra3 tablet.

Here E_i is the exposure time of the i -th frame; $I_i(x, y)$ is the pixel at (x, y) of the i -th frame. The formula in Equation (6.1) penalizes pixel values that deviate from 0.5, and the penalty is offset so that saturated or blank pixels have zero weight. This offset is necessary because there will almost always be saturation in the longest exposure.

6.2.5 Image Blending for Focal Stack Application

In the focal stack application, an all-focus image serves as a proxy for the scene, meaning that image patches for the purpose of high-dimensional lookup are drawn from this all-focus composite. This is in order to ensure that defocus blur does not interfere with the patch-based texture selection algorithm described in Chapter 4. To compute the all-focus image, a sharpness measure M at each pixel is defined to be the sum of the absolute response to the two 3x3 oriented Laplacian kernels.

$$M := \left| \begin{bmatrix} 1 & -2 & 1 \\ 1 & -2 & 1 \\ 1 & -2 & 1 \end{bmatrix} \otimes I \right| + \left| \begin{bmatrix} 1 & 1 & 1 \\ -2 & -2 & -2 \\ 1 & 1 & 1 \end{bmatrix} \otimes I \right| \quad (6.2)$$

The cached frames at each of the four focus distances are uploaded to the GPU,

and a GLSL shader is used to convert them to grayscale and then apply Equation (6.2). A 32-bit RGBA texture is generated by collecting the pixel values from the frame with the highest response at each pixel. The alpha channel is used to hold the focus distance of the corresponding frame, and is mathematically equivalent to a depth map. The CPU converts the returned RGBA texture to LogLUV format, so that the focal stack application can use the same pipeline as the HDR application onward.

6.3 User Interface

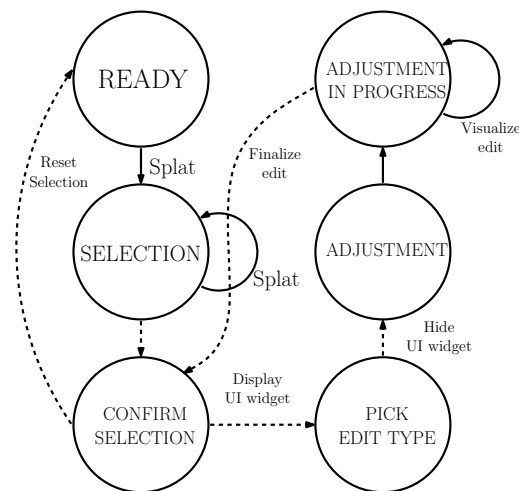


Figure 6.3: A diagram of the state machine for the viewfinder editing applications. Circles represent distinct states in the state machine. A solid arrow is a transition via a touch-down event; a dashed arrow is a transition via a touch-up event.

Both applications employ a touch-based user interface. The user interface internally relies on a simple state machine, as illustrated in Figure 6.3. The transitions between states are triggered by touch-on and touch-off events. On the desktop platform, mouse events are used instead.

Selections are indicated by an alternating pair of translucent diagonal stripes, one white and one black, scrolling over time. Refer to Figure 4.1 for an interaction that traverses across the states of the state machine.

6.3.1 Edit Modalities

Depending on the effect the user wants to achieve on his photographs or videos, the system can enable different kinds of edits. The basic modality supported by the system is a general-purpose one in which the user selects regions and change the brightness, saturation or hue. This modality is demonstrated in [11]. The process of specifying a hue edit is demonstrated in Figure 4.1.

In subsequent work, two additional modalities were implemented for viewfinder editing. In the first, the viewfinder begins desaturated, and the user can selectively restore regions to its original color, enabling the user to easily obtain an image in which everything but a specific object is desaturated. In the second, the user can additively paint desaturated regions with red, green, or blue, allowing her to re-colorize the scene, using the following mathematical formula:

$$\{r_{\text{out}}, g_{\text{out}}, b_{\text{out}}\} := y_{\text{in}} \cdot (\{S_1(\cdot), S_2(\cdot), S_3(\cdot)\} + \{1, 1, 1\}), \quad (6.3)$$

where $S_1(\cdot), S_2(\cdot), S_3(\cdot) \in [-1, 1]$ are the edit map values returned by the viewfinder editing module. As can be deduced from the formula, the user can increase the current RGB values up to a factor of two, or completely mute individual channels. Figure 6.4 shows an example use case.

As mentioned before, a separate application implements the focus editing. In this application, focus edit replaces exposure edit from the basic modality.

All in all, each application instance supports three distinct types of edits. This requires 3 edit masks (S_1, S_2, S_3) in addition to the selection mask (S_0). Hence, the underlying permutohedral lattice represents a function mapping 16-dimensional texture descriptors ($d_p = 16$) to 8 values ($d_v = 8$) corresponding to the four pairs of homogeneous coordinates for the four masks.

6.3.2 Selection Inversion

The system supports a rudimentary form of inverting user selection, which is a common operation in many image editing programs. This feature is accessible from the

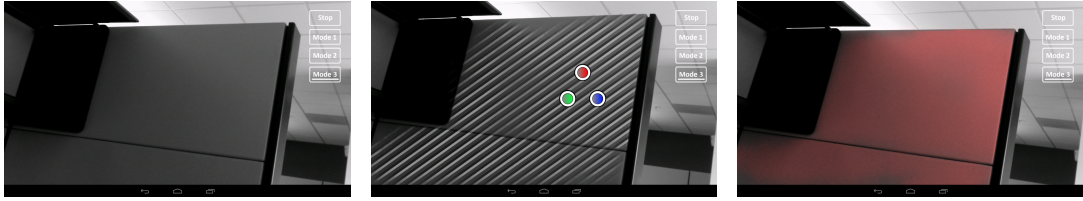


Figure 6.4: A demonstration of colorization of the scene. **Left:** The input scene is shown, desaturated. **Middle:** Selecting a region and confirming the selection brings up a context menu with three icons, each corresponding to one of the primary colors. **Right:** The user has chosen to paint the wall red, and the viewfinder reflects the edit in real time. The screenshots are from the tablet implementation.

UI widget that appears when the user confirms his selection. When this feature is engaged, the selection mask returned by the viewfinder editing module is flipped prior to visualization. In practice, however, simply replacing every value of the selection mask $S_0(x, y)$ with $1.0 - S_0(x, y)$ does not work well, because most selected patches tend to have a selection value well below the maximum. Hence, a mapping that clamps the input from above will create a more visually pleasing result:

$$S_0(x, y) \leftarrow 1.0 - \frac{\max(0.5, S_0(x, y))}{0.5}. \quad (6.4)$$

Folding the edit into the lattice requires a special care, because an edit with inverted selection must apply to any future patches that have not yet been encountered. Hence, this is implemented by amending the default return value of the lattice appropriately, and then applying the uninverted edit to the values for all existing vertices.

6.4 Rendering Pipeline

After the viewfinder editing module processes the incoming image data and user inputs, the results are rendered onto the screen using OpenGL ES shaders. Figure 6.5 summarizes the rendering pipeline for the two applications.

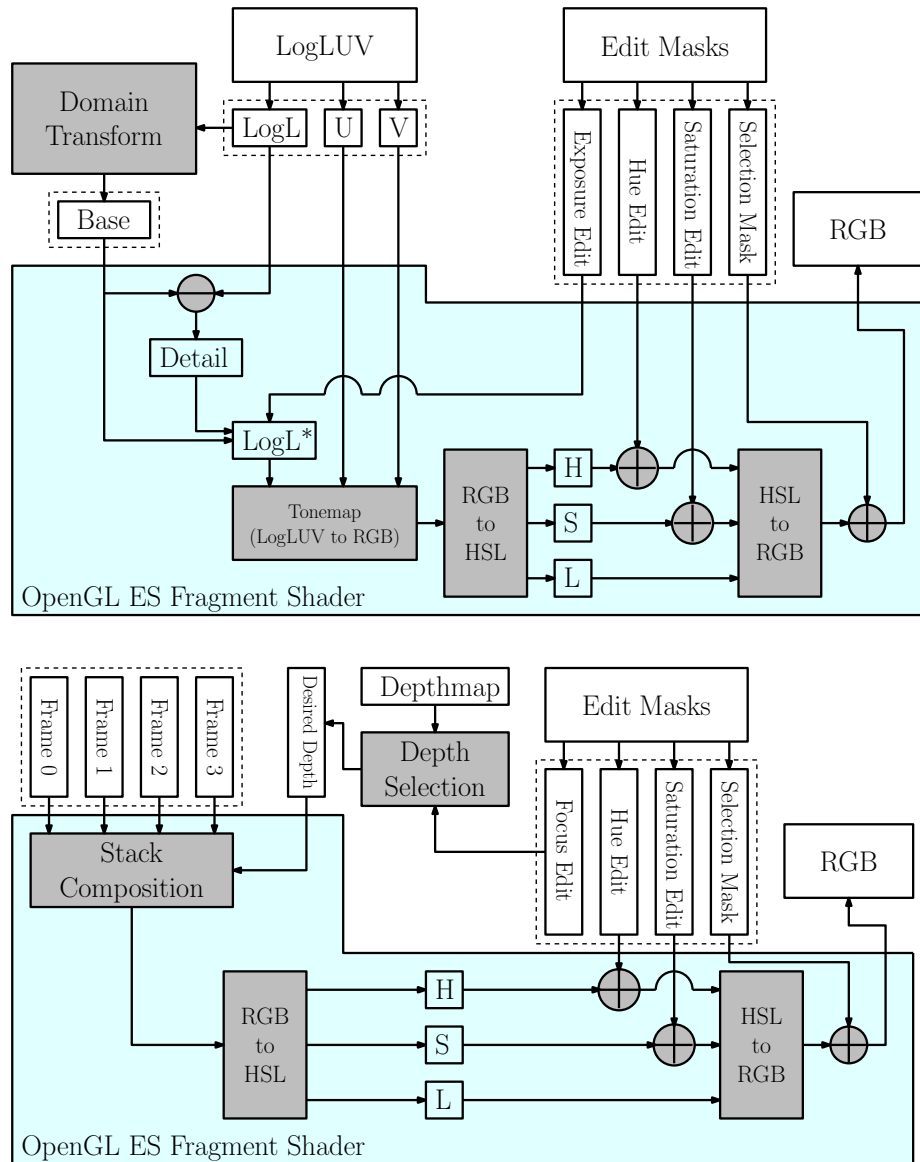


Figure 6.5: A diagram of the rendering pipelines. **Top:** The rendering pipeline for the HDR application. The inputs to the stage are the LogLUV radiance map and the edit masks. **Bottom:** The rendering pipeline for the focal stack application. The inputs to the stage are the 4-frame focal stack described in Section 6.2.2, along with the scene depthmap and the edit masks. In both stages, the output is the RGB pixel values to be displayed. The darker rectangles are modules for algorithmic or arithmetic operations.

6.4.1 Rendering for HDR Application

The rendering pipeline for the HDR application involves tonemapping the LogLUV radiance map to screen-space RGB, then to HSL space [95] to apply the hue and saturation edits, and then finally converting back to RGB for final display. Because both LogLUV radiance map and the edit masks are 32-bit-per-pixel images, uploading them to GPU via OpenGL calls are straightforward.

Exposure edits can be applied by additively shifting the log-luminance value appropriately by the edit mask values. This operation is equivalent to adjusting the “stops” in a photograph and is well-understood. Mathematically, the mask values in $[-1, 1]$ were mapped linearly to a range of twelve stops $([-6, 6])$. However, simply adjusting the log-luminance value tends to compress the contrast. Under the assumption that the edited regions are more salient to the user, local contrast was additionally enhanced by first computing the smooth base layer of the log-luminance channel via domain transform [59] and then interpolating away from this base layer appropriately as shown in Equation (6.5):

$$\begin{aligned} \text{LogL}(x, y) \quad := \quad & [\text{LogL}(x, y) + 6 \cdot S_1(x, y)] \\ & + (\text{LogL}(x, y) - \text{DT}(\text{LogL})(x, y)) \cdot \max(6 \cdot S_1(x, y), 0). \end{aligned} \quad (6.5)$$

Consistent with the assumptions made in the camera control algorithms, the tonemapping stage uses the method proposed by Reinhard et al. [31] to convert the resulting LogLUV value to RGB. Then, hue edits are applied by shifting the present hue value in the $[0, 6]$ range by twice the mask value in $[-1, 1]$, and then taking the remainder modulo 6 so that the final hue value falls in $[0, 6]$. Saturation edits are applied by also shifting the present saturation value in the $[0, 1]$ range by the mask value in $[-1, 1]$ and clamping to $[0, 1]$. Hence, regardless of the current saturation level, the user can completely saturate or desaturate the pixel by specifying the appropriate mask value. For the other modalities discussed earlier, the edits are easily implemented via the GLSL `mix` function.

Finally, the visualization for the selection mask can be added. See Appendix A for the GLSL shader code that accomplishes the rendering.

6.4.2 Rendering for Focal Stack Application

The rendering pipeline for the focal stack application first requires the computation of the desired depth at each pixel, denoted by $\hat{z}(x, y)$ in Chapter 5, from the scene depthmap and the focus edit mask as described earlier. Because focal stack slices are available only at fixed focus distances, the GLSL shader determines the closest focus distance to $\hat{z}(x, y)$ and indexes into the appropriate focal stack slice, generating a focal stack composite. The rest of the pipeline is the same as in the HDR application: hue and saturation edits are applied after temporary conversion to HSL space, and the selection mask is added at the end.

6.5 High-Resolution Capture and Offline Processing

Once the user is satisfied with the composite shown on the viewfinder, he can order an appropriate stack to be captured at higher resolution by pressing an on-screen button. In contrast to the viewfinder stream that does not make use of the appearance-based camera control, the high-resolution capture configures a vector of frames to be acquired in accordance with the optimal parameters. Because the camera drivers in the two implementations do not support dynamic resizing of the camera buffer without restart, the system orders the frames at same size (1296x972 for the tablet, and 1280x960 for the PointGrey camera). However, the demosaicking step forgoes downsampling, and instead demosaics at the same resolution via adaptive color-plane interpolation [96] as implemented in ImageStack utility [97]. Larger image sizes were possible with the given sensors, but changing image resolutions via the appropriate API calls would have required restarting the camera pipeline and incurred delay.

Along with the exposure (or focal) stack, the system exports other metadata, such as a copy of the current instance of the permutohedral lattice and the capture parameters, to be used in further processing. The processing applied to the higher-resolution stack is roughly equivalent to the real-time pipeline for the viewfinder. However, several steps are performed with higher fidelity in order to minimize visual artifacts in the final result. The subsequent sections describe these steps.

6.5.1 Homogeneous Edit Propagation

For the high-resolution LogLUV radiance map, the per-patch descriptor is computed at every possible 16x16 *overlapping* patch instead of at each 8x8 or 16x16 *non-overlapping* patch. While this extension increases the cost of descriptor computation and lattice lookup significantly, it promises a good quality for the resulting edit masks. As a result, the edit masks produced by the lattice are already of the same scale as the input data, and no upsampling is required. However, edge-aware smoothing with domain transform is still necessary in order to ensure that the edit masks are noise-free and conform to the strong edges in the scene.

To enhance the mask quality further, domain transform is performed in homogeneous coordinates: that is, mask values to be smoothed are considered to have confidence associated with them. The confidence values arise naturally from the permutohedral lattice in the form of the vertex weights. The use of the confidence value in the permutohedral lattice to enhance the quality of a smoothed map has been documented in prior work, e.g. in spatiotemporally filtering of depth data [98].

To implement this homogeneous domain transform, all 8 channels returned by the permutohedral lattice are processed independently. However, because the edge weights from the reference image can be shared, one can interleave the 8 channels—each a 32-bit floating-point image—and vectorize the code for 256-bit-wide floating point vectors. Once the channels are smoothed, they can be dehomogenized as before to arrive at four 8-bit edit masks.

Note that homogeneous edit propagation requires filtering 8 channels rather than 4 channels in case of non-homogeneous edit propagation, so the runtime doubles. A CUDA [99] implementation of domain transform would significantly speed up the execution of domain transform, for both online and offline work. While CUDA-capable mobile devices are not yet available at the time of this dissertation, at least one such SoC (system on chip) has been announced [100], and preliminary results indicate that several milliseconds will suffice for processing VGA-resolution data with domain transform.

6.5.2 Manifold-Preserving Edit Propagation

There exists a body of work in high-quality edit propagation for still images and videos. One recent method by Chen et al. [62], called *manifold-preserving edit propagation* (MPEP hereafter), expresses each pixel as a linear combination of nearby pixels, and enforces the resulting linear relationship on the mask values. This structural constraint helps generate high-quality masks. Hence, after edit masks are smoothed with domain transform, they can be further refined via MPEP.

Formally, MPEP requires an input mask, along with a binary map indicating whether the value of the input mask at a given pixel is reliable or not. (For existing applications of MPEP, the binary map is typically provided by user strokes.) To accommodate this input requirement, the edit masks are thresholded by the smoothed confidence values, and are morphologically eroded. The remaining values are considered reliable: the idea is to mark the boundary regions between selected and unselected regions as unknown, so that MPEP algorithm can fill in these regions. Figure 6.6 demonstrates the overall improvement of the edit masks via offline processing, after homogeneous edit propagation and MPEP are applied.



Figure 6.6: Improving edit mask quality offline. **Left:** An unedited viewfinder frame from the tablet implementation, which has already undergone HDR blending and global tonemapping. **Middle:** The viewfinder frame with edits made by the user, displayed live on the tablet screen. In this example, the user has requested a brightening of the unlit side of the building. **Right:** The final output after additional offline processing. The insets in the middle and right images show the computed mask before and after offline refinement. In the refined version, the selected region is more homogeneous and its edges are sharper. While imperfect, the real-time result in the middle is sufficient to give the user a feel for the final photograph.

6.5.3 Focal Stack Compositing

For offline processing of focal stack compositing, the aforementioned extensions are also applied in mask generation. In addition, the depthmap obtained from maximizing the contrast measure is smoothed with a cross-bilateral filter, as done in [43], before being used to compute the index map for the desired depth.

Chapter 7

Results and Discussion

Chapter 6 discussed the implementation of a system that realises viewfinder editing and engages in computational photography. This chapter showcases some of the results created with this system, and reports on the runtime performance. A total of three separate applications are considered: high-dynamic-range (HDR) imaging, focal stack composition, and live video editing. See Chapter 5 for qualitative evaluation of the proposed HDR metering algorithm against prior work. The results contained in this chapter are meant to display the range of pragmatic use cases and the situations in which viewfinder editing is useful.

7.1 Appearance-based HDR Acquisition

In this application, the user makes tonal and color edits on the viewfinder in order to drive HDR acquisition. During the editing session, the user may alter the tone, hue, or saturation locally, or tweak the global tonemapping. Once the shutter is actuated, the camera application orders the image sensor to capture high-resolution images at the exposure(s) chosen by the appearance-based HDR metering. Figure 7.1 shows some empirical results, in addition to those shown in Section 5.3.

7.2 Appearance-based Focal Stack Acquisition

As described in Section 6.2.2, the lens module on the Tegra3 tablet was driven to cycle through multiple focus values in order to provide a “live” focal stack to the camera application, which then combines the stack into a composite and enables the user to perform edits on the said composite. Figure 7.2 and Figure 7.3 demonstrate two user interactions and the resulting composites.

It was found that for most practical scenes, local edits were crucial for producing a pronounced reduction in depth of field. The chief reason is that multiple objects at a meter away or farther (0.0 – 1.0 diopters) will have approximately equal-sized blur, since the circle of confusion grows with the depth disparity in diopters. Hence, global edits cannot cause them to have distinctive blurs.

Lastly, producing a faithful all-focus image while the camera is quickly moving is difficult because of registration artifacts: a strong edge that is incorrectly registered will create ghosting. This can be quite jarring to the user. Turning off image blending when the camera is moving fast mitigated this problem, since the user cannot distinguish between the ghosting artifacts and the motion blur of the device itself.

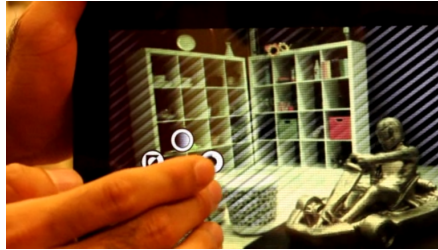
7.3 Live Video Editing

The viewfinder editing algorithm is fast enough to be used in editing live videos as they are filmed. This obviates the need to revisit the videos afterward and decode them for processing. Figure 7.4 demonstrates an example. A full video of the interaction, along with additional results, are available on the project website [101].

Many existing edit propagation algorithms operating on offline videos require keyframes containing user strokes every 20 frames or so [102], whereas the proposed method does not. The need for keyframe is tied to the fact that the spatial locations of the edits are used in these algorithms: if objects are displaced significantly over time, then the spatiotemporal distance in patch space becomes large enough that edits do not properly stick. The proposed method, in contrast, places no emphasis on spatial or temporal distance, obtaining robustness against scene and camera motion.



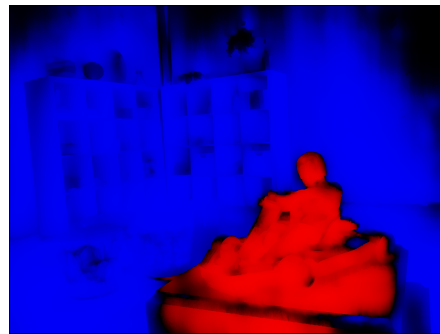
Figure 7.1: Additional results of HDR capture with viewfinder editing, demonstrating various scenarios in which viewfinder editing is useful. **Left:** Results without local edits. **Right:** Results with local edits. The images were locally brightened and/or darkened, or had saturation adjusted, after which appropriate exposure stacks were acquired.



(a) A still shot of the user interaction



(b) Results without local edit



(c) Computed mask for focus edit



(d) Results with local edit



(e) Reference photograph

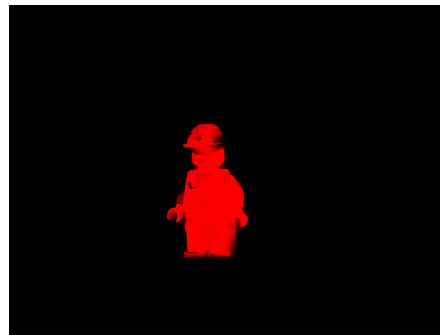
Figure 7.2: Focus edit for the `racer` scene. **(a)**: A still shot of the user interaction on the tablet. **(b)**: The result that was presented to the user initially. **(c)**: The edit mask computed for focus edits. Red corresponds to sharpening, and blue corresponds to blurring. **(d)**: The result after the user performs local edits. The focal stack was appropriately indexed to generate this composite. **(e)**: A reference photograph taken by the tablet, focused on the foreground. The edited result in (d) simulates a far wider aperture than what is physically possible on the device.



(a) A still shot of the user interaction



(b) Results without local edit



(c) Computed mask for focus edit



(d) Results with local edit



(e) Reference photograph

Figure 7.3: Focus edit for the lego scene. **(a)**: A still shot of the user interaction on the tablet. **(b)**: The result that was presented to the user initially. **(c)**: The edit mask computed for focus edits. Red corresponds to sharpening, and blue corresponds to blurring. No further blur was requested by the user. **(d)**: The result after the user performs local edits. The focal stack was appropriately indexed to generate this composite. **(e)**: A reference photograph taken by the tablet, focused on the foreground. The edited result in (d) simulates a non-physical depth of field, allowing the user greater artistic freedom than conventional devices.

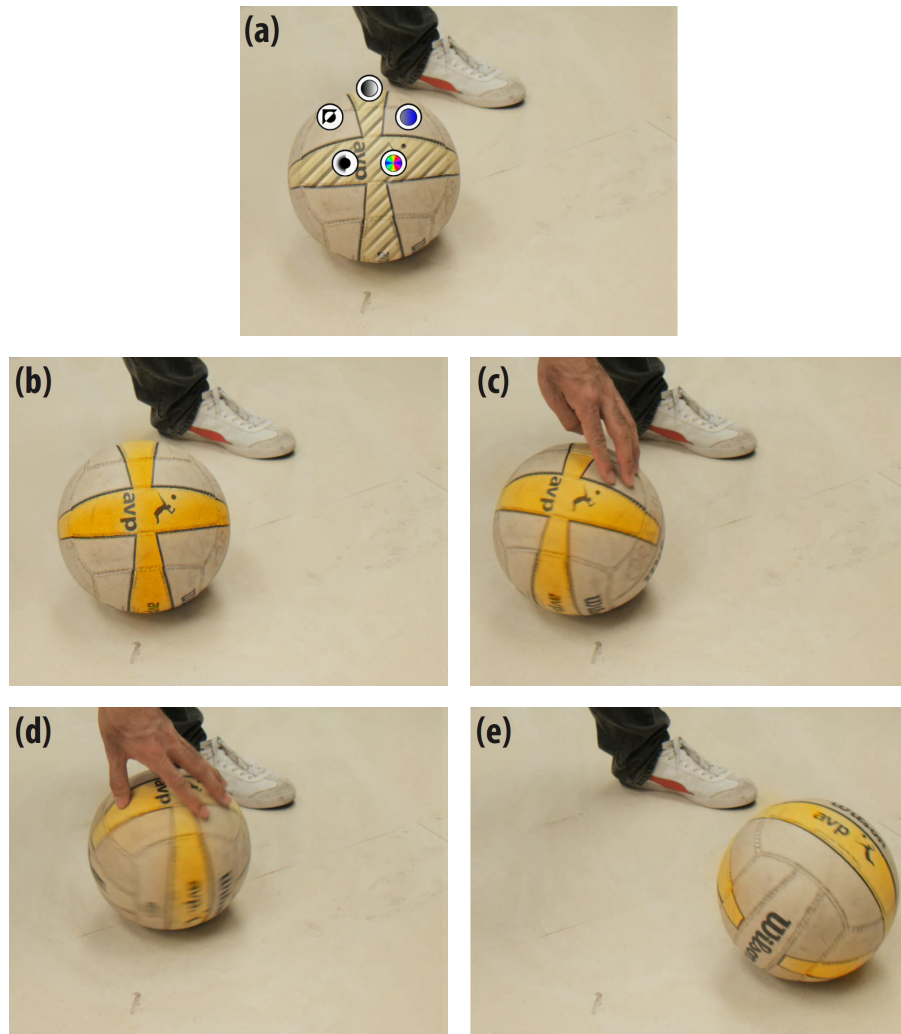


Figure 7.4: Edit propagation on a live video. **(a)**: In this example featuring a faded volleyball, the user marks a region as selected, corresponding to the ball's stripes as shown. **(b)**: The user then applies an edit to accentuate the color, restoring the unfaded look. **(c-e)**: This edit can be seen persisting through frames #150, #300, and #420 of a 30-fps video sequence, despite considerable motion of the ball between frames and the resulting motion blur. See the supplementary video on the project page [101] for the entire sequence.

7.4 Performance

The implementation of viewfinder editing has undergone several iterations of engineering throughout the lifetime of the project, with each iteration improving the runtime of a module via parallelism, vectorization and algorithmic improvements. Figures from various stages of engineering are reported.

Task	Platform	
	Laptop (Intel x86)	Tablet (ARM, Tegra3)
HDR blending	12.18 ms	48.46 ms
Patch descriptor computation	1.49 ms	18.37 ms
Lattice lookup	3.34 ms	12.92 ms
Domain transform	8.77 ms	24.64 ms
Appearance-based metering	1.79 ms	10.35 ms
GPU Processing	1.70 ms	44.38 ms
Total	29.27 ms	159.12 ms
RAW capture rate	20.0 FPS	10.0 FPS
Application FPS	34.2 FPS	6.3 FPS

Table 7.1: Runtime on both laptop and tablet implementations for the HDR application. Timing of the viewfinder editing framework at VGA resolution are reported. The GPU processing consists of LogLUV decoding, tonemapping, applying edits, and UI rendering. Note, however, that the implementation used is dated, without all the optimizations. See Table 7.2 for more recent numbers.

Table 7.1 summarizes the breakdown of the runtime of the HDR application on both the laptop and tablet platform, at the time of the submission of the project to the SIGGRAPH Asia 2013 conference [11]. There are a few interesting observations to be made from the breakdown. First, real-time HDR blending was quite costly, and was the most time-consuming step for either platform. This step involves registering individual LDR frames and blending them into a single HDR frame—see Section 6.2 for details. Warping multiple frames via homography is expensive on the CPU. While warping texture on the GPU is faster, the cost of moving the image data back and forth negated any performance gain. Also, handling HDR radiance map required high-precision arithmetic in either full 32-bit floating point or 32-bit fixed point, which increased the final blending cost.

The second observation is that the GPU portion of the application (color space decoding, tonemapping, rendering) was exceedingly large for the tablet platform. This was largely due to driver issues that had trouble channeling data between the CPU and the GPU, and was improved in the subsequent generation of the hardware running Tegra4, the successor to the Tegra3 SoC.

Viewfinder editing was in fact ported to a Tegra4 tablet, and the runtime breakdown is presented in Table 7.2, representing the most recent progress. For this measurement, HDR blending and registration were turned off, allowing the camera to run at its native speed of 30 fps. Compared to the Tegra3 implementation used in Table 7.1, the lattice lookup implements the linear-time quantization described in Section 3.3, and domain transform was simplified and optimized further. Multi-scale approach described in Section 4.4.2 was added. Most importantly, the GPU overhead largely disappeared, owing to improved drivers and firmware. In the end, the system boasts an impressive frame rate of 44.7 fps.

Task	Runtime on Tablet (ARM, Tegra4)
Patch descriptor computation	8.64 ms
Lattice lookup	3.71 ms
Multi-scale composition	1.16 ms
Spatiotemporal filtering	1.52 ms
Overhead	3.45 ms
Other	3.87 ms
Total	22.35 ms
RAW Capture rate	30.0 FPS
Application FPS	44.7 FPS

Table 7.2: Breakdown of the runtime on a Tegra4 tablet. A non-stack-based LDR viewfinder stream was processed. “Overhead” corresponds to the cost of multi-threading, operating mutexes, function and application overheads. “Other” corresponds to the cost of rendering and the time taken by the camera thread and other system processes running on the background.

7.4.1 Discussion

Neither platform—the PointGrey SDK nor the FCam implementation on the Tegra tablets—on which viewfinder editing was implemented supported per-frame alteration of capture parameters. To alter camera parameters while streaming viewfinder content, the camera module had to drop two frames. As a result, the rate at which the camera application fetched image frames was cut in a third. In practice, this meant that the PointGrey camera capable of running at 60 fps was actually providing an LDR frame at 20 fps, so an entirely new HDR frame with an unoverlapping sliding window was being obtained at 6.66 fps. The ability to alter capture parameters per-frame without loss of frame rate is critical in stack photography applications. A fully programmable camera module (like the Frankencamera [49]) would be necessary to realise a full frame rate on today’s mobile devices.

Lens shading, if left uncorrected, can affect the performance of texture matching. The vignetting that results from lens shading is typically low-frequency and may be inconspicuous to human eyes, but it can alter the appearance of the same texture depending on its location relative to the center of the frame.

Programming mobile devices remains a challenge, as the available computes are distributed among many subsystems, such as the CPU, the vector co-processor unit (ARM NEON), the GPU and the DSP. The programmability of these subsystems also varies wildly. Implementing viewfinder editing and appearance-based camera control encompassed writing C++ code, NEON intrinsics, handwritten assembly, and GLSL scripts. Various multi-threaded and single-threaded schedules were tested manually. Even then, the overall performance varied greatly, depending on the flaws and idiosyncrasies of various drivers and firmware controlling the hardware—the narrative surrounding Table 7.1 and Table 7.2 comes to mind. There are current efforts to facilitate harnessing computes from such a heterogeneous environment [103], which would significantly reduce the burden on the researchers for implementing a compute-intensive algorithm on mobile platforms.

Chapter 8

Conclusions

This dissertation realises a full-fledged camera application running on a mobile platform, introducing the notion of viewfinder editing for the first time, and demonstrating its usefulness in both regular photography and stack-based computational photography. This system is made possible by simultaneous advances in several areas, including fast high-dimensional filtering, appearance-based image editing, and multi-exposure camera control. Each of these advances is validated by both theoretical analyses and empirical experiments, showing improvement over the state of the art in runtime complexity, runtime benchmarks, and/or the final image quality.

Each of these advances provides value in its own right: appearance-based camera control is still applicable without any local editing. Appearance-based edit propagation can be applied to still images for still image manipulation. Fast high-dimensional filtering has many well-documented applications in other fields, such as physics, finance, and computer vision.

All of these advances were combined to enable viewfinder editing, as described in this dissertation. Viewfinder editing can be a powerful tool for enhancing both the photography experience and the photographs themselves. This paradigm of enabling new ways of content manipulation for photographers should lead to other applications and contribute to the continued cultural success of photography.

In the remainder of this dissertation, the limitations of the proposed system are considered, and avenues for future work are discussed.

8.1 Usability Issues

The system was designed primarily to serve casual users. These are users who, subconsciously or not, derive more utility from the act of taking photographs than the resulting photographs themselves, and as such, they will find the pre-capture interaction novel, valuable and engaging.

On the other hand, professional photographers will always be willing to take the image manipulation offline in order to produce the best result. It will still be beneficial to augment the current system to allow post-capture edits, e.g. enable a more precise specification of the edits or corrections to the online result, so that the final output can be refined. In any case, professionals can still benefit from appearance-based camera control, reducing the time required to capture an exposure or focus stack.

There were following concerns raised on the user interaction for the system by observers and reviewers of the work.

Screen Brightness In a bright, sunny environment, the angle of the screen surface with respect to the sun can sometimes cause glare, making it difficult to properly see the content on the screen. There is also the issue of physiological adaptation of the human visual system: if the user is staring into the sun and his eyes adapt to the bright sky, the screen becomes comparatively dark. This is a fundamental problem with electronic screens in outdoor environment, but OLED technology keeps improving, nonetheless, towards eliminating this problem. As it stands, the screen is viewed unhampered in an indoor environment, or in outdoors on an overcast day. The metering, editing and capture algorithms still work as intended in all situations.

Ergonomics It was found that the form factor of a tablet supported more precise edits than that of a smartphone, simply by the virtue of having a larger screen—7-10 inches in diagonal, compared to 3-5 inches for a typical smartphone. At the same time, however, the larger mass can put more strain on the user, since users typically support the device with only one hand, making strokes on the screen with his other. Other types of input modalities for mobile devices, such as voice or gaze tracking, are on the rise currently, and may be able to help ameliorate the ergonomics issue in the future.

Input Precision Specifying edits via sparse strokes on a touch interface has been commonly employed in many recent papers. However, it is inherently less precise than other input modalities available on desktside environment, like a trackball or a mouse. A stylus may be able to provide a more precise input. There is also recent research on tolerating user mistakes on sparse strokes [104], which would enhance the user interaction considerably when applied to the current system.

8.2 Algorithmic Issues

The limitations of mobile platforms render the underlying algorithms less precise and less robust than their desktop implementations. Some limitations of the underlying algorithms and future work are discussed.

Latency Almost all digital imaging systems exhibit some level of latency: the viewfinder screen lags behind the physical world by some time. The current system is pipelined so that the viewfinder imagery has no visible latency beyond that of a typical digital camera, but the edit masks are behind by 20-120 ms, depending on the platform. In practice, this delay did not hinder the user experience substantially. Further optimization should reduce it to a more reasonable amount on existing hardware. In comparison, some commercial products, such as Sony DSC-QX10 lens [105] that connects to a smartphone via wireless, exhibit latency in the order of a second for the viewfinder content.

Frame rate Most digital camera viewfinders run at 30 fps, which the current implementation of the system can keep up with at VGA resolution. Higher frame rates, such as 60 fps or even 120 fps, are becoming more popular, however. For the live video editing application, processing 1080p video at 120 fps represents a 27-fold increase in the number of pixels to process in a given unit of time, and will require significant increase in the capabilities of the hardware, along with improvements in the algorithm. It is worthwhile to note that CUDA-capable mobile devices have been announced [100]. The permutohedral lattice [8] has been implemented on CUDA, and so has domain transform. Moving these compute-intensive operations fully onto the GPU may already go a long way.

Handling texture The proposed algorithm in Chapter 4 is not as robust as some existing algorithms that propagate edits onto offline datasets. This difference is a design choice: it arose primarily as a result of the substantially less available compute in mobile devices. Also, the viewfinder stream tends to exhibit noise and other deficiencies, compared to the clean images and videos typically used to evaluate edit propagation algorithms.

The biggest difficulty that was encountered in manipulating live viewfinder content was handling objects with diverse texture. Objects with relatively simple texture, such as sky, foliage, building façades, faces, are easy to select and manipulate. On the other hand, cabinets containing various objects, heavily checkered fabric and other heavily textured surfaces were troublesome. Multi-scale processing (Section 4.4.2) was introduced in the second iteration of the system in response to these objects. However, because the texture classification occurs on a subsampled grid (Section 4.4.1) and the ensuing spatial smoothing (Section 4.4.3) is only edge-aware instead of “texture-aware,” the online result has difficulty generating a clean mask that preserves strong edges between texture boundaries. A truly “texture-aware” spatial smoothing algorithm in place of domain transform would solve this issue. In addition, increase in the computing power of mobile device should close the gap between offline and online performances in the future.

Illumination Because the proposed edit-propagation method is based on the local appearance, illumination currently plays a larger role in determining whether an object is fully selected than it ought to. This can cause problems in two ways: first, for a surface with sufficiently specular component in its reflectance, the angle at which the camera points at the surface will alter its brightness. Second, a large surface may be unevenly lit, and the difference in the brightness levels of two regions at either end of the surface can be large enough that stroking over one end does not select the other end of the surface.

Because spatial propagation of edits can overcome the conservative selection caused by illumination issues, identifying which image patches become selected via domain transform and then splatting them into the lattice as a feedback system may help ameliorate this issue.

Scale and focus If the object moves towards or away from the sensor, the size of the object as seen in the viewfinder will change, enlarging or shrinking the relevant textures. For complicated textures, the resized patches may sufficiently be different from the originals. Also, the degree of sharpness of its texture may also change, depending on the extent of the motion. In practice, the algorithm can handle minor changes in scale and sharpness, but is not entirely robust against them.

8.3 Extensions

Viewfinder editing and other modules developed in this dissertation can be extended or improved in various ways, beyond what has been discussed already in Sections 8.1 and 8.2. A few examples are studied here.

Spatial locality Because the selection is based on appearance, distinct objects that share the same appearance cannot be disambiguated currently. If the user wants to select one out of multiple identical objects, some notion of image- or world-space coordinates will be necessary, which can be provided by 2D- or 3D-tracking. Once image- or world-coordinates that compensate for camera motion are available, they can be included in the texture descriptor in order to localize edits, as done already in many edit-propagation works that deal with a single photograph. Handling scene motion, however, is an area of open research in computer vision.

Geometric edits All edits currently supported by the system are photometric, per-pixel edits, with the exception of focus edits. One can consider geometric edits, such as distortions, object removal, duplication, in the style of Barnes et al. [106] as an example. However, such algorithms are very expensive even with a powerful desktop, considerably more so than appearance-based edit-propagation algorithms.

A plausible approach may be to obtain a planar (or otherwise simple) proxy for the target, perform geometric edits on the proxy, and display the edited proxy on top of the viewfinder content in the proper orientation and scale to compensate for any change in the camera pose.

Common textures Some textures are prone to being manipulated by users across many cases, such as the sky, foliage and faces. These textures can be learned

a priori and found via face detection [50] and other specialized searches that may be more efficient than the general texture-matching in the permutohedral lattice.

Presets In the current implementation, the current state of the permutohedral lattice can be saved along with the viewfinder content and be loaded again, so that the same type of edit can be applied to multiple photos or scenes. This practice can be generalized as *presets*, enabling the users to plan and save edits prior to a shoot, or save useful edits after a shoot for future events. For instance, it would be interesting to attend a sporting event armed with a preset in which the appearance of the ball is enhanced or edited in some way, as in Figure 7.4. However, for presets to be truly useful, the algorithm will have to be improved in order to handle subtle changes in illumination, scene scale and focus settings.

Power efficiency This thesis does not address one important aspect of mobile application design, which is power efficiency. Mobile devices have limited battery capacity. Future work will benefit from carefully incorporating the power requirements into design decisions.

8.4 Closing Remarks

In the strictest sense of the words, the notion of *viewfinder editing* is not new. Photographers have always had the ability to alter what flows through their viewfinders by physically moving their devices, and turning the knobs to manually adjust exposure, gain and focus. More recently, photographers have been able to tap on the electronic viewfinder to drive touch-to-exposure or touch-to-focus algorithms. This dissertation further extends the photographers' ability to direct their content creation process by allowing local edits prior to capture, and demonstrates its benefits for both the user and the camera application. Especially in this day and age when uploading and sharing photos immediately after capture is popular (and even being the default option for many camera applications), enabling such creative control prior to capture is an exciting avenue of work.

As with any endeavor to develop a full end-to-end system, this dissertation has encountered challenges in several places, with some fruitful results. Both individual

parts and the system as a whole—along with insights and observations mined from the engineering effort—will be useful to researchers in the field of computational photography, and to others who are tackling problems requiring fast high-dimensional lookup, better camera control algorithms, et cetera. Finally, we hope that this dissertation inspires and enables future efforts to reinvent the experience of taking photos in the 21st century.

Appendix A

GLSL Shader for HDR Application

```
precision mediump float;
uniform sampler2D uLogLUV;
uniform sampler2D uMasks;
uniform sampler2D uBlurredL;

uniform float uFormulaK;
uniform float uReinhardScalar;
uniform float uTargetExposure;
uniform float uFractionalTime;
uniform vec3 uColorScale;
uniform int uEditInProgress;

varying vec2 vTexCoord;
mat3 XLZ_TO_RGB = mat3( 2.5651, -1.0217, 0.0753,
                       -1.1665, 1.9777, -0.2543,
                       -0.3986, 0.0439, 1.1892 );

void main(void)
{
    vec4 LogLUV = texture2D(uLogLUV, vTexCoord);
    vec2 LogL = LogLUV.xy;
    float U = LogLUV.z;
    float V = LogLUV.w;
    vec4 mask = texture2D(uMasks, vTexCoord);
    vec2 LogLBlurred = texture2D(uBlurredL, vTexCoord).rg;

    const float LOG2 = 0.6931472;
```

```

// Construct the linear luminance.
float logEdit = ( ( mask.y - 0.5 ) * 2.0 ) * 6.0;
LogL.x = dot(LogL.xy, vec2(255.0, 255.0 * 256.0));
LogLBlurred.x = dot(LogLBlurred.xy, vec2(255.0, 255.0 * 256.0));
LogL.x = (LogL.x - LogLBlurred.x) * (1.0 + max(logEdit, 0.0)) + LogLBlurred.x;
float L = exp( ( LogL.x / 256.0 - 64.0 + logEdit * 0.5 ) * LOG2 ) * ←
    uTargetExposure * uFormulaK * uReinhardScalar;

// Convert to RGB after Reinhard tonemapping.
U = U * 0.62;
V = V * 0.62;
float S = 1.0 / (U * 6.0 - V * 16.0 + 12.0);
U = U * S * 9.0;
V = V * S * 4.0;
vec3 XLZ = vec3(U, V, 1.0-U-V) * (L / V);
mat3 colorScaleMatrix = mat3(uColorScale.r, 0.0, 0.0, 0.0, uColorScale.g, 0.0, ←
    0.0, 0.0, uColorScale.b);
vec3 rgb_linear = XLZ_T0_RGB * colorScaleMatrix * XLZ;
float maxRGB = max(max(rgb_linear.r, rgb_linear.g), rgb_linear.b);
float Lnormalized = (L / (L + 65535.0));
vec3 normalizer = mix(vec3(L,L,L), rgb_linear, Lnormalized * Lnormalized);
rgb_linear = rgb_linear / ( normalizer + 65535.0);
vec4 rgb = vec4(rgb_linear, 1.0);

// Convert from RGB to HSL
vec3 hsl;
bvec3 comp1 = greaterThanEqual(rgb.xyz, rgb.yzx);
bvec3 comp2 = greaterThanEqual(rgb.xyz, rgb.zxy);
vec3 max_finder = vec3(comp1) * vec3(comp2);
max_finder.y = max_finder.y * (1.0 - max_finder.x);
max_finder.z = max_finder.z * (1.0 - max_finder.x - max_finder.y);

float max_rgb = dot(max_finder, rgb.xyz);
float min_rgb = min(min(rgb.x, rgb.y), rgb.z);
float delta = max_rgb - min_rgb;

hsl.z = (max_rgb + min_rgb) * 0.5;
hsl.y = delta / (1.0001 - abs(2.0 * hsl.z - 1.0));
hsl.x = dot(max_finder, vec3(0.0, 2.0, 4.0)) + (dot(max_finder.zxy, rgb.xyz) - ←
    dot(max_finder.yzx, rgb.xyz)) / (delta + 0.0001);
hsl.x = fract(hsl.x / 6.0) * 6.0;
mask.z = sqrt(abs(mask.z - 0.5) * 2.0) * 0.5 * sign(mask.z - 0.5) + 0.5;

// Apply hue edit.
hsl.x = fract((hsl.x + (mask.z - 0.5) * 4.0) / 6.0) * 6.0;

// Apply saturation edit.

```

```

    if (hsl.y > 0.0)
    {
        hsl.y = clamp(hsl.y + (mask.w - 0.5) * 2.0, 0.0, 1.0);
    }

    // Convert from HSL back to RGB.
    float c = (1.0 - abs(2.0 * hsl.z - 1.0)) * hsl.y;
    float q = fract(hsl.x * 0.5);
    float x = c * (1.0 - abs(q * 2.0 - 1.0));
    rgb.x = c;
    rgb.y = x;
    rgb.z = 0.0;

    if (q >= 0.5)
    {
        rgb.xyz = rgb.xzy;
    }
    if (hsl.x >= 1.0 && hsl.x < 3.0)
    {
        rgb.xyz = rgb.zxy;
    }
    else if (hsl.x >= 3.0 && hsl.x < 5.0)
    {
        rgb.xyz = rgb.yzx;
    }
    rgb.xyz = rgb.xyz + vec3(hsl.z - 0.5 * c);

    // Draw the selection mask.
    if (uEditInProgress == 0)
    {
        mask.x = max(0.0, mask.x - 0.1) * (1.0 / 0.9);

        float r = fract((vTexCoord.x * 1.333 + vTexCoord.y) * 20.0 + uFractionalTime ←
            * 0.001);
        if (r < 0.3)
        {
            rgb = mix(rgb, vec4(1.0,1.0,1.0,1.0), mask.x * 0.5);
        }
        else if (r > 0.7)
        {
            rgb = mix(rgb, vec4(0.0,0.0,0.0,1.0), mask.x * 0.5);
        }
    }
    gl_FragColor = rgb;
}

```


Appendix B

Fast Quantization Source Code

The following code snippet can be dropped into the publicly available implementation of the permutohedral lattice from Adams et al. [8], replacing the method with the same signature. Refer to Algorithm 3.4 for more concise pseudocode, and Section 3.3 for commentary.

```
template<int kd, int vd> void PermutohedralLattice<kd, vd>::slice(const float* ←  
    position, float* const value)  
{  
    float elevated_tmp[kd + 1];  
    float barycentric_tmp[kd + 2];  
    short greedy_tmp[kd + 2];  
    int rank_tmp[kd + 1];  
    short key_tmp[kd + 2];  
  
    mapToHyperplane(position, elevated_tmp);  
    // Try to find a nearby remainder=0 point.  
    const float scale = 1.0f / (kd + 1);  
    int sum = 0;  
    for (int i = 0; i <= kd; i++)  
    {  
        greedy_tmp[i] = ((short)floorf(elevated_tmp[i] * scale + 0.5f)) * (kd + 1);  
        sum += greedy_tmp[i];  
    }  
    sum /= kd + 1;  
  
    // Build the bins.  
    char bins[2 * kd + 2];
```

```

char invBin[kd + 1];
float maxBin[2 * kd + 2];
float minBin[2 * kd + 2];
float binSum[2 * kd + 2];
for (int i = 0; i < 2 * kd + 2; i++)
{
    bins[i] = 0;
    binSum[i] = 0;
}
for (int i = 0; i <= kd; i++)
{
    const float delta = elevated_tmp[i] - greedy_tmp[i];
    const int index = (int)delta + (kd + 1);
    invBin[i] = index;
    char count = bins[index]++;
    if (count == 0 || delta > maxBin[index])
        maxBin[index] = delta;
    if (count == 0 || delta < minBin[index])
        minBin[index] = delta;
    binSum[index] += delta;
}

// Iterate through bins, checking the ones that satisfy the lemma.
int lastNonemptyBin = -1;
char cellCount = 0;
float cellSum = 0;
char bestK = 0;
int bestCellIndex = 0;
float bestObjective = 0;
for (int i = 0; i < 2 * kd + 2; i++)
{
    float score = (kd + 1) * (cellCount - sum) - (cellCount - sum) * (cellCount ←
        - sum) + 2 * cellSum;
    if (score < bestObjective)
    {
        bestObjective = score;
        bestCellIndex = lastNonemptyBin;
        bestK = cellCount;
    }
    cellCount += bins[i];
    cellSum += binSum[i];
    lastNonemptyBin = i;
}

int sumsum = sum - bestK;
bestK = (bestK - sum + (kd + 1)) % (kd + 1);
sumsum += bestK;

```

```
int offset = sumsum < 0 ? (kd + 1) : (sumsum == 0 ? 0 : (-1 - kd));
for (int i = 0; i <= kd; i++)
{
    key_tmp[i] = greedy_tmp[i] + bestK - ((invBin[i] <= bestCellIndex) ? (kd + ←
        1) : 0) + offset;
}

// Retrieve pointer to the value at this vertex.
Entry<kd, vd> *entry = hashTable.lookup(key_tmp, false);
if (entry)
{
    for (int i = 0; i < vd; i++)
        value[i] = entry->values[i];
}
else
{
    for (int i = 0; i < vd; i++)
        value[i] = default_values[i];
}
}
```

Bibliography

- [1] Camera & Imaging Products Association. Production, shipment of digital still cameras. http://www.cipa.jp/english/data/pdf/d_2012_e.pdf, 2012. Accessed: 8/31/2013.
- [2] The Economist Newspaper. Camera-phones: Dotty but dashing. *The Economist*, 2010-04-10, 2010.
- [3] Facebook, Inc. Capturing growth: Photo apps and open graph. <http://developers.facebook.com/blog/post/2012/07/17/capturing-growth--photo-apps-and-open-graph/>, 2012. Accessed: 8/27/2013.
- [4] Eino-Ville Talvala. *The Frankencamera: Building a Programmable Camera for Computational Photography*. PhD thesis, Stanford University, 2011.
- [5] Paul Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. *ACM Transactions on Graphics*, pages 369–378, 1997.
- [6] Anat Levin, Robert Fergus, Frédo Durand, and William T. Freeman. Image and depth from a conventional camera with a coded aperture. *ACM Transactions on Graphics*, 26, July 2007.
- [7] Ren Ng. Fourier slice photography. *ACM Transactions on Graphics*, 24:735–744, July 2005.

- [8] Andrew Adams, Jongmin Baek, and Abraham Davis. High-dimensional filtering with the permutohedral lattice. *Computer Graphics Forum (Proceedings of Eurographics 2010)*, 29(2):753–762, 2010.
- [9] Jongmin Baek, Andrew Adams, and Jennifer Dolson. Lattice-based high-dimensional gaussian filtering and the permutohedral lattice. *Journal of Mathematical Imaging and Vision*, 46(2):211–237, June 2013.
- [10] Andrew Adams. *High-Dimensional Gaussian Filtering for Computational Photography*. PhD thesis, Stanford University, 2011.
- [11] Jongmin Baek, Dawid Pająk, Kihwan Kim, Kari Pulli, and Marc Levoy. Wysiwyg computational photography via viewfinder editing. *ACM Transactions on Graphics*, 32(5), December 2013.
- [12] Stephen M. Smith and J. M. Brady. SUSAN: A new approach to low level image processing. *International Journal of Computer Vision*, 23:45–78, 1997.
- [13] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Proc. International Conference on Computer Vision, 1998*, pages 836–846, 1998.
- [14] Johannes Kopf, Michael F. Cohen, Dani Lischinski, and Matt Uyttendaele. Joint bilateral upsampling. *ACM Transactions on Graphics*, 26, July 2007.
- [15] Antoni Buades, Bartomeu Coll, and Jean-Michael Morel. A non-local algorithm for image denoising. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition, 2005*, volume 2, pages 60–65, 2005.
- [16] Leslie F. Greengard and John A. Strain. The fast gauss transform. *SIAM Journal on Scientific and Statistical Computing*, 12:79–94, 1991.
- [17] Sylvain Paris and Frédo Durand. A fast approximation of the bilateral filter using a signal processing approach. In *Proc. European Conference on Computer Vision, 2006*, pages 568–580, 2006.

- [18] Changjiang Yang, Ramani Duraiswami, Nail A. Gumerov, and Larry Davis. Improved fast gauss transform and efficient kernel density estimation. In *Proc. International Conference on Computer Vision, 2003*, volume 1, pages 664–671, 2003.
- [19] Andrew Adams, Natasha Gelfand, Jennifer Dolson, and Marc Levoy. Gaussian kd-trees for fast high-dimensional filtering. *ACM Transactions on Graphics*, 28(3):21:1–21:12, August 2009.
- [20] Barbara London, John Upton, and Jim Stone. *Photography*. Pearson, 2010.
- [21] Canon Inc. *EF Lens Work III: The Eyes of EOS*. Canon Inc., 2009.
- [22] Doug A. Kerr. Principle of the split image focusing aid and the phase comparison autofocus detector in single lens reflex cameras. http://doug.kerr.home.att.net/pumpkin/Split_Prism.pdf, 2005. Accessed: 8/17/2013.
- [23] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 2005.
- [24] Miguel Granados, Boris Ajudin, Michael Wand, Christian Theobalt, Hans-Peter Seidel, and Hendrik P. A. Lensch. Optimal HDR reconstruction with linear digital cameras. *IEEE Computer Vision and Pattern Recognition*, pages 215–222, 2010.
- [25] Andrew Adams, Natasha Gelfand, and Kari Pulli. Viewfinder alignment. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27(2):597–606, 2007.
- [26] Suyu You, Ulrich Neumann, and Ronald Azuma. Hybrid inertial and vision tracking for augmented reality registration. In *Proceedings of Virtual Reality, 1999*, pages 260–267, March 1999.
- [27] Orazio Gallo, Natasha Gelfand, Wei-Chao Chen, Marius Tico, and Kari Pulli. Artifact-free high dynamic range imaging. *International Conference on Computational Photography*, pages 1–7, 2009.

- [28] Jun Hu, Orazio Gallo, and Kari Pulli. Exposure stacks of live scenes with hand-held cameras. In *European Conference on Computer Vision*, pages 499–512. Springer, 2012.
- [29] Samuel W. Hasinoff, Frédo Durand, and William T. Freeman. Noise-optimal capture for high dynamic range photography. In *Computer Vision and Pattern Recognition*. IEEE, 2010.
- [30] Orazio Gallo, Marius Tico, Roberto Manduchi, Natasha Gelfand, and Kari Pulli. Metering for exposure stacks. *Computer Graphics Forum (Proceedings of Eurographics 2012)*, 31(2):479–488, 2012.
- [31] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21:267–276, 2002.
- [32] Martin Čadík, Michael Wimmer, Laszlo Neumann, and Alessandro Artusi. Evaluation of hdr tone mapping methods using essential perceptual attributes. *Computers & Graphics*, 32(20):330–349, 2008.
- [33] Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. Interactive digital photomontage. *ACM Transactions on Graphics*, 23(3):294–302, August 2004.
- [34] Eugne Hecht. *Optics*. Addison-Wesley, 1987.
- [35] Jongmin Baek. Transfer efficiency and depth invariance in computational cameras. *International Conference on Computational Photography*, pages 1–8, March 2010.
- [36] Hajime Nagahara, Sujit Kuthirummal, Changyin Zhou, and Shree K. Nayar. Flexible depth of field photography. In *European Conference on Computer Vision*, 2008.
- [37] Ashok Veeraraghavan, Ramesh Raskar, Amit Agrawal, Ankit Mohan, and Jack Tumblin. Dappled photography: Mask enhanced cameras for heterodyned light

- fields and coded aperture refocusing. *ACM Transactions on Graphics*, 26, July 2007.
- [38] Edward R. Dowski and W. Thomas Cathey. Extended depth of field through wave-front coding. *Applied Optics*, 34(11), 1995.
- [39] Anat Levin, Samuel W. Hasinoff, Paul Green, Frédo Durand, and William T. Freeman. 4D frequency analysis of computational cameras for depth of field extension. *ACM Transactions on Graphics*, 28(3):97:1–97:14, July 2009.
- [40] Anat Levin, Yair Weiss, Frédo Durand, and William T. Freeman. Understanding and evaluating blind deconvolution algorithms. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition, 2009*, pages 1964–1971, 2009.
- [41] Samuel W. Hasinoff and Kiriakos N. Kutulakos. Light-efficient photography. In *European Conference on Computer Vision*. Springer, 2008.
- [42] Daniel Vaquero, Natasha Gelfand, Marius Tico, Kari Pulli, and Matthew Turk. Generalized autofocus. In *Workshop on Applications of Computer Vision*. IEEE, 2011.
- [43] David E. Jacobs, Jongmin Baek, and Marc Levoy. Focal stack compositing for depth of field control. Technical Report 1, Stanford Computer Graphics Laboratory, 10 2012.
- [44] Soonmin Bae and Frédo Durand. Defocus magnification. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26(3):571–579, 2007.
- [45] Joseph W. Goodman. *Introduction to Fourier Optics*. Roberts & Company, 2005.
- [46] Soonmin Bae, Aseem Agarwala, and Frédo Durand. Computational rephotography. *ACM Transactions on Graphics*, 29(3):24:1–24:15, July 2010.
- [47] Ligang Liu, Renjie Chen, Lior Wolf, and Daniel Cohen-Or. Optimizing photo composition. *Computer Graphics Forum (Proceedings of Eurographics 2010)*, 29(2):469–478, 2010.

- [48] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings of SIGGRAPH '93*, pages 279–288, 1993.
- [49] Andrew B. Adams, E. Talvala, Sung Hee Park, David E. Jacobs, Boris Adjin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Henrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The Frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics*, 29(4):29:1–29:12, July 2010.
- [50] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition, 2001*, volume 2, pages 511–518, December 2001.
- [51] Anat Levin, Dani Lischinski, and Yair Weiss. Colorization using optimization. *ACM Transactions on Graphics*, 23(3):689–694, 2004.
- [52] Dani Lischinski, Zeev Farbman, Matt Uyttendaele, and Richard Szeliski. Interactive local adjustment of tonal values. *ACM Transactions on Graphics*, 25(3):646–653, July 2006.
- [53] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18:253–259, July 1984.
- [54] Alvy Ray Smith and James F. Blinn. Blue screen matting. In *Proceedings of SIGGRAPH '96*, pages 259–268, August 1996.
- [55] Mark A. Ruzon and Carlo Tomasi. Alpha estimation in natural images. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition, 2000*, pages 18–25, June 2000.
- [56] Yung-Yu Chuang, Brian Curless, David Salesin, and Richard Szeliski. A Bayesian approach to digital matting. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition, 2001*, volume 2, pages 264–271, December 2001.

- [57] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Analysis Machine Intelligence*, 12:629–639, July 1990.
- [58] Raanan Fattal. Edge-avoiding wavelets and their applications. *ACM Transactions on Graphics*, 28(3):1–10, August 2009.
- [59] Eduardo S. L. Gastal and Manuel M. Oliveira. Domain transform for edge-aware image and video processing. *ACM Transactions on Graphics*, 30(4):69:1–69:12, July 2011.
- [60] Xiaobo An and Fabio Pellacini. Approp: all-pairs appearance-space edit propagation. *ACM Transactions on Graphics*, 27(3):40:1–40:9, August 2008.
- [61] Zeev Farbman, Raanan Fattal, and Dani Lischinski. Diffusion maps for edge-aware image editing. *ACM Transactions on Graphics*, 29(6):145:1–145:10, December 2010.
- [62] Xiaowu Chen, Dongqing Zou, Qinqing Zhao, and Ping Tan. Manifold preserving edit propagation. *ACM Transactions on Graphics*, 31(6):132:1–132:7, November 2012.
- [63] Yuanzhen Li, Edward H. Adelson, and Aseem Agarwala. ScribbleBoost: Adding classification to edge-aware interpolation of local image and video adjustments. *Computer Graphics Forum*, 27(4), 2008.
- [64] Kun Xu, Yong Li, Tao Ju, Shi-Min Hu, and Tian-Qiang Liu. Efficient affinity-based edit propagation using k-d tree. *ACM Transactions on Graphics*, 28(5):118:1–118:6, December 2009.
- [65] Xiaohui Bie, Haoda Huang, and Wencheng Wang. Real time edit propagation by efficient sampling. *Computer Graphics Forum*, 30(7), 2011.
- [66] Yong Li, Tao Ju, and Shi-Min Hu. Instant propagation of sparse edits on images and videos. *Computer Graphics Forum*, 29(7), 2010.

- [67] Chia-Kai Liang, Wei-Chao Chen, and Natasha Gelfand. Touchtone: Interactive local image adjustment using point-and-swipe. *Computer Graphics Forum (Proceedings of Eurographics 2010)*, 29(2), 2010.
- [68] Nik Software. Snapseed | snap it. tweak it. love it. share it. <http://www.snapseed.com>, 2012. Accessed: 12/11/2012.
- [69] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. *International Joint Conference on Artificial Intelligence*, pages 674–679, 1981.
- [70] Carlo Tomasi and Takeo Kanade. Detection and tracking of point features. Technical Report 132, Carnegie Mellon University, 4 1991.
- [71] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.
- [72] Harold S. M. Coxeter. Extreme forms. *Canadian Journal of Mathematics*, 3:391–441, 1951.
- [73] Claude A. Rogers. *Packing and covering*. Cambridge University Press, 1964.
- [74] Ram P. Bambah and Neil J. A. Sloane. On a problem of Ryskov concerning lattice coverings. *Acta Arithmetica*, 42:107–109, 1982.
- [75] Boris N. Delaunay and Sergei S. Ryskov. Solution of the problem of least dense lattice covering of a four-dimensional space by equal spheres. *Soviet Math. Dokl.*, 4:1014–1016, 1963.
- [76] Sergei S. Ryskov and Evgenii P. Baranovskii. Solution of the problem of least dense lattice covering of five-dimensional space by equal spheres. *Soviet Math. Dokl.*, 16:586–590, 1975.
- [77] John H. Conway and Neil J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer, 1998.

- [78] Daniel P. Petersen and David Middleton. Sampling and reconstruction of wave-number-limited functions in N-dimensional Euclidean spaces. *Information and Control*, 5:279–323, 1962.
- [79] John H. Conway and Neil J. A. Sloane. Fast quantizing and decoding and algorithms for lattice quantizers and codes. *IEEE Transactions on Information Theory*, 28(2):227–232, March 1982.
- [80] John H. Conway and Neil J. A. Sloane. A fast encoding method for lattice codes and quantizers. *IEEE Transactions on Information Theory*, 29:820–824, 1983.
- [81] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, , and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [82] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time edge-aware image processing with the bilateral grid. *ACM Transactions on Graphics*, 26, July 2007.
- [83] Sylvain Paris. Edge-preserving smoothing and mean-shift segmentation of video streams. In *European Conference on Computer Vision*. Springer, 2008.
- [84] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, 2005.
- [85] Kostandin Dabov, Alessandro Foi, and Karen Egiazarian. Image restoration by sparse 3D transform-domain collaborative filtering. In *Proceedings of SPIE Electronic Imaging '08*, January 2008.
- [86] Ian T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.
- [87] William T. Freeman and E. H. Adelson. The design and use of steerable filters. *IEEE Trans. Pattern Analysis Machine Intelligence*, 13:891–906, 1991.
- [88] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.

- [89] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1615–1630, October 2005.
- [90] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.
- [91] Point Grey Research, Inc. Point Grey FlyCapture SDK. <http://http://ww2.ptgrey.com/sdk/flycap>, August 2013.
- [92] Google, Inc. Camera | android developers. <http://developer.android.com/guide/topics/media/camera.html>, 2013. Accessed: 9/2/2013.
- [93] Point Grey Research, Inc. Point Grey Flea3 USB 3.0. <http://ww2.ptgrey.com/USB3/Flea3>, August 2013.
- [94] Mark A. Robertson, Sean Borman, and Robert L. Stevenson. Dynamic range improvement through multiple exposures. In *Int. Conf. on Image Processing*. IEEE, 1999.
- [95] George H. Joblove and Donald Greenberg. Color spaces for computer graphics. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12:20–25, August 1978.
- [96] John F. Hamilton Jr. and James E. Adams. Adaptive color plane interpolation in single color electronic camera. U.S. Patent 5,629,734, May 1997.
- [97] Andrew Adams. ImageStack - A command line stack calculator for images. <http://code.google.com/p/imagestack/>. Accessed: 8/20/2013.
- [98] Jennifer Dolson, Jongmin Baek, Christian Plagemann, and Sebastian Thrun. Upsampling range data in dynamic environments. *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pages 1141–1148, 2010.
- [99] Ian Buck. GPU computing: Programming a massively parallel processor. In *International Symposium on Code Generation and Optimization*, page 17, March 2007.

- [100] NVIDIA Corporation. NVIDIA brings Kepler, world's most advanced graphics architecture, to mobile devices. <http://developer.nvidia.com/content/nvidia-brings-kepler-world's-most-advanced-graphics-architecture-mobile-devices>, August 2013. Accessed: 8/20/2013.
- [101] Jongmin Baek. WYSIWYG computational photography via viewfinder editing - SIGGRAPH Asia 2013. <http://graphics.stanford.edu/papers/wysiwyg/>, October 2013.
- [102] Manuel Lang, Oliver Wang, Tunc Aydin, Aljoscha Smolic, and Markus Gross. Practical temporal consistency for image-based graphics applications. *ACM Transactions on Graphics*, 31(4), July 2012.
- [103] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4), July 2012.
- [104] Kartic Subr, Sylvain Paris, Cyril Soler, and Jan Kautz. Accurate binary image selection from inaccurate user input. *Computer Graphics Forum (Proceedings of Eurographics 2013)*, 32(2):479–488, 2013.
- [105] Sony Communications. New sony QX series “lens-style cameras” redefine the mobile photography experience. <http://blog.sony.com/press/new-sony-qx-series-lens-style-cameras-redefine-the-mobile-photography-experience/>, September 2013. Accessed: 10/23/2013.
- [106] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B. Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics*, 28(3), August 2009.