HIROSHIMA UNIVERSITY
GRADUATE SCHOOL OF ENGINEERING
DEPARTMENT OF INFORMATION ENGINEERING

MARCOS PAULO BERTELI SLOMP

GPU-accelerated Algorithms for Photorealistic, Non-Photorealistic and Perceptually-Based Rendering

(GPUを用いたフォトリアリスティック，ノンフォトリアリスティック，そして視覚特性を考慮したレンダリング手法の高速化に関する研究)

A thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Engineering

Prof. Dr. Kazufumi Kaneda
Supervisor

Higashihiroshima, March 2012

“Education never ends, Watson.
It is a series of lessons with the greatest for the last.”
— SHERLOCK HOLMES
in “The Adventure of the Red Circle” by Sir Arthur Conan Doyle

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 7

LIST OF FIGURES 8

LIST OF TABLES 15

EXECUTIVE SUMMARY 16

ABSTRACT 17

1 INTRODUCTION 18

1.1 Retrospective of the Modern Graphics Hardware 18

1.2 Thesis Contributions 27

1.3 Structure of the Thesis 28

2 PHOTOREALISTIC REAL-TIME RENDERING OF SPHERICAL RAINDROPS WITH HIERARCHICAL REFLECTIVE AND REFRACTIVE MAPS 29

2.1 Abstract 29

2.2 Introduction 30

2.3 Related Work 32

2.4 Proposed Method 34

2.4.1 Assumptions and Observations 35

2.4.2 Preprocessing 36

2.4.3 Run Time 37

2.5 Implementation Details . . . 39
2.6 Results . . . 41
2.7 Limitations . . . 44
2.8 Conclusion . . . 44
2.9 Future Work . . . 44
2.10 Related Publications . . . 46
2.A Appendix A: Computer Graphics Forum (CGF) 2011 Cover Image Contest Finalist . . . 47

3 GPU-BASED SOFTASSIGN FOR MAXIMIZING IMAGE UTILIZATION IN PHOTOMOSAICS . . . 49
3.1 Abstract . . . 49
3.2 Introduction . . . 50
3.3 Maximizing Image Utilization on Photomosaics . . . 51
3.4 Photomosaic Optimization Strategies . . . 53
3.4.1 The Distance Matrix . . . 53
3.4.2 The Greedy-Based Search . . . 54
3.4.3 Simulated Annealing . . . 54
3.4.4 SoftAssign . . . 56
3.5 Initial Results . . . 57
3.6 SoftAssign Implementation . . . 59
3.6.1 Addressing Precision Issues . . . 60
3.6.2 Parallelism in SoftAssign . . . 61
3.7 SoftAssign on GPU . . . 61
3.7.1 Mapping SoftAssign on the Graphics Pipeline . . . 62
3.7.2 SoftAssign Implementation on GPU . . . 62
3.7.3 Implementation Details . . . 64
3.8 Extended Results . . . 65
3.9 Conclusion . . . 67
3.10 Future work . . . 68
3.11 Related Publications . . . 71

3.A Appendix A: Photomosaic Optimization Based on Ant Colony Optimization . . . 72

4 EFFICIENT SUMMED-AREA TABLE AND PREFIX-SUM GENERATION ON THE GPU . . . 76

4.1 Abstract . . . 76

4.2 Introduction . . . 77

4.3 Fast Summed-Area Table Generation on the GPU . . . 79

4.4 Parallel Scan Generation on the GPU with Recursive Doubling . . . 81

4.5 Fast Parallel Scan Generation on the GPU Based on Balanced-Trees . . . 83

4.6 A Note on Precision Issues with Summed-Area Tables . . . 85

4.7 Results . . . 86

4.8 Closing Comments: Limitations, Conclusion and Future Work . . . 90

4.9 Related Publications . . . 91

4.A Appendix A: Complexity Analysis . . . 92

4.A.1 Recursive Doubling . . . 92

4.A.2 Balanced Tree . . . 94

4.A.3 SAT Generation Algorithms . . . 95

5 SCREEN-SPACE AMBIENT OCCLUSION THROUGH SUMMED AREA TABLES . . . 97

5.1 Abstract . . . 97

5.2 Introduction . . . 98

5.3 Related Work . . . 100

5.3.1 Ambient Occlusion (AO) . . . 100

5.3.2 Screen-Space Ambient Occlusion . . . 102

5.4 Technique Explained . . . 103

5.4.1 The minimalist approach . . . 105

5.4.2 Normal-guided sampling . . . 106

5.4.3 Depth refinement . . . 107

5.5 Implementation Details . . . 109

5.6 Results . . . 110

5.7 Limitations . . . 114
5.8 Conclusion . . . 114
5.9 Related Publications . . . 115

6 CONCLUSION . . . 116

COMPLETE LIST OF PUBLICATIONS . . . 118

TECHNICAL ACKNOWLEDGMENTS . . . 120

A FAST AND ROBUST SPATIALLY-VARYING MESOPIC VISION SIMULATION FOR TONE MAPPING OPERATORS . . . 121
A.1 Abstract . . . 121
A.2 Introduction . . . 122
A.3 Related Work . . . 124
A.3.1 Real-time Local Tone-mapping . . . 125
A.3.2 Mesopic Vision Simulation . . . 126
A.3.3 Review of the Photographic Tone Reproduction Operator . . . 127
A.4 Photographic Local Tone Mapping with Summed-Area Tables . . . 130
A.4.1 Generating the Scaled Luminance Image on the GPU . . . 130
A.5 Mesopic Vision Simulation . . . 132
A.5.1 Equivalent Lightness Curve . . . 132
A.5.2 Opponent-Color Theory and the $L^*a^*b^*$ Color Space . . . 133
A.5.3 Overview of the Mesopic Vision Reproduction Operator . . . 134
A.5.4 Spatially-Uniform Mesopic Vision Reproduction Operator . . . 135
A.5.5 Spatially-Varying Mesopic Vision Reproduction Operator . . . 136
A.6 Results . . . 139
A.7 Limitations and Future Work . . . 144
A.8 Conclusion . . . 145
A.9 Related Publications . . . 146

REFERENCES . . . 147

LIST OF ABBREVIATIONS AND ACRONYMS

cd/m^2	Candela per Square Metre, photometric measurement of luminance
CTM	AMD/ATI Close To Metal
CUDA	NVIDIA Compute Unified Device Architecture
GLSL	OpenGL Shading Language
GPGPU	General-Purpose Computation on GPU
GPU	Graphics Processor Unit
HDR	High Dynamic Range
HLSL	Microsoft's Direct3D High Level Shading Language
HVS	Human Visual System
IBL	Image-Based Lighting
IDE	Integrated Development Environment
LDR	Low Dynamic Range
NPR	Non-Photorealistic Rendering
SAT	Summed-Area Table
SDK	Software Development Kit
TMO	Tone-Mapping Operator

LIST OF FIGURES

1.1 Performance increase between NVIDIA GPU and Intel CPU architectures over time. The retail price at launch time of latest hardware is shown in parenthesis. 24

2.1 Left) Closeup of a ray-traced raindrop. Center) Same raindrop rendered using the texture-based approximation described in this paper. Right) A rainy scene rendered using the proposed technique. Results were generated using high dynamic range environment maps together with tone-mapping. Left and center images are nearly identical but the proposed technique can render one million raindrops at 1024x1024 screen resolution at 140FPS on a GeForce GTX 280 (performance measurement includes animation and tone-mapping). . . . 30

2.2 Raindrop comparison: Left) A raindrop rendered using low dynamic range environment map. Center) Raindrop rendered using a high dynamic range environment map. Right) Photographs of two real-world water droplets taken outside by using a water dropper and a regular digital camera. Note that despite the crude equipment used the droplets closely resemble spherical shapes. When using HDR lighting the results are more similar to the real-world. Reflections in the upper portion of the raindrops which do not exist in the LDR case are clearly visible. 31

2.3 Raindrop distribution and the corresponding geometrical shape: in a typical rainy scene most raindrops are smaller than 1mm in radius thus being nearly spherical in shape. 33

2.4 Mip-map Generation: Positioning raindrops so that their surface area is maximized in the projection plane is important. Given an aperture angle, the distance (hypotenuse) is obtained by simple trigonometry, using the radius of the sphere (opposite side) and the observation that aperture vector tangencies the sphere forming a right angle with the radius vector at that point (marked in red). Below are the corresponding renderings of the illustration in the upper portion. As can be seen the distance from the raindrop to the camera drastically changes the raindrop's appearance and must be taken into account for accurate rendering. 35

2.5 Fake color representation of the hierarchical maps generated in the preprocessing step. The transparency mask is the multi-sampled alpha mask discussed in the end of Section 2.4.2. Although the refraction layers seem similar in this fake-color representation, their variance have a strong impact in the resulting raindrop's appearance. . 38

2.6 Run-time shader tree as described in Section 2.4.3. 40

2.7 Additional results rendered using the proposed technique in different HDR environment maps, viewing conditions, and number of raindrops. Results account for tone-mapping. 43

3.1 A 20×20 photomosaic assembled from 1500 images assigned through one of the proposed algorithms (simulated annealing). The input image is miniaturized at the left for reference. The usage of available images was maximized so that no image appears repeatedly in the resulting mosaic. 50

3.2 The distance matrix stores the color similarities between patches and tiles. 54

3.3 Greedy-Based Search: The corresponding patches of a multi-assigned tile are sequentially replaced by unassigned tiles until that multi-assigned tile becomes uniquely assigned. The reassignment process is based on the best-match criteria, i.e., lowest distance (highest visual similarity). 55

3.4 Summary of experimental results: the upper left image is the reference image; the lower left one is a 20x20 photomosaic generated from 1500 tiles (similar to Case 3) through a simple *best-match* (minimal cost) algorithm which allows tile repetition (only 82 tiles were selected; final absolute cost of 79989). Note how such recurring tiles tend to bias the photomosaic, leading to weak aesthetics. The remaining rows corresponds to one of the three studied configurations enumerated in the Experimental Results Section, and each column corresponds to one of the three tile-repetition-free discussed algorithms. 57

3.5 Parallel Reduction instance that performs the total sum of each row of a given table. The result is held in a column-vector whose elements correspond to the sum of all elements of that corresponding row in the original input table. At each step, two consecutive elements are gathered and accumulated together from the partial results of the previous step. 63

3.6 Performance chart comparing the CPU and GPU implementations of SoftAssign. For clarity, the chart was plotted with time being expressed in a base-10 logarithmic scale. The samples in the horizontal axis are spaced linearly according to the total number of elements in the distance matrix, that is, $rows \times columns$. Note that the GPU performance is far better than the CPU: the 1s barrier is never reached by the GPU while in the CPU cases this barrier is crossed at very small distance matrix dimensions. 66

3.7 A kimono photomosaic made of 30x30 patches selected from a thematic Japanese tile set of 4096 images. The input image is shown miniaturized in the left. 68

3.8 A fish image photomosaic made of 30x30 patches selected from a thematic fish tile set of 8192 images. The input image is shown miniaturized in the left. 69

3.9 A chair image photomosaic made of 30x30 patches selected from a thematic chair tile set of 8192 images. The input image is shown miniaturized in the left. 69

3.10 A butterfly image photomosaic made of 30x30 patches selected from a thematic butterfly tile set of 4096 images. The input image is shown miniaturized in the left. 70

4.1 A small 6×5 table (a) and its corresponding SAT (b). The highlighted blue cell on the SAT is the sum of all the highlighted blue cells in the input table (all cells up and to the left, inclusive). In order to filter the 8 elements marked in green in the input table (c), only the four red elements A, B, C and D need to be fetched from the SAT (d), a fact that holds true for arbitrary sizes. The filtering result is given by: $\frac{A-B-C+D}{area} = \frac{54-6-25+4}{4*2} = \frac{27}{8} = 3.375$. 77

4.2 Example of SAT-based filtering for regions that extend outside the boundaries of the table. The areas hatched in light green in (b) and (d) should be discarded from the original areas R in (a) and (c), respectively, in order to evaluate the correct filtering result; the correct area R' is shown in dark green in (b) and (d). 78

4.3 An example of a prescan (b) and a scan (c), under the addition operation, on some input array (a). 79

4.4 SAT generation as a set of 1D array scans. Each row of the input table (a) is submitted to an 1D array scan, leading to a partial SAT (b). Each column of this partial SAT (c) is then submitted to another 1D array scan, resulting in the complete SAT (d). 80

4.5 An walk-through on scan generation using the recursive-doubling approach. In each step, every element updates itself by accumulating two values: itself (blue arrows) and a neighbor to the left (orange arrows). At each step the offset to the neighbor doubles. In (a) all elements are updated at each step. It can be observed in (a) that the first few elements of each step do not need to be updated again since they would accumulate with a *ghost* neighbor (zero-valued elements in violet) which would not modify their values. The image in (b) exploits this fact and exclude such elements from the update (heading elements marked in green), only updating those who need to be updated (marked in blue). 81

4.6 An walk-through on a GPU-based scan generation using the recursive-doubling approach. At each pass the input and the auxiliary arrays are swapped. An important procedure in this *ping-pong* approach is to keep the values of both arrays consistent for subsequent passes by copying a number of elements from the current read-only array to the write-only array, as illustrated in (a) where these copies are marked in dark-yellow. Without these copies, subsequent passes would operate with (and output) inconsistent values, as shown in (b), with red values indicating the inconsistencies. 82

4.7 An walk-through on scan generation using the balanced-tree approach. The reduction stage is shown in the left and the expansion stage in the right. 83

4.8 Suggested layout for the auxiliary GPU memory during the scan. Simultaneous read/write from/to a same buffer never happens. In order to expand aux.#4 the expanded parent buffer aux.#3 and the reduced *sibling* buffer aux.#2 must be accessed. Similarly, expanding aux.#5 needs access to aux.#4 and aux.#1. The final expansion uses aux.#5 and the input buffer itself. The extra memory amounts to about three times the size of the input. 84

4.9 Summed-Area Tables of luminance images are inherently monotonic and prone to unpleasing noise artifacts (a) if the quantities involved have a wide dynamic range (b). Making the SAT non-monotonic by first subtracting the average luminance mitigates such artifacts. 86

4.10 Summed-Area Table generation time using recursive-doubling. 87

4.11 Summed-Area Table generation time using balanced-trees. 87

4.12 Relative speed-up between balanced-trees and recursive-doubling for SAT generation based on the best (fastest k) times recorded for each algorithm for each image size, according to the performance results of Figures 4.10 and 4.11. 88

4.13 Speedup between different GPUs running the balanced tree method ($k = 4$). 88

5.1 Comparison between sharp shadows generated from standard Shadow Maps and soft shadows produced by Ambient Occlusion. 99

5.2 Ambient Occlusion techniques have been successfully employed in a wide range of fields, from data visualization to professional games and film production. 99

5.3 Ambient occlusion overview (a): the accessibility of a given point x with normal vector $\vec{n}$ is the ratio between all non-blocked directions (hatched in red) over the total number of directions of a visibility hemisphere Ω around $\vec{n}$ (gray). The image in (b) depicts a visibility hemisphere Ω centered around a normal vector $\vec{n}$ in 3D. 101

5.4 An example of a typical G-buffer produced by a deferred shading pipeline, consisting of the depth-buffer, a normal buffer and a material/reflectance buffer. 103

5.5 Visual comparison of the three methods for SAT-based screen-space ambient occlusion introduced in this Section. 104

5.6 The minimalist approach: the amount of occlusion over a pixel (in green) can be approximated through the difference of the average depth around that pixel (in magenta) and the pixel depth itself. 105

5.7 A square-shaped area (in magenta) is selected around each inspected pixel (in green); the size of the area depends on the current pixel's depth. This area is then filtered using the SAT and the difference between the filtered depth and the actual depth of the inspected pixel is used to determine the amount of occlusion. 105

5.8 Surface orientation can redirect filtering onto a more influential regions. The initial filtering subspace (in magenta) is translated along the direction of the projected normal (orange arrow), resulting in a new filtering area (dark blue). 106

5.9 The normal-buffer is sampled along with the depth-buffer to determine an offset direction and length (light-green arrow). This offset then relocates the initial sampling area (in magenta) to another location (in light-blue) where the average depth is computed using the associated Summed-Area Table. 107

5.10 Depth refinement can avoid pitfalls such as the one produced when averaged depths cancel each other influences. Once filtered, if the absolute difference between the filtered depth and the pixel central to the region is below some threshold τ, then a depth trap may be in place and subdivision is triggered, seeking to identify more potentially occluding subregions within it. 107

5.11 Once the initial sampling region is offset by the normal vector, subdivision can take place. Each subdivision step splits the parent area into 4 equally-sized subregions, analogous to a quad-tree subdivision. 108

5.12 Efficient SAT region sub-filtering. Starting from some large region R, the four cells required to filter such area are retrieved. From this point, in order to filter the subregion R_1 one would only need to fetch 3 cells, since the upper-leftmost is already known from its parent region. As for R_2, only one cell must be fetched, since the other cells are already known from its parent and sibling R_1; similarly for R_3. The last subregion R_4 does not require any additional look-ups. 109

5.13 Comparative results between traditional SSAO [MITTRING (2007)] (left, upper row) and horizon-based SSAO [BAVOIL; SAINZ; DIMITROV (2008)] (left, bottom row) before and after the low-pass filter is applied. The right-most image is the result obtained with the proposed SAT-based SSAO technique (normal-guided with depth refinement). The SAT-based approach intrinsically performs such filtering along with accessibility calculation, without incurring into any additional performance penalty. 111

5.14 Additional examples of the proposed SAT-based SSAO using the depth refinement strategy along with the normal-guided one (Subsection 5.4.3). 112

5.15 The top image represents Phong shading only [PHONG (1975)], without ambient occlusion information. The subsequent images modulate shading with the minimalist, normal-guided and normal-guided with depth refinement approaches, respectively. Note the increasing introduction of darkening features around the mouth, crest and tail of the dragon, progressively enhancing the overall depth perception. (Minor contrast adjustments were applied to these images in order to highlight shadowing details in printed media.) 113

5.16 Performance scalability under increasing image resolutions. Performance measurements account for all stages, from deferred buffer generation, depth-buffer down-sampling, accessibility calculations to shading at last. 114

A.1 A comparison between the global and the local variants of the photographic operator of Reinhard et al. (2002). Local operators are capable of preserving more contrastive detail. 122

A.2 An evening urban scene (a) without mesopic vision simulation and (b) with the mesopic vision strategy later described in Section A.5. As it can be seen the sky changes from purple to a more blueish tone, while distant artificial lights shift from red to orange and yellow. . . . 123

A.3 Comparison between different implementations of the local photographic operator. Note that words vanished from the book in (b), and halos appeared around the lamp in (c). 126

A.4 Overview of the photographic tone mapping operator of REINHARD et al. (2002). The input Y is the HDR luminance ($L(x, y)$) and the output Y' is the compressed luminance ($L_d(x, y)$). 128

A.5 Overview of the SAT-based local photographic operator of SLOMP; OLIVEIRA (2008). 130

A.6 Efficient generation of the scaled luminance image L_r on the GPU. The logarithm of the luminance (b) of the input image (a) is submitted to a mip-mapping reduction stage (c) and then used to produce the scaled luminance image (d). 131

A.7 Equivalent lightness curve for red and blue according to the experiments of IKEDA; ASHIZAWA (1991). These curves summarize the experienced relative brightness from several colored cards against gray-scale patterns in different isoluminant environment conditions by several test subjects. 133

A.8 Opponent chromatic response of the HVS (a), and the $L^*a^*b^*$ color space (b). 134

A.9 Overview of the spatially-uniform mesopic vision reproduction filter. 136

A.10 Comparison between an image without mesopic simulation (a), with global mesopic filter (b) and per-pixel mesopic filter (c). All red intensities shift toward orange/yellow in (b), while only those not sufficiently bright enough change in (c) like the light reflex in the leftmost wall. Also note that purple tones shifted towards a more blueish hue in the mesopic images. 136

A.11 Overview of the spatially-varying mesopic vision reproduction filter. . 137

A.12 Overview of all the stages implemented in the demo program for the display of a complete frame. 139

A.13 Performance of the tone mapping stage alone, with and without the mesopic filters. 140

A.14 Performance times accounting for all the stages presented in Figure A.12 required for a full-frame display. 140

A.15 Overhead of the spatially-varying mesopic filter to the tone mapping stage. 140

A.16 Examples of the spatially-uniform and spatially-varying mesopic filters. The first row of the top images is without the filter, the second row is the spatially-uniform filter and the third row is the spatially-varying filter; similarly for the images at the bottom, but arranged in columns instead of in rows. 141

A.17 More examples of the spatially-uniform and spatially-varying mesopic filters. The leftmost image of each image set is without any filter; the rightmost is the spatially-varying filter. For these images, either the global or the local mesopic filters produce very similar results because there are no strong light intensities in the images, which makes the local averages to be somewhat close to the global average. 143

LIST OF TABLES

2.1 Hierachical map generation details for quick reference. 37

2.2 Frame rates using our method. All measurements account for animation time as well. Screen resolution of 1024x1024 pixels. FPS-T denotes tone-mapping. 42

3.1 Summary of results. The final minimized cost (absolute and relative) of each algorithm in each case is listed, as well as the total time that each algorithm took to find the solution. 58

3.2 Performance results for the CPU and GPU implementations of SoftAssign. CPUx1 stands for single-threaded execution, while CPUx4 represents a multi-threaded execution context with 4 threads. All time measurements are expressed in seconds. 66

4.1 Compilation of SAT generation times on different GPUs; **RD** stands for *recursive doubling* and **BT** stands for *balanced tree*. Performance values for both techniques with $k = 2, 4, 6$ and 8 are listed. 89

5.1 Performance of different screen-space ambient occlusion techniques. 111

A.1 Complete list of performance times profiled. The abbreviations in the table are as follows: (RL) relative luminance; (TM.N) tone mapping without mesopic simulation; (TM.MU) tone mapping with the uniform mesopic filter; (TM.MV) tone mapping with the varying mesopic filter; (FT) full frame time. 142

EXECUTIVE SUMMARY

This thesis encloses the achievements of research and development activities performed in a span of about 30 months (2.5 years, from early 2009 to mid-2011) at Hiroshima University. The problems addressed in this thesis are not particularly connected to each other, but the common theme surrounding them is the pursue of high-performing algorithms assisted by the programmable graphics hardware (GPU), targeting real-time frame rates whenever possible. The topics covered are distributed in two main categories, photorealistic and non-photorealistic rendering.

Namely, raindrop rendering (natural phenomenon) and soft ambient-like shadow generation (a subset of global illumination) were investigated in regard of photorealistic rendering. Another subject, photomosaic synthesis, serves as an example of a non-photorealistic rendering effect where numerical optimization techniques can substantially help to improve the quality of the results without the need of immense image databases. The implementation of one of these optimization algorithms, SoftAssign, was investigated and mapped to the GPU, thus comprising an instance of general-purpose computations on the GPU. Summed-Area Table generation is another example of general purpose computation investigated on the GPU, proving itself very useful for ambient shadow generation as well as for tone mapping and mesopic vision (a peculiar perceptual effect) simulation.

The research accomplishments and original contributions of this theses are all supported by (and published in) international peer-reviewed publication vehicles such as journals, conference proceedings, conference posters, book chapters and magazines.

The introductory chapter of the thesis offers an historical review on the evolution of the graphics hardware. This introductory chapter does not intimately relates to the subsequent technical chapters of this document; however many of the hardware terminology and features that are often referenced and used throughout the coming technical sections are briefed within the introductory historical review. This introduction is primarily intended for readers that are not accustomed with the past developments on GPU technology. Nonetheless, the chapter also serves as as a gentle recapitulation on graphics hardware the for the curious or experienced reader.

ABSTRACT

Recent advances on graphics hardware technology have been providing impressive breakthroughs on real-time graphics. With an ever increasing level of programmability and flexibility, the graphics hardware is being revised towards a more general-purpose architecture (GPGPU); a number of rendering-unrelated tasks already benefit from its performance and parallelism. Many problems, however, still exist on the ambit of real-time rendering in regard to natural phenomena simulation, dynamic global illumination and perceptual effects. The first contribution of this thesis is a technique for rendering millions of spherical raindrops at real-time frame rates. Following that, a GPGPU implementation of SoftAssign is investigated and applied to optimize photomosaic synthesis, a well-known non-photorealistic technique. This thesis also introduces a variant of the balanced tree approach for prefix-sum and summed-area table (SAT) generation which leads to improved performance, also implemented in a GPGPU fashion. Powered by this fast SAT generation, other techniques were devised: a screen-space ambient occlusion technique which can be setup in three distinct modes (each trading quality with performance) and a fast physiologically-based spatially-varying (per-pixel, local) mesopic vision filter for tone mapping operators.

Keywords: Natural phenomena, rain rendering, photomosaic, SoftAssign, prefix-sum, summed-area tables (SAT), high dynamic range (HDR), tone mapping, mesopic vision, ambient occlusion.

1 INTRODUCTION

Over the past two decades the impact of real-time 3D graphics has accentuated in the personal computer market. From the early ray-casting real-time 3D engines running entirely on low-end CPU to the advent of dedicated commodity 3D graphics hardware, home computer users were exposed to a new immersive and interactive (and perhaps addictive) experience at the comfort of their own homes, being this experience delivered by means of digital entertainment media (e.g. games), asset/office toolkit (e.g. CAD and drawing software) or even through new forms of information visualization. This trend has grown largely more recently, with 3D human-machine interaction being a common task on a daily basis: today, powerful 3D capabilities are embedded even into small devices such as smart phones, navigation devices and portable video-game consoles.

The following Section of this Chapter offers an historical review on the evolution of the graphics hardware, from the early days of 3D hardware technology for personal computers up to the current state-of-the-art. This reviewing Section does not intimately relates to the subsequent technical chapters of this document; however many of the hardware terminology and features that are often referenced and used throughout the technical Chapters are briefed within this retrospective. The following Section is primarily intended for readers that are not accustomed with the past developments on GPU technology. Nonetheless, the chapter also serves as as a gentle recapitulation on graphics hardware the for the curious or experienced reader.

1.1 Retrospective of the Modern Graphics Hardware

The typical raster-based graphics pipeline has low granularity which in turn allows many of its tasks to be performed in parallel over streams of data. This fact was exploited by early dedicated 3D hardware manufacturer pioneers in the late 70's and early 80's. A decade later hardware technology had evolved to a point that such dedicated hardware could be reproduced at a relatively low cost and shipped to layman's PC users. Up until the turn of the year 2000, the dedicated graphics hardware was mainly concerned in parallelizing demanding tasks of the graphics pipeline that involved geometric transforms and pixel processing. This hardware could be configured in a number of ways, but the shading tasks that it was capable were "cast-in-silicon" and could only be customized or reprogrammed up to some very limited extent. his category of graphics hardware was to become late referred to as as *fixed-function* pipeline or *configurable* pipeline..

By the early 2001, graphics hardware manufacturers (NVIDIA in particular) gambled on a new set of functionalities for their upcoming technology review: *programmable* shading units. Although very simplistic and limited in early releases, this novel concept gave to graphics software developers a powerful tool to experiment with new real-time effects in a number of creative ways. The flexibility and potential provided by such remarkable hardware was soon widely adopted and spread: future hardware releases not only kept this programmable subset but also offered ever-increasing functionality by eliminating previous programming limitations. Competition was also forced to embrace this new paradigm for high-performance graphics if they were to keep their share on the graphics hardware market. The term *Graphics Processing Unit*, or **GPU**, was soon coined by NVIDIA to this new graphics hardware architecture, while ATI had suggested VPU (Visual Processing Unit). The former came to be more widely acknowledged and the term VPU ended up fading out from the literature.

In order to understand how the transition from the fixed-function pipeline to programmable shading happened, it is worth investigating the transition between software-based 3D graphics pipeline (all steps done in CPU) and dedicated fixed-function graphics hardware. A typical raster-based graphics pipeline is described by the following steps:

- vertex transform: input "template" vertices in *object space* are transformed and animated to a different pose in *world space* which are then transformed to *camera (view) space* and, finally, transformed to a *projective space* (usually *perspective* projection); per-vertex lighting is also performed during this stage given that normal vectors were assigned to each vertex.

- primitive assembly: the transformed vertices are connected together by some implicit topology which the pipeline must be priorly configured for (such as triangles) and promoted to a *normalized device coordinate space* (this is the role of the "division-by-w" in homogeneous 4D space).

- polygon visibility: each primitive is submitted to a *culling* stage which determines if the polygon is entirely/partially visible or not (back-face culling also applies); partially visible polygons are submitted to a clipping procedure that modifies their topology to prevent further unnecessary processing.

- rasterization: each visible/clipped polygon is break into several fragments based on their coverage, each mapping to a single pixel in the render-target (usually the screen); per-vertex attributes (such as lighting, color, texture coordinates) are interpolated and assigned to each fragment as well.

- shading: each fragment is assigned a color based on interpolated per-vertex attributes and other implicitly bound resources (such as textures); each fragment is also assigned a *depth* value; per-pixel lighting can also be computed at this stage.

- composition: given a fragment's location, color and depth, the fragment is combined with potential already composed pixels in the render target where operations such as depth-test, alpha blending and stenciling are performed.

The input of the pipeline are essentially vertices with some attributes associated to them, as well as texture resources and configured states (such as the primitive topology,

blending factors, etc.). The output is a frame of pixels, the color buffer, which is then read by the display device and presented on the screen (although the output could also be redirected to some off-screen color buffer in memory). All stages can be busy independently processing different parts of the mesh being rendered, each performing their own specific tasks and forwarding the results to the next stage, allowing for a great level of parallelism.

The first generation of dedicated graphics hardware addressed the most costly operations of the pipeline: rasterization, shading and composition. A CPU program would then transform the vertices, assign attributes and perform per-vertex lighting and then issue them to the dedicated hardware. Since no actual 3D computation happened in the dedicated chip, this generation of graphics hardware became known as *2D accelerators*. Inside the chip, performance could be increased just by replicating more rasterization, shading and composition units as new hardware revisions were released.

As pixel throughput alleviated with the aid of 2D accelerators, applications could now spent more time with elaborated vertex transformations and increasing the polygon count of 3D meshes for additional detail. Soon these applications would be caught into a new bottleneck, this time on vertex processing.

Hardware architects then incorporated the vertex transform stage to the graphics hardware, and now all of the stages of the graphics pipeline lied within the same dedicated chip. This generation then came to be the first "true" *3D accelerators*. Once again performance could be increased throughout the hardware by replicating more of each dedicated processing unit within the chip. Applications were finally free of time demanding rendering tasks and could spent the extra CPU time on other tasks and preparing for the next frame as the 3D accelerator was asynchronously rendering the current frame.

Even though only hardware aspects were considered so far, software plays an equally important role. With 3D accelerators being produced by several distinct vendors with different architectural philosophies, it would be tedious, impractical and discouraging for a 3D graphics software developer to offer software compatibility with such a multiverse of hardware. Graphics APIs such as OpenGL and Direct3D gave freedom to the developer through an abstract layer to orchestrate the graphics hardware; an unified "language" between developers and hardware peculiarities. Graphics hardware vendors were then responsible for providing *drivers* that adhere to the requirements of a particular graphics API and expose hardware features through a standard abstraction model.

Software developers were given a powerful framework to produce rich 3D applications, but the spectrum of graphical effects possible was limited to the feature set implemented and supported by each individual hardware and the marketing strategy of each respective vendor. As already mentioned, the subsequent programmable graphics hardware gave this much wanted flexibility and power to the hands of creative graphics practitioners. The level of programmability offered was slowly introduced as new hardware revisions were released due to tight budget limitations and risk assessment connected to hardware technology trends at the time of launch.

Hardware architects started by identifying the stages of the pipeline that could benefit more from programmability, namely vertex transform and shading. The other stages are relatively standard procedures in which a graphics software developer would refrain of modifying due to the trade-off between the complexity involved, development time constraints and justifiable benefits achieved. Typical effects reproduced in early pro-

grammable GPUs are per-vertex displacement with noise, enhanced vertex *skinning*, improved per-pixel lighting and post-effects such as motion blur and glow.

The first generation of programmable GPUs was very restrictive in the number of instructions, video memory availability and flow control within each programmable unit. These restrictions were alleviated in the subsequent generation, but vertex and fragment shader remained distinct in the functionality each one could provide. A vertex shader, for instance, was forbidden of accessing texture memory. As new generations were introduced, these constraints were largely reduced, but vertex and fragment shading units were still kept separated in the chip, with typically many more fragment units than vertex units; fragment units were still more capable of accessing video memory, while vertex units had better dynamic branching support. Such architectural arrangement was also impeding the hardware to scale its performance between vertex-bound and pixel-bound applications.

An important feature that followed was the capability of fragment shaders to output data to multiple render-targets (color-buffers) at the same time. This allowed efficient implementation of sophisticated rendering techniques such as deferred shading. Textures and render-targets storage capabilities also improved, enabling floating-point texel data to be stored on them, which in turn reflected on more elaborated and efficient High Dynamic Range (HDR) rendering support.

As these and many other features were progressively added at each following hardware release, inter-compatibility between hardware vendors and software API was managed by the specification of standardized *Shading Models*. These models assist developers to safely plan, author and target different graphical effects based on the capabilities offered by the graphics hardware currently installed on a client's computer.

Programmable shading capabilities continued to evolve, and hardware architects decided to adopt an *unified* model to internally organize and distribute the processing shading units. With this unified shading model, no longer was a vertex unit "handicapped" in some aspects to a pixel shader unit or vice-versa: in fact, there was no more distinction between them. The developer would still implement vertex programs and fragment programs separately, but he same processing and memory access functionality was available equally for both programs. Moreover, the load balance between vertex and fragment processing was managed entirely by the hardware, making it much more scalable while, at the same time, more transparent to the developer.

An interesting change that came with unified shading was the use of *scalar* arithmetic units instead of 4D SIMD-based vector arithmetic units. This way the hardware could leverage the usage of each individual processor and no longer processing was wasted when using a whole 4D SIMD unit to process 1D, 2D or 3D data. Developers were now free of worrying with another hardware peculiarity. The unified shading model still remains to date as the architectural choice for arranging the shading units in the chip.

Similarly to the case of graphics API, standardized ways of writing shader code and targeting different hardwares and vendors was made necessary. In the early days of programmable GPUs one would be forced to write vendor-specific assembler code that not only would differ amongst vendors, but would as well change between hardware releases of the same vendor. This lead to a rapid development of *shading languages* right from the infancy of programmable GPUs with the release of NVIDIA Cg and Microsoft HLSL (High-Level Shading Language) which later culminated in the GLSL (OpenGL Shading

Language). Such languages not only provided a common ground for developers to implement shaders in a portable, multi-platform fashion, but also provided an indispensable framework to produce highly-optimized machine code on-the-fly.

Programmability also increased with the addition of geometry shaders. For the first time a programmer was given the freedom of processing entire primitives instead of only streams of vertices. This way a particular topology could be either amplified, simplified or completely removed. Geometry shading happens just after vertex shading but before culling/clipping. Amplification could only be performed up to some limited extend, but enough to allow for interesting level-of-detail effects to be computed on-the-fly; the geometry around an object's silhouette, for instance, could be refined to provide more circular curves. Hair, fur, grass, rain and even entire particle engines could also be generated, managed and animated more efficiently with the aid of geometry shaders.

Another improvement on the hardware side was the possibility of streaming-out processed vertices (or entire primitives) directly to a memory buffer without the need of rasterizing the polygons. Stream-out happens after either the vertex shading or geometry shading stage. Before this feature was established, vertices fed to the pipeline would follow all the way through the stages until appearing somewhere in the color buffer, since rasterization was a must. This stream-out feature became known as *transform feedback* and positively favored applications that required physically-based particle simulation, hierarchical animation frame updates and isosurface extraction, to cite a few.

The latest programmability improvement introduced, *tessellation*, came to complement the limited amplification capabilities of geometry shaders. Developers were given two extra programmable stages: the *hull shader* and the *domain shader*. The hull shader operates on the control points of connected quadrilateral surface *patches* for effects like skeletal animation. The transformed control points are then feed to a *fixed-function* tessellation unit which will subdivide these patches into refined triangles based on a tessellation factor. The newly tessellated vertices are then submitted to the domain shader which can apply fine-controlled perturbations such as per-vertex displacement. The domain output connects to the geometry shader which in turn connects to the rest of the pipeline.

This summarizes the evolution of the graphics hardware, as well as the main real-time graphics software technology and standards, up to the current state-of-the-art. Most of the evolution was aimed towards enhancing the flexibility and possibilities of the graphics pipeline. However, there was another driving force also influencing on some of the hardware development cycle decisions: the curiosity, affordability and prospects of using the underlying graphics hardware infrastructure for solving general-purpose, rendering-unrelated problems. When programmability first appeared in the GPU (and even before) it did not take long for creative enthusiasts to experiment on mapping graphics-unrelated tasks to the patterns of the graphics pipeline. These attempts caught the attention of hardware manufacturers for future market share possibilities. This development paradigm came to be later coined as General-Purpose GPU Programming, or GPGPU for short.

A typical GPGPU application would have to somehow speak the language of rendering: data would have to be encoded as textures, the output would be encoded as one or more color-buffers, and at some point vertices would have to be emitted in order to trigger the pipeline. As a simplistic example, imagine the multiplication of a matrix with some scalar. The matrix would be uploaded to GPU memory as a 2D texture, the scalar would

be uploaded as a shader constant (could also be a single-texel texture), and the target color-buffer set to be of the same size and pixel format of the input matrix texture. The fragment shader code would perform the multiplications by accessing one element of the matrix (one texel) and multiplying its value with the scalar. In order to activate the fragment shaders, the application would issue the rendering of a single planar quadrilateral large enough to cover all the pixels of the render-target (processed by a very minimalist vertex shader code). Once the quadrilateral is transformed, rasterization follows by generating the corresponding fragments and interpolated attributes, such as texture coordinates. For each produced fragment, the fragment shader is invoked with the interpolated texture coordinates serving as an address/index that uniquely identifies one element of the matrix (one texel in the encoded texture). Once all fragments are processed and merged in the render-target, the application could read-back the output color-buffer to CPU memory or keep it in video memory for subsequent general-purpose operations.

This poses an interesting question: why would someone bother to walk through unintuitive grounds to solve some generic problem using a dedicated hardware that was never designed for it in the first place? The short answer should be easy to guess: performance. Dedicated, task-specific hardware, such as a GPU, can be up to orders of magnitude faster than a general purpose CPU (for that task). Besides, a GPU also offers parallel computational units that vastly supersede in numbers the amount of cores of typical CPU dice. The long answer includes two other attractivenesses of GPU: availability and affordability. Over the past decade GPUs become increasingly available even in low-end computers due to the ever decreasing monetary costs of GPUs of previous generations.

These GPUs were also expected to stay idle most of the time on a typical daily-basis computer user experience. Even though a regular user is unlikely to benefit from some computationally demanding, complex, graphics-unrelated scientific simulation application, his/her idle GPU could be used to accelerate more likely tasks on a daily-basis such as video decompression and playback, network packet filtering, file compression and encryption, data sorting, image and audio editing and processing, just to mention a few.

Another favorable, indirect factor for GPGPU was the fact the consumer-level CPUs seem to have achieved a stall in terms of clock frequency over the past decade, with an increasing effort to resolve more instructions per clock (superscalarity) and add more cores to the same CPU die. It is a known fact that a GPU has much lower clock rates than a CPU of the same period, but in contrast these GPUs have orders of magnitude more of arithmetic cores than those CPUs. Moreover, the architectures differ substantially since the problems that each attempt to solve are, at best, detached. Clock frequency alone, therefore, can be misleading and is hardly used as a raw performance gauge nowadays, even when confronting similar architectures. If one wants to compare the performance heterogeneous architectures such as GPUs and CPUs, another metric[1] provides a much more reliable face-off: FLOPS, or floating point operations per second, which is widely adopted by the hardware industry.

The chart presented in Figure 1.1 depicts the theoretical peak performance in GFLOPS (Giga – 10^9 – FLOPS; billions of FLOPS) of Intel CPUs and NVIDIA GPUs over a period of about 5 years. Although a bit out-dated for today's standards, it is clear that GPUs have

[1] Transistor count is also an interesting measurement, but correlates more closely to the chip complexity, die size and power consumption than to raw performance. As a matter of fact, GPU chips are currently vastly superseding CPU chips in absolute transistor count.

a radical ante in the FLOPS race. As of March 2011, the NVIDIA GTX 590 is anonymously being reported to peak at 2500 GFLOPS by Internet peers, while an Intel Core i7 990X barely crosses the 100 GFLOPS barrier. The retail price is also more favorable towards GPUs: U$699,00 for the GTX 590 against U$999,00 for the i7 990X.

The short conclusion is clear: GPUs crossed the Tera (10^{12}) FLOPS, while CPUs still crawl for a few hundreds of GLFOPS. The reality, however, deviates from these numbers. In practice, CPUs achieve theoretical peaks much easier and stably than GPUs. The tasks for which each was designed also vary dramatically. The floating point precision inspected and compiled in the chart is single precision for GPUs and double precision for CPUs. Regardless, single precision suffices for many tasks and the raw performance of GPUs is just to advantageous to be left untapped; after all, letting the numbers alone speak for themselves, 10x to 25x for less (price) is just too attractive to be ignored, not to mention that GPU prices tend to lower faster than CPU prices over time.

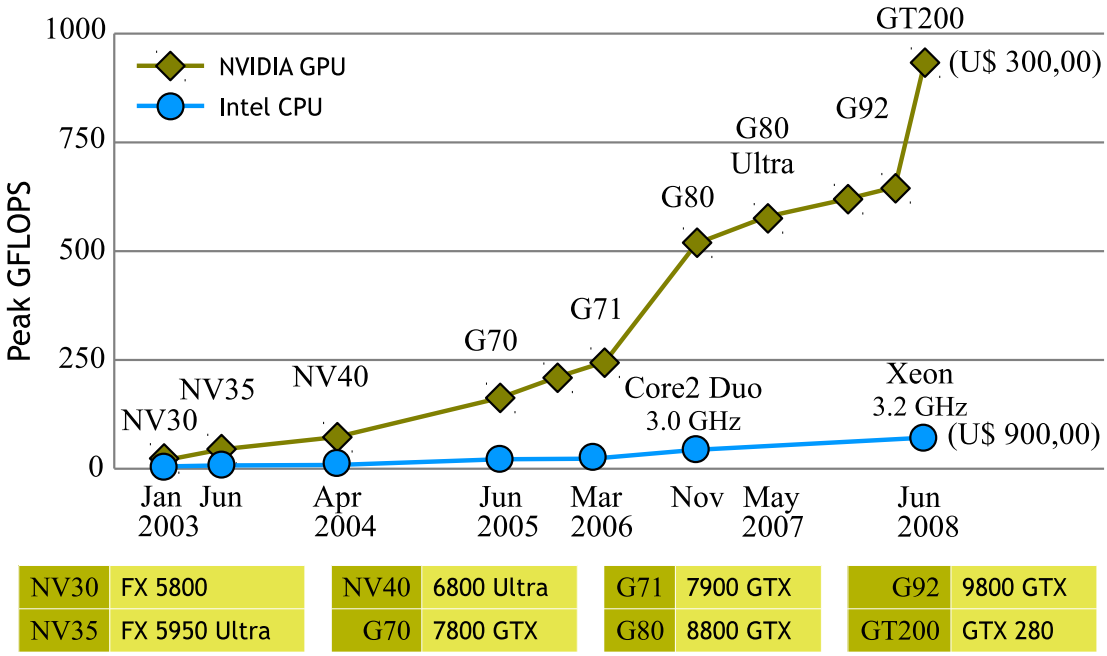


Figure 1.1: Performance increase between NVIDIA GPU and Intel CPU architectures over time. The retail price at launch time of latest hardware is shown in parenthesis.

Another question then arises: how come for GPUs to be retailed cheaper while holding more transistors and performing better than their CPU counterparts? The answer lies again on the fact that GPUs and CPUs are designed to assist in solving farily distinct problems. Even with the programmability and higher level of parallelism of GPUs, the main goal is to support and accelerate rendering tasks, while CPUs take a much more broad road to aid any task (even rendering) in a more equally performing, "democratic" fashion. As a result, CPUs tend to spend a big portion of their transistors on additional efficiency and flexibility for randomly accessed memory patterns, complex cache hierarchies to hide memory latency, and micro-level optimizations such as branching prediction, speculative execution and loop/flow control. In contrast, GPUs do not need large or sophisticated caches or efficient random-access memory support since data flowing in the pipeline is mostly stream-based; in fact, reading/writing simultaneously from/to the same video memory region is strictly forbidden or, at best, leads to undefined behavior.

However, GPUs do need as many arithmetic units as they can possibly fit in the die; this extreme parallelism in turn also helps on hiding memory and flow latencies by switching the execution of pending units with non-pending ones, going back to the former when data becomes readily available.

Even with performance and price advantages, GPUs still comprise dedicated hardware for rendering. Anyone interested on GPGPU would need at least some moderate familiarity and experience with real-time computer graphics and the rendering pipeline. Along with the early experiments with GPGPU, computer scientists began researching, designing and developing GPGPU languages that could abstract the details of the rendering pipeline for unexperienced graphics developers. Two of these languages, Brooks (Stanford University) and Sh (University of Waterloo), attracted the attention of many researchers. These languages not only hid the pipeline, but also provided programming functionalities through a more familiar C/C++ syntax than the existing shading languages. Needless to say that these languages also caught the attention of the GPU industry and vendors. This would soon culminate in vendor-specific GPGPU languages: NVIDIA CUDA (Compute Unified Device Architecture) and AMD/ATI CTM (Close To Metal).

Important developments also happened to the underlying GPU architectures as these vendor-specific languages were introduced. Unified shading and scalar processing units are two of them, since on a GPGPU scenario having dissimilar processors with different capabilities built upon vector-based arithmetic units would ultimately hurt scalability of the hardware to solve more generic rendering-unrelated problems. These architectures were also careful to include small, but fast on-chip memory blocks that could be shared amongst groups of threads executing similar tasks, as well as much-wanted synchronization directives, albeit in a very localized fashion.

Developers were once again given power, expressiveness and flexibility, but bound to vendor-specific frameworks. The industry as a whole started defining programming interfaces to unify this multitude of GPGPU-capable hardware under open, standard software abstraction layers, which came to be realized in current technologies like Khronos' OpenCL (Open Compute Language) and Microsoft's DirectCompute and C++AMP. As for vendor-specific languages, AMD/ATI decided to discontinue the development of CTM and adopted OpenCL as their main GPGPU development platform. NVIDIA still prioritizes its proprietary CUDA technology where experimental features can be introduced and managed in a much faster pace (not to mention the availability of numerous reliable libraries and documentation produced over the years); other standard GPGPU platforms like OpenCL are implemented by NVIDIA as wrappers around the CUDA framework (similarly, GLSL is supported through a transparent layer around NVIDIA Cg).

A wide range of applications already benefit from GPGPU, most of them scientific or engineering-related. That does not mean that all sorts of applications can significantly be accelerated with GPGPU, which is a common misassumption taken for granted by novice GPGPU practitioners. Writing optimal GPGPU code requires smart usage of on-chip shared memory, caution with memory coalescence and memory bank access patterns, clever dispatch and granularization of computing kernels, and careful management of small, localized synchronization points while at the same time being consistent to the larger context that encapsulates these execution (thread) groups.

This overwhelming amount of low-level details may distract the implementation from the central task being performed, which can imply in large development cycles and unintuitive code that complicates further maintainability. In fact, efficient GPGPU programming using dedicated GPGPU frameworks such as NVIDIA CUDA can be as challenging (sometimes perhaps more) as with traditional GPGPU over the rendering pipeline. Interoperability between GPGPU APIs and rendering APIs (like OpenGL) is still not free of overheads, although the gap between them has widely reduced over the past technological advances. Some of the rendering functionality, such as texture filtering and caching, mip-mapping, color blending, hardware interpolators and discrete derivatives can still be invaluable useful on some generic tasks, yet these features were just recently exposed, up to some extent, to more general GPGPU frameworks.

In conclusion, it can be seen that GPU architectures have radically evolved in a comparatively short period of time of a decade or so. The graphics hardware first appeared as a parallel realization of a very specific task, the rendering pipeline, with little-to-none level of customization. As programmability in GPU became mainstream, other graphics-unrelated parallel problems started to be solved in the GPU, which matured in a hardware that is becoming increasingly less specific. It is accurate to say that today's state-of-the-art GPU technology resembles much more a general-purpose parallel computing than a dedicated rendering hardware. The graphics pipeline no longer sits at the core of the hardware; instead the pipeline is implemented around the parallel computing capabilities offered by the hardware.

As GPUs become more programmable and flexible, and as CPUs become more multi-core, a clash between these technologies seems unavoidable. Current GPU architectures are expanding their capabilities towards double-precision floating-point computation, and video-memory continues to increase. In fact, some hardware like NVIDIA Tesla are being marketed as an exclusive high-performing computing co-processor which do not include video output. At the same time, CPU technology continues to increase the amount of cores per die for increasing parallelism. Technologies such as Larrabee (Intel's prototype codename) and the Cell processor (Sony/Toshiba/IBM) are clear examples of CPU architectures aiming vast levels of parallelism based on many simplified CPU cores, similarly to how GPUs are architected.

Despite the expected upcoming technology collision, one has to keep in mind that the main driving force behind GPUs is still rendering. The level of realism attained by talented developers on modern computer games is indeed impressive, but real-time computer graphics still has many problems that require solutions. The ultimate goal of computer graphics remains as the accurate (photorealistic) synthesis of real-world scenes, a task that requires judicious observation and simulation of complex light-matter interaction. Natural phenomena simulation and dynamic global illumination and shadowing still impose great challenge for accurate reproduction in real-time. Perceptual effects of the Human Visual System are equally important and challenging because they comprise the interface between the physical world and the perceived world.

1.2 Thesis Contributions

The main, novel contributions of this thesis are listed below:

- A raindrop rendering technique that is capable of animating and rendering millions of spherical raindrops in real-time with visual quality comparable to ray-tracing. The technique accounts for Fresnel-based refraction and reflection. Multi-sampled transparency is also efficiently handled around the edges of the raindrops for increasing quality. Hardware mip-mapping is exploited in a non-orthodox way.

- The analysis of three optimization algorithms applied to photomosaic optimization: a greedy-based search, Simulated Annealing and SoftAssign. These optimization strategies largely reduce the amount of tiles required to build quality photomosaics, specially when repetitions are prohibited. A GPGPU implementation of SoftAssign is also investigated, leading to speed-ups of up to 60x when compared to optimized CPU implementations.

- An improved version of the binary balanced tree pattern applied to perfix-sum and summed-area table generation which allows the reduction and expansion of more elements per pass (k-ary balanced tree). The technique only requires a slightly more involved implementation which maps well to the GPU. This improved version not only accelerates the traditional algorithm (optimal speed-up is hardware-dependent), but also reduces the amount of intermediate memory necessary for GPGPU implementations.

- A screen-space ambient occlusion technique based on summed-area tables. Despite some practical limitations, the technique is faster that previous screen-space ambient occlusion techniques and eliminates the need for further palliative amends such as low-pass filtering. The foundations of the proposed technique can be applied in three different ways: minimalist, normal-guided and depth-refined. Each method trades-off image quality with performance and memory consumption.

- A spatially-varying mesopic vision filter that can be applied together with any existing tone mapping operator, as long as the operator provides measurements of local luminance. Chrominance alteration is computed in an orthogonal stage to luminance compression. The performance overhead introduced is practically negligible. The devised mesopic model is simple yet effective since it based on psychophysical experiments and supported by recent physiological evidence.

1.3 Structure of the Thesis

The remaining of the text of this thesis is organized in self-contained Chapters: each Chapter has its own abstract, introduction, background review, technique description, result analysis, conclusion, limitations and future work. At the end of each Chapter, a list of publications based on the results of the corresponding research is presented. Some Chapters may also contain appendices that appear following the list of publications. All references cited throughout the Chapters are compiled together in an unified bibliographic section at the very end of this document.

A total of four technical Chapters follow this introduction and are listed below:

- **Chapter 2: Photorealistic Real-time Rendering of Spherical Raindrops with Hierarchical Reflective and Refractive Maps** (pages 29–49)
- **Chapter 3: GPU-based SoftAssign for Maximizing Image Utilization in Photomosaics** (pages 49–76)
- **Chapter 4: Efficient Summed-Area Table and Prefix-Sum Generation on the GPU** (pages 76–121)
- **Chapter 5: Screen-Space Ambient Occlusion Through Summed Area Tables** (pages 97–116)

The thesis also contains one appendix – this appendix appears disconnected from the main technical chapters because the research and results presented on this appendix Chapter were not published in open publication vehicles:

- **Appendix A: Fast and Robust Spatially-Varying Mesopic Vision Simulation for Tone Mapping Operators** (pages 121–97)

There is also a general conclusion that summarizes the thesis in Chapter 6. A complete, unified list of all of the published works related to the thesis is presented immediately after the conclusion in Chapter 6, but before the bibliographic section. A list of abbreviations and acronyms, figures and tables referenced throughout this thesis are presented after the Table of Contents but before the Executive Summary.

Even though the technical Chapters are self-contained, with little-to-none cross-referencing between them, and can be browsed in any order, it is recommended to read **Chapter 4: Efficient Summed-Area Table and Prefix-Sum Generation on the GPU** before proceeding to either **Chapter A: Fast and Robust Spatially-Varying Mesopic Vision Simulation for Tone Mapping Operators** or **Appendix 5: Screen-Space Ambient Occlusion Through Summed Area Tables**, since summed-area tables are used as a building-block for the spatially-varying mesopic filter and for the screen-space ambient occlusion technique that follows.

2 PHOTOREALISTIC REAL-TIME RENDERING OF SPHERICAL RAINDROPS WITH HIERARCHICAL REFLECTIVE AND REFRACTIVE MAPS

2.1 Abstract

Synthesizing rainy images is a common challenge found in film, game-engines, driving simulators and architectural design. Simulating light transport through a raindrop's optical properties is a view-dependent problem and large quantities of raindrops are required to produce a plausible rainy scene. Accurate methods for rendering raindrops exist but are often off-line techniques which are cost prohibitive for real-time applications. Most real-time solutions use textures to approximate the appearance of moving raindrops as streaks. These approaches produce plausible results but do not address the problem of temporal effects such as slow-motion or paused simulations. In such conditions, streak based approximations are not suitable and proper raindrop geometry should be considered. This chapter describes a fast and practical approach for rendering raindrops in such temporal conditions. The proposed technique consists of a preprocessing stage which generates a raindrop mask and a run-time stage that renders raindrops as screen-aligned billboards. The mask's contents are adjusted based on the viewpoint, viewing direction, and raindrop position. The proposed method renders millions of raindrops at real-time rates in current graphics hardware, making it suitable for applications that require high visual quality without compromising performance.

2.2 Introduction

Realistic image synthesis is one of the most relevant subjects in computer graphics, and rendering realistic rainy scenes remains a challenge. Rain is a common natural phenomena and its simulation has been widely used by the film industry, graphics engines, driving simulators and architectural design, to improve immersion and realism, or to set mood. Common approaches to the problem, although accurate, usually have a computational cost that is prohibitive for real-time applications. Moreover, a plausible rainy scene may have to hold large numbers of raindrops, introducing even more complexity to the problem and significantly penalizing performance at the same time.

Determining a raindrop's visual appearance is a view-dependent problem, and can be divided in two parts: reflection and refraction. Reflection happens in the raindrop's front-facing surface (with respect to the viewer) and its overall influence on a drop's visual appearance is given by the Fresnel equations. Similarly, refraction determined by Snell's Law also happens in the front-facing surface, but once a ray is refracted, it travels inside the raindrop's geometry and eventually leaves it from the back-facing surface, refracting again. Both the reflected and the secondary refracted rays continue traveling in the scene and may intersect objects, light sources or other raindrops, leading to a recursive problem. This recursive characteristic makes this problem well suited for ray-tracing methods [WHITTED (1980a)]; however, they are computationally expensive, thus approximations are required to allow real-time performance.

In addition to the optical properties of raindrops, a realistic reproduction of a rainy scene requires extra simulation efforts regarding how the human visual system responds to light arriving from a real raindrop. The retina in the human eye takes time for an image to fade and thus high velocity raindrops are perceived as streaks [CHANGBO et al. (2008)]. This perception phenomena is often referred to as retinal persistence and has been exploited by many researchers in order to approximate the overall appearance of rain in interactive applications [GARG; NAYAR (2006); STARIK; WERMAN (2002)].

Figure 2.1: Left) Closeup of a ray-traced raindrop. Center) Same raindrop rendered using the texture-based approximation described in this paper. Right) A rainy scene rendered using the proposed technique. Results were generated using high dynamic range environment maps together with tone-mapping. Left and center images are nearly identical but the proposed technique can render one million raindrops at 1024x1024 screen resolution at 140FPS on a GeForce GTX 280 (performance measurement includes animation and tone-mapping).

Retinal persistence allow applications to approximate raindrops as streaks while still producing convincing images. Although these techniques are effective for raindrops moving at high speeds, they fail to reproduce all the specular details of droplets in stationary, paused simulation or slow-motion scenarios, which are common in recent games and films. Typical tasks include instant-replays, changing the rate of time, and raindrops on the window of a car. In such situations rendering rain as streaks is unacceptable and proper rendering of its specular details would require ray-tracing based approaches, prohibiting their use in real-time. This paper does not deal with streak-based raindrops but instead focuses on reproducing the optical characteristics considering geometry for large quantities of raindrops at high frame rates.

The proposed technique addresses the problem of rendering raindrops by using screen-aligned billboards. An hierarchical map containing both reflection and refraction data of a single raindrop is generated in a preprocessing step. This hierarchy is used as a mip-map texture during run-time but its contents are not rendered directly on the billboard. Instead it is used as input data to estimate the reflected and refracted vectors of a raindrop based on its relative position and orientation to the viewpoint. The transformed vectors of the raindrop texture are then used to sample an environment map, determining the color of each pixel by modulating the reflection and refraction contributions based on Fresnel coefficients. Although previous research suggested that reflection does not contribute significantly to the overall appearance of a raindrop due to the Fresnel effect [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)], we reinforce the observations of Garg and Nayar [GARG; NAYAR (2004a)] that when using high dynamic range environment maps, in vogue in the computer graphics' community, the contribution of reflection to a raindrop's appearance is significant, even after luminance compression (tone-mapping) and should be preserved, as is apparent in Figure 2.2.

Figure 2.2: Raindrop comparison: Left) A raindrop rendered using low dynamic range environment map. Center) Raindrop rendered using a high dynamic range environment map. Right) Photographs of two real-world water droplets taken outside by using a water dropper and a regular digital camera. Note that despite the crude equipment used the droplets closely resemble spherical shapes. When using HDR lighting the results are more similar to the real-world. Reflections in the upper portion of the raindrops which do not exist in the LDR case are clearly visible.

The proposed method handles millions of raindrops simulated and rendered at the same time, while maintaining real-time performance with current graphics hardware. Despite our high frame rates the visual quality of the synthesized raindrops are nearly equivalent to ray-traced raindrops, as shown in Figure 2.1, making it suitable for applications that require high visual quality without compromising performance. This paper makes the following contributions:

- Rendering of millions of raindrops in real-time, in current high-end consumer graphics hardware, with visual quality comparable to ray-tracing;
- Uniformly accounts for refraction, reflection, and Fresnel, demonstrating that reflection is significant when using HDR illumination and should be preserved;
- Exploits simple and efficient features of graphics hardware, making this method less intrusive and easy to integrate into existing graphics engines.

2.3 Related Work

Rendering raindrops has been a common research subject of recent years. A real-time rendering solution for rainy scenes was proposed by Tatarchuk where raindrops are rendered as rain streaks, using artist-driven procedural texture generation, which are dynamically updated and illuminated in an image processing step [TATARCHUK (2006)]. Wang *et al.* proposed a method which involves analyzing rain in videos and then extracting textures to place in a new scene [WANG et al. (2006)]; the technique is primarily concerned with rain streak quality. Recently Changbo *et al.* proposed a real-time solution which accounts for many of rain's lighting side effects, such as fog and rainbows, but raindrops are presented as streaks here as well [CHANGBO et al. (2008)]. Although rain streak based methods provide attractive results, they suffer from the inability to render stationary droplets or rain in slow-motion and paused simulations, relying in roughly approximated refraction values which are unable to recreate the shape and specular properties of raindrops as would be seen in such temporal conditions. All of these streak-based methods have convincing quality and perform well. The remaining discussion will therefore focus on techniques to render raindrops for cases that are related to stationary or temporal conditions.

A method for rendering raindrops moving down surfaces was proposed by Kaneda *et al.*, where a mesh surface is created to determine droplet motion based on mesh region parameters [KANEDA; IKEDA; YAMASHITA (1999)]. Their rendering approach is described in [KANEDA; KAGAWA; YAMASHITA (1993)], where scene objects are projected onto an environment map, and ray-tracing is performed, retrieving light intensities from the scene-object-imposed environment map, thus impeding real-time performance. Similarly Wang *et al.* developed a method for rendering water drops, allowing raindrops to separate and recombine as they flow down arbitrary surfaces [WANG; MUCHA; TURK (2005)]. Their approach is more concerned with simulating the way small portions of water move about on a surface, thus intrinsically having a high computational cost, so they performed rendering with ray tracing.

Garg and Nayar did a number of works about raindrops: a raindrop's appearance [GARG; NAYAR (2004a)], rain removal from video [GARG; NAYAR (2004b)], and later

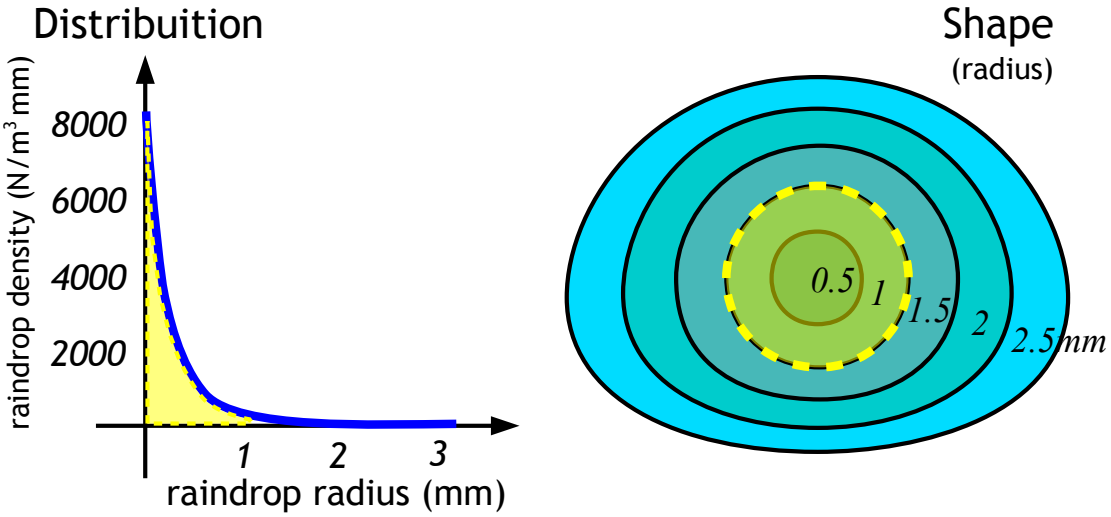


Figure 2.3: Raindrop distribution and the corresponding geometrical shape: in a typical rainy scene most raindrops are smaller than 1mm in radius thus being nearly spherical in shape.
Image adapted from Garg and Nayar [GARG; NAYAR (2004a)] using distribution and shape data from Marshall and Palmer [MARSHALL; PALMER (1948)] and Beard and Chuang [BEARD; CHUANG (1987)].

a system which creates a vast database of rain-streak textures to best simulate a wide variety of lighting conditions [GARG; NAYAR (2006)]. In their report [GARG; NAYAR (2004a)], they make many important observations about a raindrop's appearance. Based on two models [MARSHALL; PALMER (1948); BEARD; CHUANG (1987)] they show that a significant portion of raindrops in a rainy scene are less than 1mm in radius, and thus almost spherical in shape, as summarized in Figure 2.3. Their work provides the foundation for the geometrical representation of raindrops used in this paper.

Using the observation that raindrops can be geometrically represented as spheres, naïveapproaches can be done on the GPU. A sphere can be rendered for each raindrop, and for each generated fragment, a single refracted vector is computed and used to fetch color in an environment map. However, a single refracted vector is inaccurate. One could think of performing a secondary refraction by using the position and size of the raindrop and computing the internal intersection, thus, determining the outgoing (secondary) refracted ray. In any case, raindrops near the viewer would require a high polygon count sphere in order to produce smooth results. Level of detail techniques reduce geometrical overhead, but require sorting which may be expensive when a large number of raindrops are present. Furthermore the computation of the second intersection and Fresnel coefficient is a relatively expensive operation, one that would have to be done for every fragment, potentially introducing a bottleneck. Another option would be to use GPU-accelerated ray-tracing [PURCELL et al. (2005); CARR et al. (2006)], but even with the parallel processing of current graphics hardware performance would be difficult to maintain when a large number of raindrops are present. Keeping dynamic data structures to accelerate rendering is expensive in both computational effort and memory requirements in such conditions.

The image based approaches for interactive refraction proposed by Wyman, as well as Brauwers and Oliveira, could also be suitable for raindrop rendering, since they provide

a solution for dual layer refraction [WYMAN (2005); BRAUWERS; OLIVEIRA (2007)]. Their methods render the back-face surface of the refractive object in one step, and in a second step, the fragments' position in the frontal surface is used to produce an internal refracted ray, which is then used to determine the internal intersection and perform the next refraction in the aforementioned back-face surface. Their approaches require finding the distance between the front and the back surfaces, forcing objects to be rendered in individual steps. Thus, these methods are suitable to render small numbers of complex refractive objects. In the case of rainy scenes, individual rendering steps for each raindrop would drastically affect performance.

The method proposed by Rousseau *et al.* introduces a novel approach for raindrop rendering, using a preprocessed vector mask to store offset vectors to adjust the refraction of raindrops in run-time [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)]. Based on the maximum angle that a refracted vector can have in a raindrop, a background image is rendered every frame using a wide field of view, roughly 135 degrees. Raindrops are rendered as billboards and their appearance is determined by projecting the fragment's world position to the background image and applying the offset vector previously stored in the vector mask. Since the background image holds distortions near its edges due to the wide field-of-view, refraction may not be accurate. Additionally, reflection is not possible, since it would require a secondary background image where its field-of-view would exceed 180 degrees. In their method, only one vector mask is used, regardless of the distance between the raindrops and the viewpoint. This leads to inaccurate results, since refraction strongly varies with the distance between raindrop and viewpoint, as shown in Figure 2.4. Lastly the resolution of their background image is critical to determine the sampling quality. Although their method uses arbitrary raindrop shapes, based on Beard and Chuang model [BEARD; CHUANG (1987)], non-spherical shapes do not allow for arbitrary camera rotations, since the silhouette of the raindrop would change but not the shape baked in the mask.

The approach to be described in this chapter is based on Rousseau *et al.*'s method [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)], but the aforementioned limitations are alleviated by exploiting features of the graphics hardware. The proposed method does not require additional rendering steps: vectors are adjusted using a rotation matrix instead of offset vectors and fragment color is retrieved from the environment map without the need of background images. Unlike Rousseau *et al.*'s method, the proposed technique also naturally extends to reflection. Only spherical-shaped raindrops are considered which could be seen as a limitation regarding Rousseau *et al.*'s work, but we reinforce that spherical-shaped raindrops suffice for typical rainy scenes as demonstrated in [MARSHALL; PALMER (1948); BEARD; CHUANG (1987); GARG; NAYAR (2004a)] (see Figure 2.3). Additionally, spherical raindrops allow arbitrary camera rotations, preserving the raindrop's shape. This leads to a uniform, less intrusive and easy to integrate solution. Our results are similar to ray-traced images yet still rendered at considerably higher frame rates than the aforementioned techniques.

2.4 Proposed Method

The proposed technique is split in two steps: a preprocessing and a run-time stage. First a raindrop mask is ray-traced but unlike traditional ray-tracing, vector data regarding

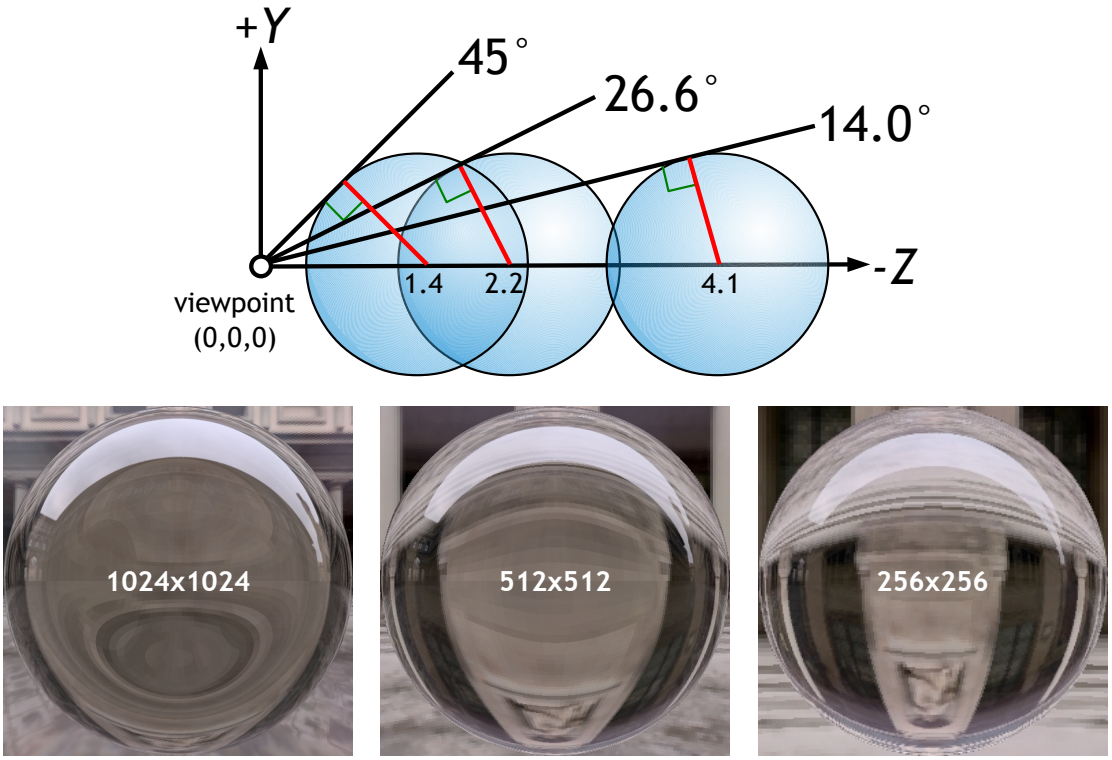


Figure 2.4: Mip-map Generation: Positioning raindrops so that their surface area is maximized in the projection plane is important. Given an aperture angle, the distance (hypotenuse) is obtained by simple trigonometry, using the radius of the sphere (opposite side) and the observation that aperture vector tangencies the sphere forming a right angle with the radius vector at that point (marked in red). Below are the corresponding renderings of the illustration in the upper portion. As can be seen the distance from the raindrop to the camera drastically changes the raindrop's appearance and must be taken into account for accurate rendering.

light's reflection and refraction directions are stored rather than color data. The mask is generated at predefined and convenient conditions in order to simplify calculations during the run-time stage. In fact, the resulting mask is composed of several masks, represented as a hierarchical structure identical to a mip-map chain. Once the mask is ready, it is used as a texture map to render a single screen-aligned billboard for each raindrop. The texture's contents (reflection and refraction vectors) are transformed per-fragment according to a rotation matrix derived in a per-raindrop (vertex) basis. Finally the resulting vectors are used to fetch color information stored in the environment map. Both preprocessing and run-time stages will be discussed in more detail in the following subsections, as well as the assumptions and observations made to allow for real-time frame rates.

2.4.1 Assumptions and Observations

We base the geometry of our raindrops on the Marshall-Palmer model [MARSHALL; PALMER (1948)] and the Beard and Chuang model [BEARD; CHUANG (1987)], suggesting that most raindrops are less than 1mm in radius thus have nearly spherical shape

(recall Figure 2.3). From the observations of Garg and Nayar we can see that raindrops have three main components [GARG; NAYAR (2004a)]; in order of greatest contribution they are: refraction, reflection, and total internal reflection (TIR). Due to TIR's minuscule contribution to the overall appearance of a raindrop it is unnecessary to render.

We employ image-based lighting with environment maps to estimate the light transport through raindrops. As a result any scene objects which have not been imposed on the environment map will be ignored. This is a significant assumption since it approximates the position of scene objects, but this approach is still a very common practice for real-time graphics and creates convincing results. This assumption however, does not limit our technique to omni-directional maps to perform illumination: the adjusted reflected and refracted vectors computed in the run-time stage could be used in conjunction with typical illumination models.

Although researchers have shown that reflection is not significant when rendering raindrops in low dynamic range environments due to the falloff effect produced by the Fresnel equation [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)], this is not the case when using high dynamic range (HDR) intensities to represent lighting information. The Fresnel equation show that as the angle between an incident vector and a surface normal approaches 0, the smaller the Fresnel coefficient becomes; the lowest value exists when this angle is zero, giving an overall reflection contribution around 2%. This is small when considering LDR intensities which are normalized between 0 and 1. However an intensity of 100 which can occur in HDR already yields an intensity beyond the display capabilities of typical display devices. Properly displaying HDR intensities to the much narrower range supported by typical display devices requires luminance compression, which is often referred to as tone-mapping. We observed that, when rendering with HDR intensities and performing tone-mapping, reflection is still clearly noticeable in the raindrop surface even after luminance compression. Figure 2.2 illustrates this observation, showing raindrops rendered using both HDR and LDR light intensities, as well as real world examples.

2.4.2 Preprocessing

The goal here is to ray-trace a single raindrop producing a vector mask which will hold, for each pixel that intersects the raindrop, reflection and refraction vectors, and Fresnel coefficients; pixels that do not intersect the raindrop will have these data set to zero and will be discarded in the run-time stage. For simplicity, the viewpoint is placed at the origin (0,0,0) with the viewing direction aligned toward the world's $-$Z axis in a right-handed coordinate system. The raindrop is geometrically represented as a sphere of unit radius and its center is placed at some distance along the viewing axis ($-$Z). This controlled setup simplifies the derivation of the proper rotation matrix that will be used during the run-time stage.

The raindrop is placed at an optimal distance that maximizes its projected area in the plane of projection. Assuming a camera with 45 degrees of aperture (90 degrees of field-of-view), by using trigonometric relations, the distance of such raindrop is $\sqrt{2} \approx 1.4142$, as shown in Figure 2.4 (top). For pixels with rays that intersect the sphere, the angle of incidence is used to evaluate an initial refraction vector. This initial refraction vector travels inside the raindrop and eventually reaches the other side, refracting again when

exiting. We are interested in this outgoing refracted vector which we store in the vector mask. Refraction is evaluated based on Snell's law with the refraction coefficients of air and water, 1.00 and 1.33 respectively. The reflected vector and its corresponding Fresnel coefficient are also calculated and stored in the mask.

However, a single vector mask is not sufficient to properly estimate refraction of raindrops at varying distances. As the distance from the camera to the raindrop increases the refracted vectors smoothly vary but alters the raindrop's appearance significantly as demonstrated in Figures 2.4 (bottom) and 2.5. To address this problem we build mip-map levels based on this first case, increasing the distance of raindrops accordingly. Every subsequent mip-map level has one-fourth the size of its predecessor, and so will be the projected area. The same trigonometric relation used to position the first raindrop is applied to generate the subsequent mip-map levels, each level having decreasing aperture angles (refer again to Figure 2.4). The aperture angle θ_i of a given mip-map level i can be determined by $\arctan(\frac{\tan(\theta_{i-1})}{2})$, since $\frac{\tan(\theta_{i-1})}{2}$ gives a relative measurement of half of the previous projection plane size. This process is repeated until we end up with the lowest mip-map level of 1 pixel. For convenience, refer to Table 2.1 for the resolutions and distances that we used.

Mip-Map Resolution	Aperture Angle θ	Raindrop Distance
1024	45°	1.4142
512	26.5651°	2.2361
256	14.0362°	4.1231
128	7.1250°	8.0623
64	3.5763°	16.0312
32	1.7899°	32.0156
16	0.8952°	64.0078
8	0.4476°	128.0039
4	0.2238°	256.0020
2	0.1119°	512.0010
1	0.0560°	1024.0005

Table 2.1: Hierachical map generation details for quick reference.

Although mip-mapping itself reduces aliasing artifacts, quality can be improved around the raindrop's mask by using multi-sampling strategies. Pixels that effectively intersect the raindrop receive an alpha value of 1 while pixels that do not will have an alpha of 0. Pixels that lie on the edge of the raindrop are multi-sampled, assigning an appropriate alpha value between 0 and 1 based on pixel coverage. For our results, we used 32 randomly generated sub-samples inside the pixel's area. Once the masks are produced, they are ready to be used in the run-time stage. Figure 2.5 illustrates all the mask's mip-map levels.

2.4.3 Run Time

During run-time raindrops are rendered as screen aligned billboards. The hierarchical vector mask precomputed early is stored as mip-map textures to be mapped onto billboards, and their contents adjusted for each raindrop in run-time.

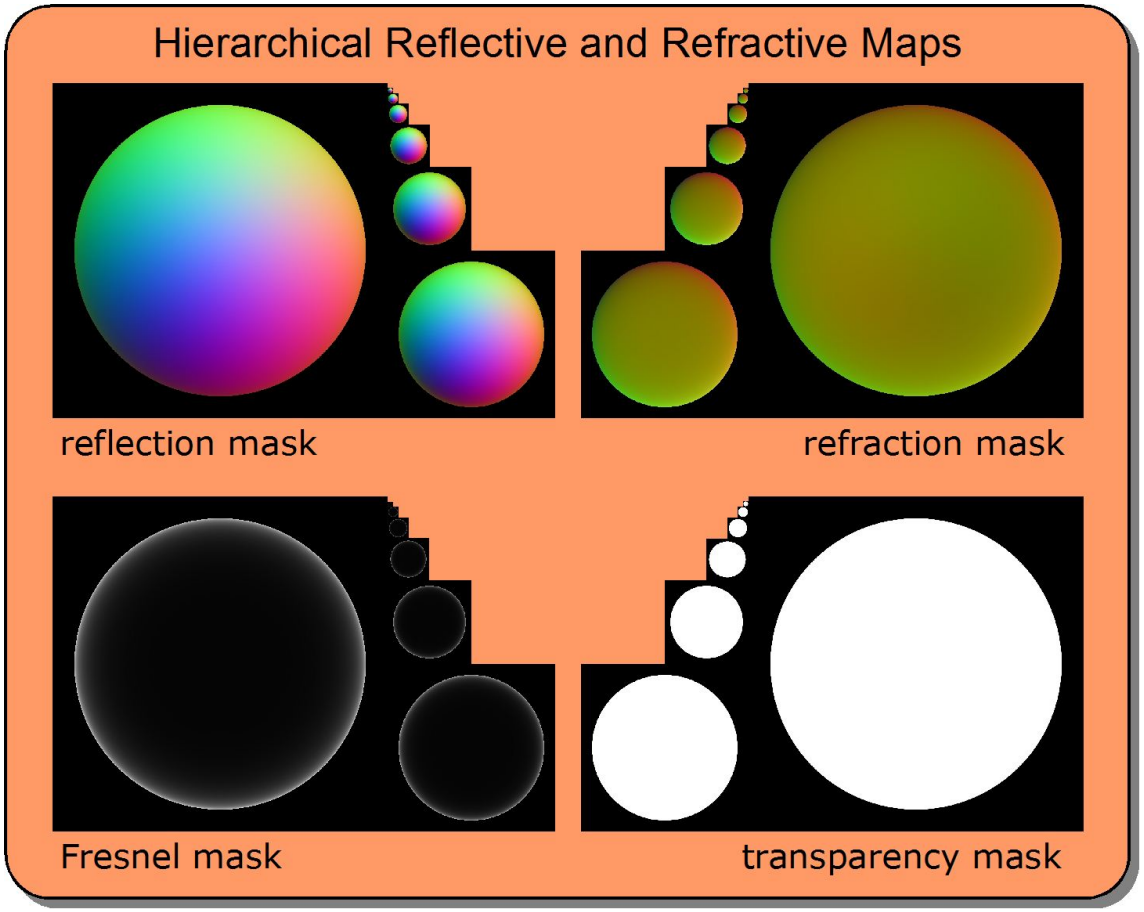


Figure 2.5: Fake color representation of the hierarchical maps generated in the preprocessing step. The transparency mask is the multi-sampled alpha mask discussed in the end of Section 2.4.2. Although the refraction layers seem similar in this fake-color representation, their variance have a strong impact in the resulting raindrop's appearance.

Each fragment of the billboard holds a normalized texture coordinate (s,t), which is used to retrieve the reflected and refracted vectors in the mask's texture, as well as the alpha and Fresnel coefficients. Remember that the original texture mask was rendered looking toward the $-$Z axis with the viewpoint located at the origin with the up-vector aligned to world's $+$Y axis, and each mip-map level used a distinct distance. In runtime, however, raindrops can be anywhere, requiring reflection and refraction values from distances that may not correspond to the preprocessed ones. Since mip-map levels were carefully generated by maximizing the raindrop's projected area, the proper correspondence between the silhouette of the raindrop within all the levels is insured; arbitrary distances can be used, and the hardware's mip-map texture-filter will provide an automatic and graceful interpolation between distances, based on the size of the billboard.

Viewing orientation and relative position of the raindrop to viewpoint are also prone to alter reflection and refraction. We address this problem by appropriately rotating the vector data in the mask. A rotation matrix R is derived at the vertex processing stage and forwarded to be used on each of its generated fragments as follows:

$$R = [\ \vec{u}\ \ \vec{v}\ \ \vec{n}\ ] = \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix} \tag{2.1}$$

where the matrix's components are defined as:

$$\vec{n} = normalize(C - P) \tag{2.2}$$

$$\vec{u} = \vec{V} \times \vec{n} \tag{2.3}$$

$$\vec{v} = \vec{n} \times \vec{u} \tag{2.4}$$

with C being the viewpoint, P the raindrop's position and $\vec{V}$ the viewer's up vector. Note that the rotation matrix becomes the identity when the viewing condition matches the one of preprocessing.

Once the vectors are properly adjusted, they are used to fetch color in the environment map. Having mip-map levels in the environment map texture is preferable in order to account for the solid angle subtended by each ray sample. The resulting color of the fragment of the raindrop is then given by modulating the reflected and refracted contributions accordingly with its corresponding Fresnel coefficient. The equation works similarly to alpha blending:

$$Frag_{color} = f * E(\vec{L}) + (1 - f) * E(\vec{R}) \tag{2.5}$$

where f is the Fresnel coefficient, $E(\vec{v})$ is the environment map and $\vec{L}$, $\vec{R}$ are the resulting adjusted reflection and refraction vectors respectively. Finally, the alpha component is assigned to the fragment's alpha value and submitted to the next stage of the pipeline, where alpha testing and blending are performed, properly changing the state of the framebuffer.

The run-time stage addresses the problem in a simple way and does not require any extra steps aside from rendering the raindrops in the frame buffer. This solution can be efficiently mapped to the GPU and uses only features that graphics hardware are highly specialized in: point sprites, mip-map based texture filtering, and matrix-vector multiplication. The complete run-time shader tree is presented in Figure 2.6. Refer to Section 2.5 for implementation details and Section 2.6 for results.

2.5 Implementation Details

The hierarchical masks can be accommodated on the GPU using two $RGBA$ float-precision mip-mapped textures: one could store reflection's XYZ components into the RGB components of one texture while reserving A for Fresnel coefficient; similarly for refraction's components and transparency into another texture. Using 32bit precision float textures require around 44MB of memory for a 1024x1024 resolution base hierarchy level, but half-precision representation is sufficient since all components lie within the range $[-1, +1]$, reducing the graphics memory requirements by half.

As for billboard generation, since each mip-map level has the same texel resolution in both dimensions they are square in shape. This leads to a common feature of graphics hardware: point sprites. This simplifies the rendering process and alleviates both geometrical throughput and calculations in the vertex transform stage, since we do not need to issue quads (4 vertices) and align them with the screen. Although GPUs often have limitations regarding the maximum size of point sprites, this constraint has been relaxed in

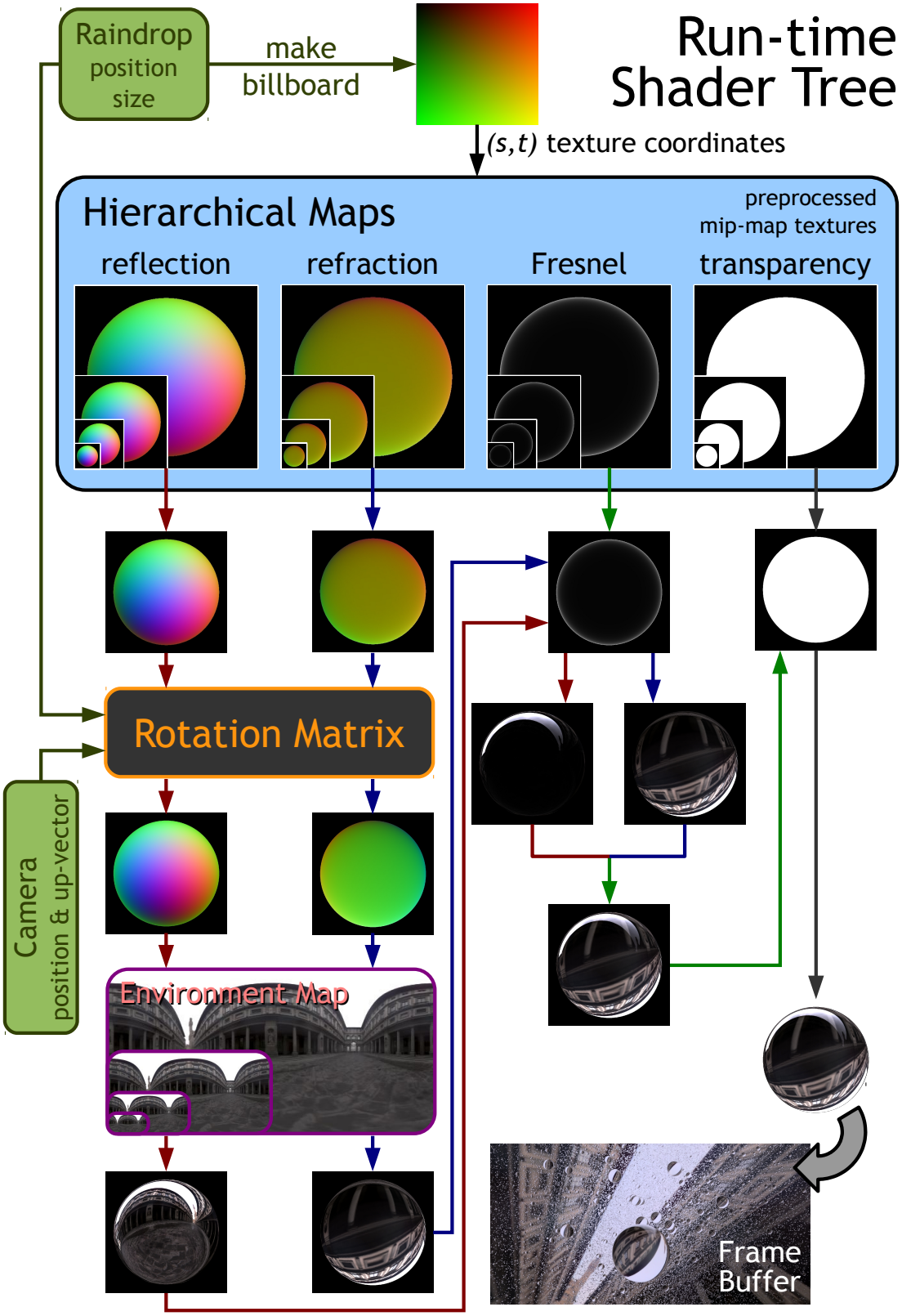


Figure 2.6: Run-time shader tree as described in Section 2.4.3.

recent programmable graphics hardware, but this forces the intended size to be explicitly specified in the vertex shader. The proper screen size of a raindrop can be determined by its corresponding distance to the camera and the base level texture resolution, in a similar process to the one used to find the raindrop's distances based on the camera's aperture angle in the preprocessing stage (see Figure 2.4).

There is an issue regarding point sprites in hardware that does not support OpenGL 3.0. When rendering point sprites to a framebuffer object one should set the point sprite coordinate origin, `GL_POINT_SPRITE_COORD_ORIGIN`, to `GL_LOWER_LEFT`; when rendering directly to the color buffer it should be set to `GL_UPPER_LEFT`. Any other configuration will make OpenGL transform the point sprites in the CPU. Additional details on the subject can be found in [GPGPU Forum (2006)].

We update the raindrops' positions on the GPU and for simplicity our simulation assumes that all raindrops have the same velocity and do not collide with each other, analogously to the process used by Rousseau *et al.* [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)]. More sophisticated particle simulation techniques exist [KOLB; LATTA; REZK-SALAMA (2004); KIPFER; SEGAL; WESTERMANN (2004)] but are out of the scope of this paper. We made use of transform-feedback, a recent feature of graphics hardware as well as the traditional render-to-texture copy-to-vertexbuffer approach for use in older hardware. For the rain volume we used a rain-box similar to Rousseau *et al.* [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)]. Each side of the rain-box extends in each direction by the maximum raindrop distance given in Table 2.1, roughly 1025. As raindrops leave the rain-box in the $-$Y direction we relocate them to the top of the rain-box.

For tone-mapping, we used the global variant of the photographic operator [REINHARD et al. (2002)] because of its robustness and simplicity. Details about its implementation are out of the scope of this paper but implementations of it are widely available in popular graphics software development kits. The interested reader is referred to [DEVLIN (2002)] for a survey of several tone-mapping operators, and to [GOODNIGHT et al. (2005); KRAWCZYK; MYSZKOWSKI; SEIDEL (2005); SLOMP; OLIVEIRA (2008)] for real-time implementations of the more attractive, but computationally expensive, local variant of the photographic operator.

2.6 Results

Performance results are summarized in Table 2.2 for three hardware configurations. All results are measured with a screen resolution of 1024x1024 pixels and make use of the transform-feedback simulation approach (except for Architecture 3 where render-to-FBO copy-to-VBO was used). The software was implemented with Microsoft's Visual C++ Express 2008 and the shaders were implemented with the OpenGL Shading Language (GLSL) version 1.20. Details regarding each configuration are presented below:

- Architecture 1 (desktop): GeForce GTX 280 1024MB VRAM (NVIDIA Driver 191.07 WHQL) on a Intel Core2 Quad 32bit CPU Q9400 2.66GHz with 4GB RAM running Windows Vista Home Premium 32bit (SP1);

- Architecture 2 (notebook): GeForce 8600M GT 256MB VRAM (NVIDIA Driver 186.81 WHQL) on a Intel Core Duo 32bit CPU T7250 2GHz with 4GB RAM running Windows Vista Home Premium 32bit (SP1);

- Architecture 3 (desktop): GeForce 7300 GT 256MB VRAM (NVIDIA Driver 191.07 WHQL) on a AMD Opteron Dual Core 64bit Processor 285 2.59GHz with 2GB RAM running Windows XP 32bit (SP3).

	Architecture 1		Architecture 2		Architecture 3	
Raindrops	FPS-T	FPS	FPS-T	FPS	FPS-T	FPS
125000	595	794	99	151	55	110
250000	430	540	75	105	45	76
500000	250	295	50	61	29	40
1 million	140	157	33	38	20	24
2 million	74	83	21	23	10	12
4 million	43	46	8	9	5	5

Table 2.2: Frame rates using our method. All measurements account for animation time as well. Screen resolution of 1024x1024 pixels. FPS-T denotes tone-mapping.

As can be seen, real-time performance is sustained even for large quantities of raindrops in current graphics hardware. In comparison with Rousseau *et al.* [ROUSSEAU; JOLIVET; GHAZANFARPOUR (2006)], which runs around 100 FPS with 5000 raindrops, our method can sustain the same performance with about 20 times more raindrops. Note that Rousseau *et al.* do not deal with HDR and thus there is no tone-map performance penalty. Although we do not have the same hardware used by Rousseau *et al.*, a GeForce 6800 GT, we believe that the results from Architecture 3 are fair for comparison since a GeForce 6800 GT has more processing units and wider memory interface bus than a GeForce 7300 GT; therefore we would expect our method to perform even better in a GeForce 6800 GT. A rather recent and detailed performance comparison chart between these GPUs can be found at [Tom's Hardware (2008)].

Additional result images are presented in Figure 2.7. The presented technique is robust enough to allow high-quality extreme close-ups of raindrops. One might argue this does not seem realistic but there are several efficient ways to hide such large raindrops in practical applications. We show the large drops to demonstrate the range of our approach. The most straight-forward method to remove excessively large drops would be to move the camera's near plane further away, effectively clipping such large droplets. Another possibility would be to gradually increase the transparency factor in the fragment shader, so closer drops have higher levels of transparency until they reach the near-clip plane.

In a real scenario, however, raindrops that are that close to the eye (or camera) do not simply disappear, but would be out of focus. One possibility is to use derivative texture access on the environment map, blurring their internal appearance in an efficient manner. A more sophisticated approach is simulating depth-of-field. Discussions on these techniques are out of the scope of this work but there is much computer graphics literature in regards to real-time post-processing techniques to reproduce such effects [DEPTH OF FIELD: A SURVEY OF TECHNIQUES (2005); FILION; MCNAUGHTON (2008a); LEE; KIM; CHOI (2008); LEE; EISEMANN; SEIDEL (2009)].

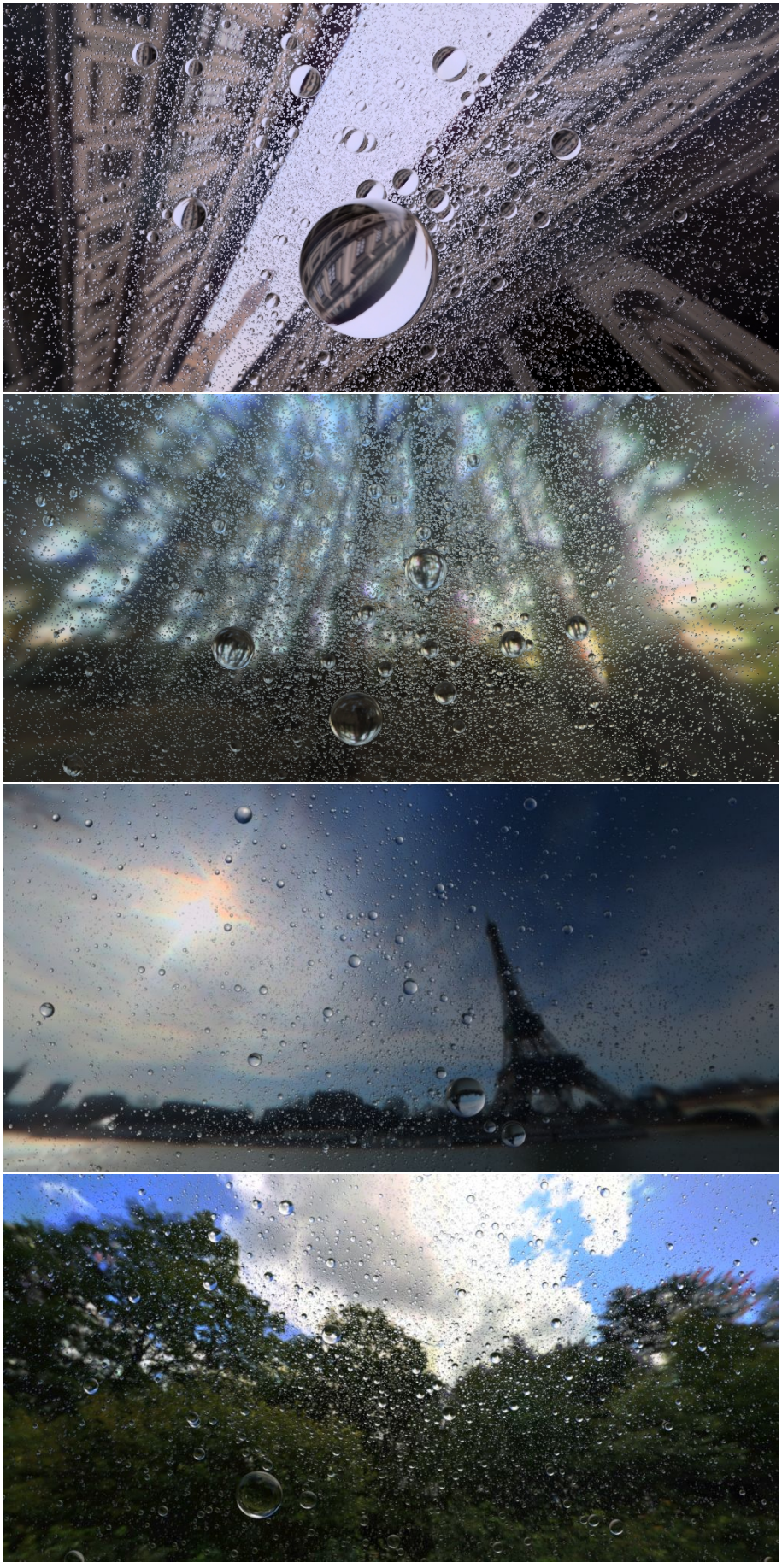


Figure 2.7: Additional results rendered using the proposed technique in different HDR environment maps, viewing conditions, and number of raindrops. Results account for tone-mapping.

2.7 Limitations

Since our method considers only the environment map, any object not represented in the environment map is ignored. To alleviate this problem many applications generate environment maps with 3D objects imposed on them based on the position of the viewpoint. This approach usually provides convincing results, but it is worth noting that it is a rough approximation of position.

Raindrops are not sorted so the presented multi-sampled alpha blending improvement is not guaranteed to work properly. However we did not notice any unpleasant artifacts in our experiments since most raindrops within the rain-box correspond to small screen-space sizes. A cheap way to sort them would be to store raindrops in the vertex-buffer according to their position inside the rain-box. This is however a palliative solution and does not solve the problem as the viewer moves freely inside the box.

Finally, the presented method makes use of square shaped billboards, which are unable to reproduce perspective distortions. Needless to say that point sprites are not a requirement, although attractive due to their performance benefits; they could be replaced with spherical billboards by drawing quadrilateral primitives for each raindrop or performing vertex amplification with the aid of the geometry shader. However, such distortions are rare, being noticed only in raindrops extremely close to the viewpoint.

2.8 Conclusion

We propose a straight forward method for rendering raindrops in real-time based on their optical properties. Our results are comparable to ray-traced solutions, providing high image quality which has not been possible using rain streak based approaches. Furthermore our technique runs entirely on the GPU, exploiting some of its well-known features allowing real-time performance with large quantities of raindrops. We believe that the current limitations are justified by the trade-off between high image quality and high performance.

We also believe that the presented method is suitable for combining with other methods which render the wide variety of rain's behaviors: splashing, sliding down surfaces, etc. We also believe that our method can benefit the film industry by providing instant feedback to artists designing rainy scenes. Finally we feel this method provides an attractive solution for rendering raindrops in game and graphics engines where temporal interactions like slow-motion replay and paused simulation effects are desirable. Some examples include: driving simulators where raindrops on the front glass do not necessarily move at high velocities, and sports/fighting games where slow motion replays are a trend.

2.9 Future Work

In this section we summarize a few future directions. First, hierarchical map storage could be eliminated by deriving a function parametrized by texture coordinates and relative distance to the viewer which would determine outgoing reflection and refrac-

tion vectors, although it could introduce an expensive overhead. Next would be to allow vector mask deformation thus allowing arbitrary shapes, and possibly allowing raindrops to collide and merge. Additionally a more perceptually-driven generation of raindrop's refractive and reflective maps could be derived using known limitations of the human vision system, such as its maximum and minimum field-of-view and focal distance. Another possible investigation would be the use of dynamic, variable-size filtering on the contents of the masks to simulate depth-of-field effects or rain-streak patterns; such varying size filtering can be efficiently performed through the use of high-order summed-area tables [CROW (1984); HECKBERT (1986a)]. Lastly our method could create underwater air bubbles by inverting the refraction coefficients, but determining if air bubbles are best represented by spheres requires more experimentation.

2.10 Related Publications

SLOMP, M. et al. Photorealistic real-time rendering of spherical raindrops with hierarchical reflective and refractive maps. **Journal of Computer Animation and Virtual Worlds (CAVW)**, [S.l.], v.22, p.393–404, August 2011

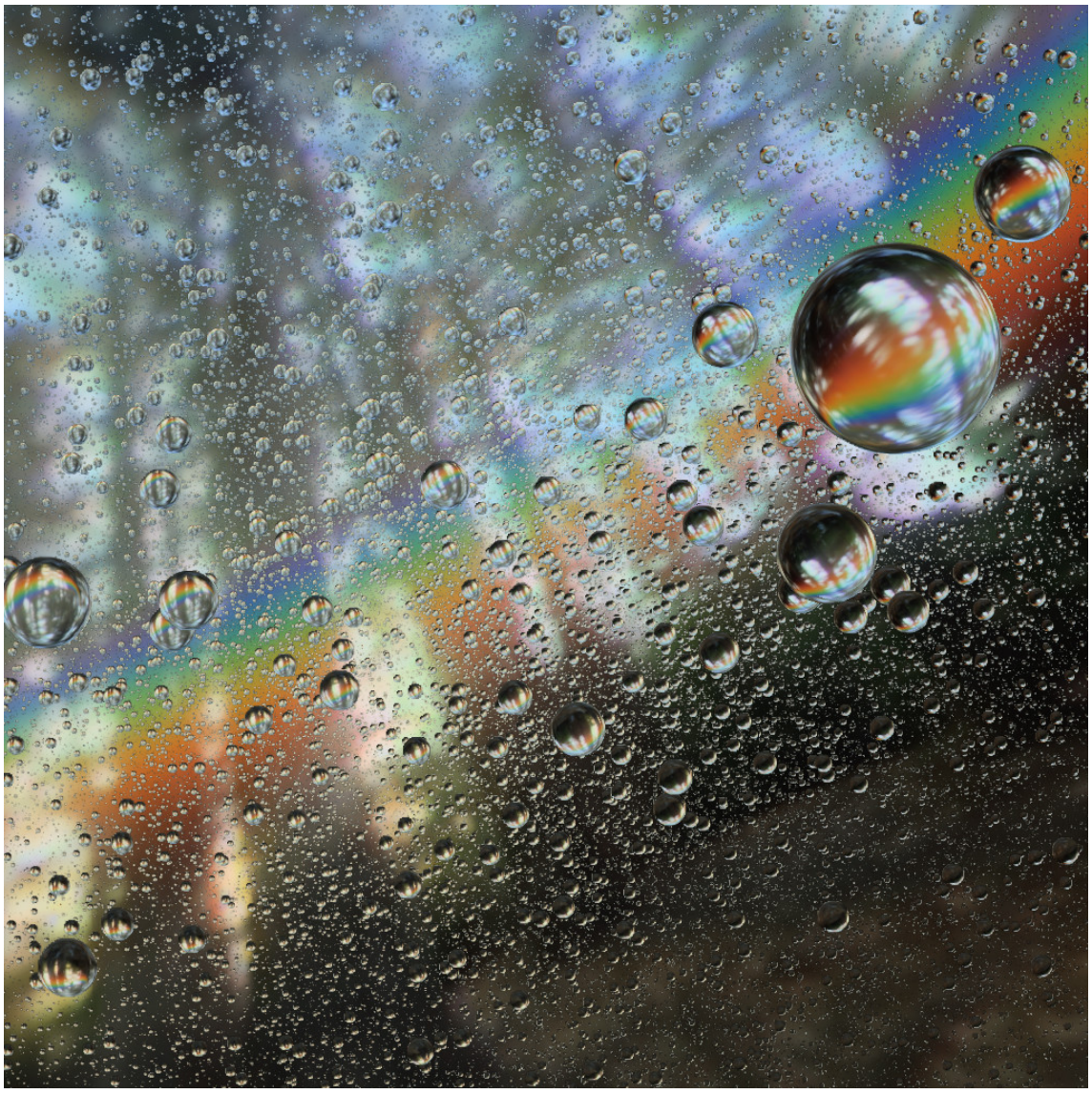
SLOMP, M. et al. Photorealistic Real-time Rendering of Spherical Raindrops with Hierarchical Reflective and Refractive Maps. In: ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES (I3D), New York, NY, USA. **Proceedings...** ACM, 2010

Marcos Slomp, Matthew W. Johnson, 玉木徹, 金田和文: "Photorealistic Real-time Rendering of Spherical Raindrops with Hierarchical Reflective and Refractive Maps", Visual Computing/グラフィックとCAD合同シンポジウム2009 予稿集, 旭川市勤労者福祉総合センター, 北海道(2009 06).

2.A Appendix A: Computer Graphics Forum (CGF) 2011 Cover Image Contest Finalist

Fantastic Rainy Scene

Marcos Slomp, Shota Kanamori and Kazufumi Kaneda
Graduate School of Engineering, Hiroshima University, Japan



Raindrops are rendered using a texture-based approach [SLOMP et al. (2011)], each treated as a point-sprite whose size varies based on the distance from the camera. Each produced fragment samples over a preprocessed reflective and refractive hierarchical map, retrieving canonical reflection and refraction vectors, along with Fresnel coefficient and multi-sampled transparency. Non-transparent fragments will have their sampled reflected and refracted vectors further transformed by a rotation matrix - derived based on the raindrop's attitude relative to the canonical camera used to preprocess the hierarchical maps - which then sample an environment map. The final color contribution is determined after proper modulation with the Fresnel coefficient.

The rainbow mask is rendered by using a ray casting method during preprocessing. The method accounts for wave optics and the distribution of raindrops [KANAMORI et al. (2010)]. These efforts enable the method to render the visual variations of the three types of rainbows: the primary and the secondary rainbows, and supernumerary rainbows.

A dynamically generated cube-map is rendered every frame by copying the respective cube-map faces from a selected static environment map and then composed with the rainbow mask through additive blending, all on GPU. This allow changes on the rainbow properties such as location, brightness and size without perturbing the original static cube-map. Raindrops' fragments samples from this dynamic environment map so that the rainbow is presented properly within the raindrops. The position of the raindrops are also updated every frame and done entirely on the GPU, either through transform-and-feedback buffers, or via classic render-to-texture copy-to-vertex-buffer approach.

Finally, if an HDR environment map is being used, tone-mapping is performed using the Photographic Operator [REINHARD et al. (2002)], thus compressing the wide luminance range of the scene into the displayable range of the device. The overall performance of the technique exceeds 70 FPS on a GeForce GTX 280 with full-screen image resolution of 1920x1200 pixels and 2 million raindrops.

3 GPU-BASED SOFTASSIGN FOR MAXIMIZING IMAGE UTILIZATION IN PHOTOMOSAICS

3.1 Abstract

Photomosaic generation is a popular non-photorealistic rendering technique, where a single image is assembled from several smaller ones. Visual responses change depending on the proximity to the photomosaic, leading to many creative prospects for publicity and art. Synthesizing photomosaics typically requires very large image databases in order to produce pleasing results. Moreover, repetitions are allowed to occur which may locally bias the mosaic. This chapter provides alternatives to prevent repetitions while still being robust enough to work with coarse image subsets. Three approaches were considered for the matching stage of photomosaics: a greedy-based procedural algorithm, simulated annealing and SoftAssign. It was found that the latter delivers adequate arrangements in cases where only a restricted number of images is available. This chapter also introduces a novel GPU-accelerated SoftAssign implementation that outperforms an optimized CPU implementation by a factor of 60 times in the tested hardware.

3.2 Introduction

As opposed to photorealistic rendering, non-photorealistic rendering (NPR) algorithms trade-off physical accuracy in exchange for feature highlighting or artistic effects. The current affordability of computers and digital cameras is empowering many inventive outcomes from user-generated content, paving access to NPR effects like photomosaics [MEIER (1996)].

Photomosaics comprise special instances of mosaics. A mosaic is a stylization of an image, consisting of a collection of bulky primitives. A photomosaic is then envisaged as a mosaic whose bulky primitives are images themselves, as illustrated in Figure 3.1. Over the past decades several photomosaic generation algorithms were conceived [SILVERS (1997); KLEIN et al. (2002); BLASI; PETRALIA (2005)].

Figure 3.1: A 20×20 photomosaic assembled from 1500 images assigned through one of the proposed algorithms (simulated annealing). The input image is miniaturized at the left for reference. The usage of available images was maximized so that no image appears repeatedly in the resulting mosaic.

Subjects experience different visual responses based on their relative proximity to a photomosaic. Despite of being a fancy effect, photomosaics are widely acclaimed by advertisement producers. In the hands of skilled marketing personnel, photomosaics can capture the essence of a product at different perspectives, ultimately entertaining and attracting potential consumers. Many commercial opportunities can arise through clever utilization of photomosaics.

Silvers *et al.* delivered the first efforts on photomosaic generation [SILVERS (1997)]. Klein *et al.* extended the concept to *video mosaics*, where lower resolution movies are

used as bulky primitives to assemble a larger video [KLEIN et al. (2002)]. Di Blasi *et al.* derived a faster method to generate photomosaics by varying the sizes of the images that appear in the photomosaic, clustered into an *antipole tree data structure* [BLASI; PETRALIA (2005)]. The interested reader can further refer to Battiato *et al.*'s survey on photomosaic generation techniques [BATTIATO et al. (2007)].

Current photomosaic generation algorithms, however, require huge amounts of images in order to produce attractive results. Repetitions are allowed to occur, which may locally bias the mosaic. Another common strategy, if only a small image set is available, is to reduce the size of the bulky primitives, subdividing the input image into very small chunks, but this ends up breaking the illusion of the photomosaic, unless observed very closely. Tuning photomosaics manually is obviously exhaustive and impractical.

This chapter investigates three methods to maximize the usage of available images in photomosaics by preventing image repetitions: a greedy-based procedural algorithm, a simulated annealing driven solution [KIRKPATRICK; GELATT.; VECCHI (1983)] and a SoftAssign-based [GOLD et al. (1997)] approach. From these strategies, the greedy-based is the fastest, but lacks in quality; simulated annealing, on the other hand, is time-prohibitive. SoftAssign is capable of producing photomosaics that are qualitatively equivalent to simulated annealing in a much faster rate, but still time demanding nonetheless. Additionally, building upon this investigation, this chapter also introduces an efficient GPU-based implementation of SoftAssign. The performance increase of the proposed GPU-based SoftAssign is of about 60 times when compared to an optimized CPU implementation in the tested hardware configuration.

The next three sections of this document present the studies developed on the above mentioned optimization strategies in the context of photomosaic optimization: Section 3.3 formalizes the problem of maximizing the number of images used in photomosaics; Section 3.4 reviews the three image assignment approaches; Section 3.5 provides initial results.

The remaining sections of this document are laid out based on the analysis of the referred research, paving the road to further investigation of the SoftAssign algorithm in GPU: Section 3.6 provides a blueprint to implement SoftAssign and discusses issues and parallelization strategies, Section 3.7 focuses on mapping the SoftAssign algorithm to the graphics pipeline in order to exploit parallelism in GPU, Section 3.8 extends the initial results with additional mosaics and performance comparisons between CPU and GPU implementations of SoftAssign, and Sections 3.9 and 3.10 conclude the paper and points future directions.

3.3 Maximizing Image Utilization on Photomosaics

In general, given a source image I and a set of n images $T = \{t_1, ..., t_n\}$, ordinarily known as *tiles*, synthesizing a photomosaic P digests to:

1. Subdivide the target image into a regular $w \times h$ lattice of rectangular regions; these regions will be referred to as *patches* and denoted as $p_i \in \{p_1, ..., p_m\}$, where $m = w \times h$;

2. For each patch p_i, search for an appropriate tile $t_j \in T$ to replace the patch; this step may require manipulation of the tile images (resizing, cropping, etc.)

The goal is to maximize the usage of distinct tiles in the resulting photomosaic. Tile utilization can be maximized by preventing any individual tile to be assigned to more than a single patch. That means that a tile can only be assigned to a patch if it is not currently assigned to any other patch. In the end, every patch must hold a unique tile assigned to it, although it is possible for some tiles to remain unassociated to any particular patch, since typically there are more tiles than patches. When there are more patches than tiles, no solution can be determined.

However, maximizing tile utilization itself does not guarantee that the resulting photomosaic P will resemble the input image I. Therefore, the *visual similarity* between patches and tiles can not be neglected, being critical to guide the optimization process. The metric chosen to determine visual similarities is a simple Euclidean distance in RGB color space. More sophisticated metrics or color spaces could be used instead, but such simplistic measure proved to be satisfactory.

In order to determine the visual similarity between a patch p_i and a tile t_j, both are further partitioned into smaller $u \times v = s$ rectangular regions: $p_i = \{p_i^1, ..., p_i^s\}$ and $t_j = \{t_j^1, ..., t_j^s\}$. The average intensity of each partition, $\overline{p}_i^k = (\overline{R}_i^k, \overline{G}_i^k, \overline{B}_i^k)$ and $\overline{t}_i^k = (\overline{r}_j^k, \overline{g}_j^k, \overline{b}_j^k)$, is then computed through simple component-wise arithmetic average. Finally, the visual similarity function can be defined as:

$$d(p_i, t_j) = \sqrt{\sum_{k=1}^{s} \left[(\overline{R}_i^k - \overline{r}_j^k)^2 + (\overline{G}_i^k - \overline{g}_j^k)^2 + (\overline{B}_i^k - \overline{b}_j^k)^2 \right]} \tag{3.1}$$

Note that the lower the value of the visual similarity function, the more closely tile t_j resembles patch p_i; a "perfect" match would evaluate $d(p_i, t_j)$ to *zero*. Therefore it is sound to think of $d(p_i, t_j)$ as a *distance* function. Such formulation will later be used to populate a *distance matrix*, a fundamental component of all of the involved algorithms (Section 3.4.1, Figure 3.2).

The problem of maximizing image utilization on photomosaics can then be formalized as an optimization scheme for minimizing the sum of $d(p_i, t_j)$ under the restriction that all patches must have uniquely assigned tiles, as formulated below:

$$min \sum_{i=1}^{m} \sum_{j=1}^{n} d(p_i, t_j)\ y(p_i, t_j) \tag{3.2}$$

subject to the following constraints:

$$y(p_i, t_j) = \begin{cases} 1 & \text{if } t_j \text{ is assigned to } p_i \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

$$\sum_{j=1}^{n} y(p_i, t_j) = 1 \tag{3.4}$$

$$\sum_{i=1}^{m} y(p_i, t_j) = 1 \quad \text{or} \quad \sum_{i=1}^{m} y(p_i, t_j) = 0 \tag{3.5}$$

The membership function $y(p_i, t_j)$ models the patch-tile assignment by assuming the binary values defined in Equation 3.3. The constraint from Equation 3.4 ensures that a patch will always have a single tile assigned to it, while the final constraints from Equation 3.5 guarantee that tiles are assigned *at most* once. All three algorithms detailed in the subsequent section enforce these requirements.

3.4 Photomosaic Optimization Strategies

The photomosaic optimization strategies investigated can be summarized as follows:

- Greedy-based Search: locates the most similar matches and sequentially revamps repetitions with unmatched ones; likely to fall into local minima solutions.
- Simulated Annealing: attempts to reach a solution close to the global optimum photomosaic through stochastic minimization algorithm that avoids local minima.
- SoftAssign: similar to simulated annealing, but the solution narrows down to the global optimum through a deterministic process that also avoids local minima.

All of these algorithms share a common resource, the *distance matrix*, which will be introduced beforehand.

3.4.1 The Distance Matrix

The purpose of the distance matrix is to track the color similarities (distances) amongst each patch p_i and each tile t_j. If we let the rows correspond to the $m = w \times h$ patches and the columns to the n available tiles, then the distance matrix $D^{m \times n}$ can be expressed as:

$$D = d_{ij} = \begin{bmatrix} d_{11} & \cdots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{m1} & \cdots & d_{mn} \end{bmatrix} \tag{3.6}$$

where $d_{ij} = d(p_i, t_j)$, according to Equation 3.1. The distance matrix is also depicted in Figure 3.2. All of the subsequent algorithms use the distance matrix as an input, and it can be computed in advance. The distance matrix is *immutable*: none of the algorithms ever modify its contents.

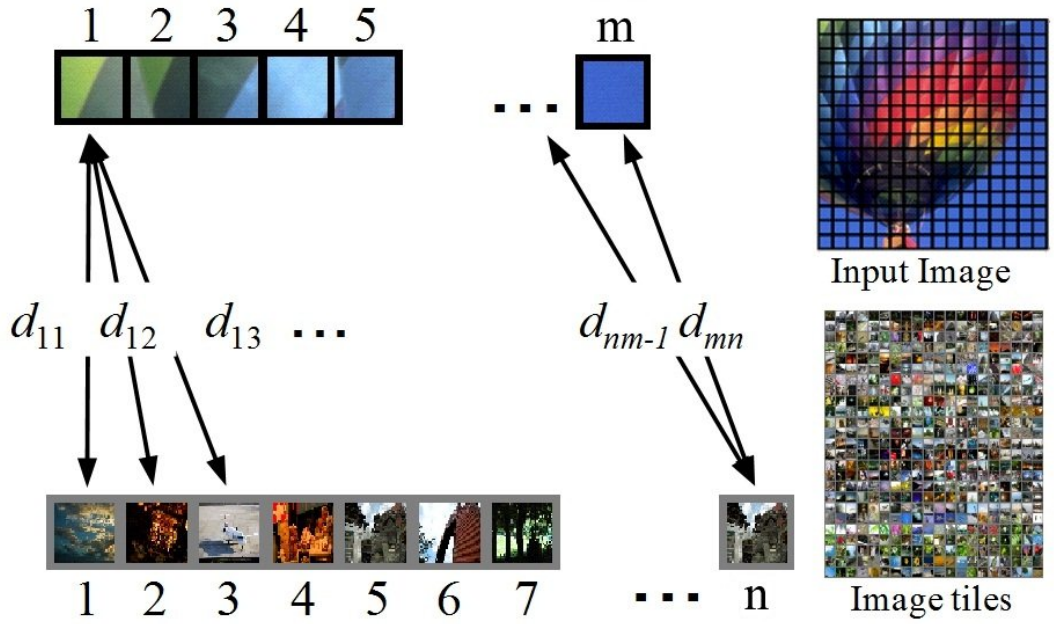


Figure 3.2: The distance matrix stores the color similarities between patches and tiles.

3.4.2 The Greedy-Based Search

The greedy approach begins by assigning tiles to patches based on the "best-match" criteria. Since repetitions are likely to occur with such criteria, the algorithm iterates once more, sequentially reassigning unused tiles to conflicting patches based once again on the same criteria.

For each patch p_i, the corresponding row of the distance matrix $D_i = [\ d_{i1}\ \cdots\ d_{in}\ ]$ is examined, searching for the element d_{iq} that holds the lowest distance (most similar). The tile t_q is then assigned to p_i as the best-match.

As repetitions are prone to happen, each tile t_j is further classified as: *assigned once*, *assigned multiple times* and *not assigned*. The rationale is then to keep uniquely assigned tiles unchanged, while replacing multi-assigned tiles by unassigned ones. This can be done by means of the absolute difference between each multi-assigned tile t_{multi} and each unassigned tile t_{free}; whichever has the lowest difference wins (Figure 3.3).

3.4.3 Simulated Annealing

Simulated Annealing (SA) [KIRKPATRICK; GELATT.; VECCHI (1983)] is a generic stochastic technique for optimization problems, identifying potential solutions through random inspection of large search spaces. SA is relatively easy to implement and can account for almost any objective function, with constraints being attached directly into the SA procedure. SA is capable of finding very close approximations to the global optimum if given enough time.

The key component of SA is the dynamic *temperature* parameter. Higher temperatures permit configurations in search space that lead to an increase in the cost function,

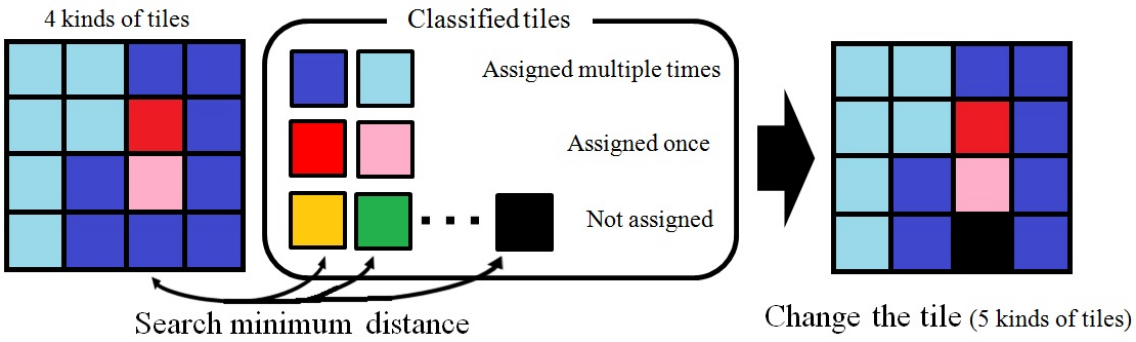


Figure 3.3: Greedy-Based Search: The corresponding patches of a multi-assigned tile are sequentially replaced by unassigned tiles until that multi-assigned tile becomes uniquely assigned. The reassignment process is based on the best-match criteria, i.e., lowest distance (highest visual similarity).

helping the algorithm to avoid local minima. As the temperature lowers, the solution is progressively enhanced and the search becomes more restricted.

Photomosaic tile matching can be expressed as a SA process through the minimization of the following cost function:

$$E(X, D) = \sum_{i=1}^{m} \sum_{j=1}^{n} (d_{ij} - \alpha)\, x_{ij} \tag{3.7}$$

where x_{ij} are elements of the *correspondence matrix* X which only takes binary values: if t_j is a possible match to p_i then $x_{ij} = 1$, otherwise $x_{ij} = 0$. The constraints from Equations 3.3 to 3.5 must also be respected; therefore every row $X_i = [\ x_{i1} \ \cdots \ x_{in}\ ]$ should hold *exactly* one element $x_{iq} = 1$, while every column of X should have *at most* one element set to 1. The semantics of the correspondence matrix X is thus equivalent to the semantics of the membership function $y(p_i, t_j)$ as expressed through Equations 3.3 to 3.5. The parameter α is used to favor a certain range of visual similarities (distances) between patches and tiles.

In order to minimize Equation 3.7, the correspondence space is stochastically sampled via a Markov process. A new solution X^* is produced at each iteration by modifying only a single row of X. This row $X_r = [\ x_{r1} \ \cdots \ x_{rn}\ ]$ is selected randomly. The single element currently set to 1 in X_r, $x_{rb} = 1$, is flipped in X_r^* so that $x_{rb}^* = 0$, thus leaving the entire row X_r^* filled with zeros. What remains to be done is to pick some aleatory element $x_{rw} \mid w \neq b$ and flip it in X_r^*, yielding $x_{rw}^* = 1$. This last step should be performed carefully to prevent the same tile t_w of being assigned twice in the new solution X^*. The index w is repeatedly shuffled if necessary until no such assignment conflict occurs.

For the particular case of *square* correspondence matrices, the procedure described above fails since a conflict-free element $x_{rw} \mid w \neq b$ would never be found. In such special case, another aleatory row $X_{\hat{r}} \mid \hat{r} \neq r$ is selected and swapped with X_r, thus making $X_r^* = X_{\hat{r}}$ and $X_{\hat{r}}^* = X_r$.

Once a new solution X^* is established, it can be accepted or rejected according to the following transition probabilities:

$$P(X \to X^*) = \begin{cases} 1 & \text{if } \Delta E \leq 0 \\ e^{-\Delta E/\tau} & \text{otherwise} \end{cases} \tag{3.8}$$

where $\Delta E = E(X^*, D) - E(X, D)$. The rationale is that state changes are allowed as long as the cost function decreases; to prevent local minima, state may also change with increasing costs based on the current temperature τ. The temperature is gradually lowered during the stochastic search process according to the annealing schedule: the temperature τ' of the next iteration is obtained by $\tau' = f\ \tau$, with a cooling factor of $0.85 \leq f \leq 0.99$ typically.

At the end of the simulation, each row $X_i = [\ x_{i1}\ \cdots\ x_{in}\ ]$ allegedly accommodates a unique element $x_{iq} = 1$, which allots t_q as the best replacement for p_i.

3.4.4 SoftAssign

In computer vision, SoftAssign [GOLD et al. (1997)] offers a robust solution to match point clouds, ensuring unique matching criteria while still avoiding being trapped into local minima cusps. SoftAssign derives from deterministic annealing and can be seen as simulated annealing (SA) when applied under the condition of mean field approximation, i.e., it does not rely on any stochastic search.

An optimal photomosaic can be reckoned as the best unique match between tiles and patches. This naturally settles SoftAssign as an alluring solution, if readapted to minimize the following cost function E:

$$E(X, D) = \sum_{i,j=1}^{m,n} x_{ij} d_{ij} - \alpha \sum_{i,j=1}^{m,n} x_{ij} + \tau \sum_{i,j=1}^{m,n} x_{ij} \ln(x_{ij}) \tag{3.9}$$

with x_{ij} being elements of the correspondence matrix X. In contrast to SA, here X holds "fuzzy" correspondences, with $0 \leq x_{ij} \leq 1$. The rationale is that each tile t_j is a potential match to each patch p_i by some weight x_{ij}. The fuzziness is guided by the last entropy term and the current temperature τ.

The elements of X are initialized randomly with very small quantities. Subsequent iterations modify X according to the following expression:

$$x'_{ij} = e^{-x_{ij}(d_{ij} - \alpha)/\tau} \tag{3.10}$$

The temperature τ decreases at each iteration akin to SA. Equation 3.10 is obtained from Equation 3.9, optimizing the same objective function of Equation 3.9 [GOLD et al. (1997)]. The key component of this equivalence is the fact that once X' is computed, a row-column normalization through *Sinkhorn iterations* [GOLD et al. (1997); SINKHORN (1964)] is performed.

Such normalization procedure forces all rows and columns of X' to sum up to 1. Because of such behavior, columns that correspond to irrelevant tiles (unassigned) will also retain some quantities in their elements, a fact that can bias the convergence process. SoftAssign originally attached to the correspondence matrix an additional row and column, referred to as *outliers*, in order to isolate discrepant matches, discarding them

once the solution is found. For the case of photomosaic optimization, only the outlier row is necessary, thus mapping unused tiles to an imaginary patch. In contrast, an outlier column would allow patches to be assigned to an imaginary tile which, once discarded, would result in patches not being assigned to any tile.

At the end of the simulation, save for the attached outlier row, every row $X_i = [\ x_{i1}\ \cdots\ x_{in}\ ]$ should contain a unique element $x_{iq} = 1$ while all others set to zero. The tile t_q is then settled as the best candidate to replace p_i.

3.5 Initial Results

We based the current results on a hot-air balloon reference photograph (Figure 3.5, upper left). As for the color similarity function evaluation (Equation 3.1), each patch and tile was partitioned into 4×4 smaller square-shaped regions.

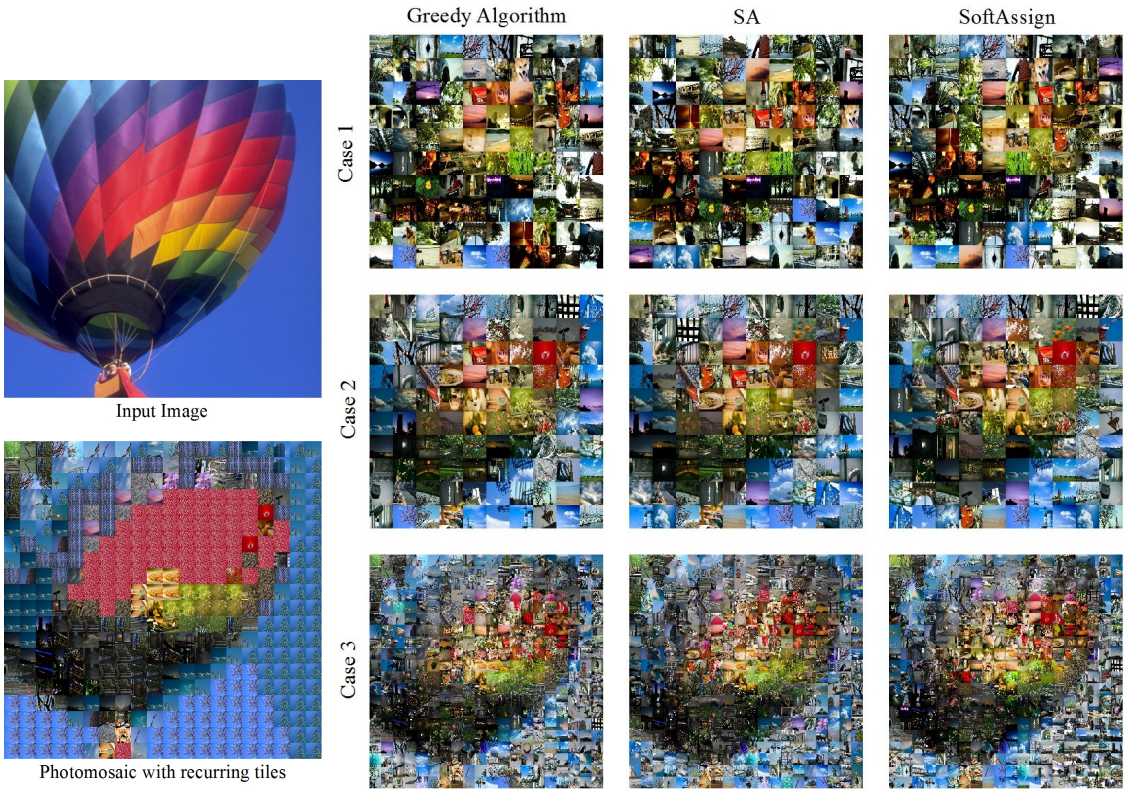


Figure 3.4: Summary of experimental results: the upper left image is the reference image; the lower left one is a 20x20 photomosaic generated from 1500 tiles (similar to Case 3) through a simple *best-match* (minimal cost) algorithm which allows tile repetition (only 82 tiles were selected; final absolute cost of 79989). Note how such recurring tiles tend to bias the photomosaic, leading to weak aesthetics. The remaining rows corresponds to one of the three studied configurations enumerated in the Experimental Results Section, and each column corresponds to one of the three tile-repetition-free discussed algorithms.

A total of three configurations were studied by varying the number of subdivided patches and the number of available tiles, as enumerated below:

1. $10 \times 10 = 100$ patches selected from 100 tiles; Figure 3.5-uppermost; (first row of Table 3.1).

2. $10 \times 10 = 100$ patches selected from 500 tiles; Figure 3.5-center; (middle row of Table 3.1).

3. $20 \times 20 = 400$ patches selected from 1500 tiles; Figure 3.5-bottom; (last row of Table 3.1).

Patches	Tiles	Measurements	Greedy	Simulated Annealing	SoftAssign
100 (10×10)	100	abs. cost	44172	43370	42713
		rel. cost	1.03	1.02	1.00
		gen. time	7s	13min	6s
100 (10×10)	500	abs. cost	31997	31282	31656
		rel. cost	1.02	1.00	1.01
		gen. time	6s	1h32min	35s
400 (20×20)	1500	abs. cost	128817	123350	127560
		rel. cost	1.04	1.00	1.03
		gen. time	3min	2 weeks	8min

Table 3.1: Summary of results. The final minimized cost (absolute and relative) of each algorithm in each case is listed, as well as the total time that each algorithm took to find the solution.

The Simulated Annealing and SoftAssign parameters were determined empirically. Their corresponding performance results in Table 3.1 were measured through our MATLAB implementation of the corresponding algorithms. The exception is the greedy-based search, which was implemented in C++, the binary being compiled and linked from the Microsoft Visual C++ 2008. The target hardware is an Intel Core2 Duo 2.5GHz with 2GB RAM running Windows XP 32bit SP3.

In the first two cases, SoftAssign and SA result in more visually appealing photomosaics, rating them as good algorithmic choices when a comparatively small number of tiles is available; although hard to perceptually decide which one feels better, SoftAssign is much faster. As for the third case, SA converges to a qualitatively lower cost than the greedy and SoftAssign approaches, but is excessively time demanding and thus impractical.

The performance and overall quality of the results originated from SoftAssign motivated us to investigate the acceleration of the algorithm with the assistance of the modern programmable graphics hardware (GPU). For that purpose, the following sections will focus on the SoftAssign method, introducing a pseudo-code template, mitigating potential issues, and clarifying decisions made while porting the technique to the GPU.

3.6 SoftAssign Implementation

The blueprint for the SoftAssign can be summarized in the following pseudo-code:

Parameters:

```
t0    : scalar, input     // The initial temperature
tk    : scalar, input     // The temperature cooling factor
alpha : scalar, input     // Bias factor for the objective function
D     : matrix, input     // The distance matrix to be minimized
S     : matrix, output    // The solution (optimal correspondence)
```

Algorithm:

```
M = matrix(#rows,#cols)
Q = matrix(#rows,#cols)
lowest = +%big
t = t0

for each temperature cooling iteration
  for each temperature stabilization iteration

    Q = M .* (D - alpha)
    M = exp(-Q / t) / sqrt(t)

    outlier_row = array[#cols] of +%tiny    // Refresh outliers
    for each Sinhkhorn iteration            // Sinkhorn method:
      M = M ./ (sum_cols(M) + outlier_row)  //   Normalize columns
      M = M ./ sum_rows(M)                  //   Normalize rows
    end-for

  end-for

  cost = sum_all(Q)       // Compute the cost (fitting)
  if (cost < lowest)      // Update solution if necessary
    lowest = cost
    S = M
  end-if

  t *= tk;                // Lower the temperature
end-for
```

The terms `#rows` and `#cols` are, respectively, the number of rows and columns of the input distance matrix `D`. The algorithm should initialize `M` with some very small random quantities. The operator `.*` denotes element-wise multiplication. The expression `exp(-Q / t)` does not denote matrix exponentiation, but a simple element-wise exponentiation. The idioms `+%big` and `+%tiny` should evaluate, respectively, to some very big and very small positive constant quantities.

The function `sum_cols(M)` results in a row-vector whose elements correspond to the sum of the respective columns in `M`. Analogously, `sum_rows(M)` yields to a column-vector holding the sum of all elements in the associated rows of `M`. As for `sum_all(Q)`, all elements of `Q` are accumulated, reducing to a single value.

The operator `./` denotes element-wise division, but such division happens in a slightly different fashion. When applied in `M ./ sum_cols(M)`, each row of `M` is divided, element-wise, by the resulting row-vector of `sum_cols(M)`. The semantics is similar for `M ./ sum_rows(M)`, but operating element-wise on columns instead.

The outlier row plays an important role: it accumulates residual weights for tiles that don't seem to fit to any particular patch. Note that the outlier row must be reassigned before triggering the Sinkhorn normalization, each element initialized with the same small value. The outlier row can be attached as an extra row to `M` and ignored during temperature stabilization and row-normalization.

The ideal number of iterations is problem-dependent and usually obtained empirically. The interested reader should refer to the original SoftAssign paper [GOLD et al. (1997)] for additional algorithmic details and parameter setup guidelines, as this is out of the scope of this paper.

3.6.1 Addressing Precision Issues

SoftAssign is prone to run into precision issues, pushing values towards infinity as they escape from the representable range of the underlying floating-point scheme. Continuing to operate on such extravagant quantities will eventually cause numerical inconsistencies which will then compromise the entire solution with no turning back.

Having the values of the distance matrix normalized into some small range is preferable. This way the SoftAssign algorithm is unlikely to run into precision issues. However, normalization itself may cause accuracy losses if the fractional part can not be accommodated properly in the underlying representation. When the normalization process is not able to cope with the accuracy required, extra care is necessary in order to prevent precision issues during the algorithm execution. In order to avoid such precision pitfalls in an elegant and efficient way, it is important to understand how and were they can potentially happen.

As the solution converges, individual values of the matrix `M` will approach one. When this happens, the subsequent updates of the associated values in the `Q` matrix will result in progressively larger quantities, based on the magnitude of `D-alpha`. When `M` is then updated based on `Q`, the term `exp(-Q/t)` is likely to evaluate beyond the maximum representable floating-point value, resulting in infinity. The sums from the Sinkhorn Normalization stage are also likely to accumulate to infinity, and the following division would possibly have to deal with $\frac{\infty}{\infty}$, which results in *not-a-number* (`NaN`). At this point, there is no way to remedy the issue and the whole solution is forever spoiled.

The obvious point to address the precision issues is by preventing `exp(-Q/t)` to ever evaluate out of the representable range. In double precision floating-point (64bit) scenario, `exp(-708)` $\approx 3.3 \times 10^{-308}$ is very close to the minimum allowed positive number, that is, 2.2×10^{-308}. Similarly, `exp(+709)` $\approx 8.2 \times 10^{+307}$ is close to the maximum allowed positive number, $1.8 \times 10^{+308}$. Single precision floating-point (32bit) is much worse: `exp(-87)` $\approx 1.6 \times 10^{-38}$ and `exp(+88)` $\approx 1.6 \times 10^{+38}$ already sit near the representable limits, respectively, 1.1×10^{-38} and $3.4 \times 10^{+38}$.

In order to eliminate numerical inconsistencies, the values of `-Q/t` are preset in

a safe range before the evaluation of `exp(-Q/t)`. One could simply bind around the numerical limits highlighted previously, but we found that giving an extra margin to the exponent limits also helps to prevent the posterior division by `sqrt(t)` and subsequent array sums to accumulate to infinity. In this paper we clamped `-Q/t` in the range $[-650, +650]$ when computing in double precision, or $[-70, +70]$ with single precision.

3.6.2 Parallelism in SoftAssign

As can be seen from the pseudo-code, SoftAssign is a heavily sequential algorithm: in order to proceed to the next iteration, all nested iterations should finish, and every operation within each loop is tightly bound to the results of the previous one. Thus, an optimized single-threaded CPU-based implementation of SoftAssign is trivial from the pseudo-code. The challenge is then to harness parallelism from such a conceptually sequential procedure.

Unfortunately *macro-parallelism* in SoftAssign is not feasible due to the sequential nature of the algorithm. What can be done is to exploit *micro-parallelism* from each individual operation within the loops. Most of the operations involved are *one-to-one*: they read-from and write-to individual elements of distinct matrices, and this is simple to parallelize. The exception is for matrix/array sums which are *many-to-one*: multiple elements must be gathered from the input in order to compute a single element of the output.

Even though threads are not intended to optimally deal with micro-parallelism, a multi-threaded CPU-based implementation of SoftAssign can provide significant speedup if designed carefully. The basic idea is to keep each thread responsible for a portion of the matrix address space, synchronizing them before continuing to the next operation. Since threads can keep local internal state indefinitely, implementing sums of arrays is simple if access to the required elements happens without races. It is also possible to further exploit micro-parallelism in CPU if a SIMD instruction subset is available such as Streaming SIMD Extensions (SSE).

On the other hand, a GPU-based implementation of SoftAssign is much more challenging. The following Section discusses in detail the design principles and implementation decisions proposed by this paper to map the SoftAssign algorithm to the GPU.

3.7 SoftAssign on GPU

Even though most of the required operations are one-to-one, thus mapping well to the programmable graphics hardware, the gathering process required by the sums of arrays imposes extra effort. Execution contexts in GPU are much more volatile than threads are in CPU, preventing them to easily hold or share state amongst multiple parallel executions.

There are two philosophies to implement SoftAssign in GPU: a) using traditional GPGPU by wrapping GPGPU concepts around the graphics pipeline, which is more portable, efficient and can inter-operate better with further rendering operations if needed; and b) the more recent GPGPU pipeline exposed through technologies such as CUDA, DirectCompute and OpenCL, which provides better synchronization scheme and read/write

memory access patterns.

In this paper the traditional GPGPU approach was adopted. Besides performance and portability, the choice for a traditional GPGPU implementation was made towards future use of the framework in interactive and progressive rendering optimization research problems. The interested reader can refer to Tamaki *et al.*'s CUDA-based implementation of SoftAssign [TAMAKI et al. (2010)], although applied to a different problem domain.

3.7.1 Mapping SoftAssign on the Graphics Pipeline

All matrices and arrays required by the SoftAssign algorithm are stored as floating-point textures; a texture being a fundamental, highly optimized structure that can access memory mostly in a two-dimensional fashion. These textures should reside in video memory whether possible in order to prevent the pipeline to stall while waiting for data to be transfered, and also to eliminate any bottleneck in the video bus while the algorithm executes.

Read operations on matrices assume that the corresponding textures are bound to texture targets, each on a separated texture unit. Write operations on a matrix assume that the related texture is bound to the framebuffer. In order to execute a given operation, a quadrilateral is issued to be rendered around the interest region. The graphics pipeline will then rasterize such rectangular region, producing fragments. Each fragment holds an automatically interpolated texture coordinate, which can be seen as a matrix index, uniquely addressing a particular element position.

A shader, that is, a small GPU program, is invoked for each fragment. The GPU schedules and executes the same shader, for each generated fragment, in multiple processing units, all in parallel. The shader code uses the texture coordinate to access elements from the currently bound texture units/targets. Once the intended computation is performed on such elements, the shader outputs the result as a color component. Such color will be placed, by the graphics pipeline, into the appropriate position in the framebuffer, which is allegedly pointing to the destination matrix (texture).

One limitation of the current graphics pipeline is that it is not permitted to have a texture bound for reading and writing simultaneously, since this could yield to race conditions and shading languages do not expose intra-synchronization directives. When such conflicts happen, an auxiliary texture can be employed to hold partial results and feedback them subsequently. Fortunately, all of the SoftAssign operations, save for the ones in the Sinkhorn Normalization stage, have distinct read and write access patterns.

3.7.2 SoftAssign Implementation on GPU

Overall, the following shaders must be implemented:

Temperature Stabilization:

- one to compute `Q = M .* (D-alpha)`
- one to compute `M = exp(-Q/t) / sqrt(t)`

Sinkhorn Normalization:

- one for the vertical parallel reduction: `V = sum_columns(M)`
- one for matrix-row division: `A = M ./ V`
- one for the horizontal parallel reduction: `H = sum_rows(A)`
- one for matrix-column division: `M = A ./ H`

The shaders for the temperature stabilization stage are trivial to implement as they only require one-to-one operations. The parameters `alpha` and `t` are defined as uniform variables within the respective shaders. The value of `alpha` has to be set only once, while `t` has to be uploaded for every temperature cooling iteration, which happens in a very low-frequency pace.

As for the shaders of the Sinkhorn normalization stage, additional considerations are required. First, the outlier row is assumed to be attached as an extra row of `M`; this is not a requirement, but reduces the amount of shaders to write and intermediate textures to manage. Second, note that an auxiliary matrix (texture), `A`, is being used in order to eliminate the simultaneous read-write conflict that would otherwise happen in `M`. Third and more important, since both `sum_columns(M)` and `sum_rows(A)` require gathering several elements of `M` and `A`, respectively, the shader must be able to keep track of the partial sums until the total amount is computed.

Although one could simply accumulate all values in a single step, this would significantly compromise the texture cache performance, thus slowing down the entire process. To complicate the matters even more, shading languages do not provide any synchronization directives, and the order of scheduling and execution of the fragments are unpredictable.

The solution is then to employ *Parallel Reduction*, a well known *multi-pass parallel gather pattern*, keeping the state stored in intermediate sets of data that feedback each other at every subsequent pass, as depicted in Figure 3.5. Such parallel reduction algorithms are typically stream-based and cache-coherent, and thus GPU-friendly.

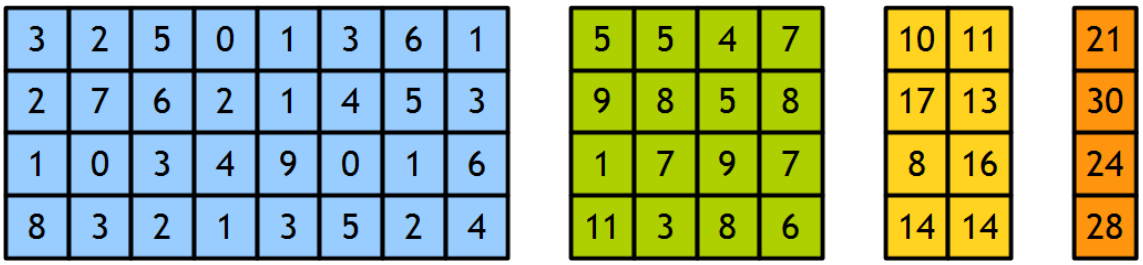


Figure 3.5: Parallel Reduction instance that performs the total sum of each row of a given table. The result is held in a column-vector whose elements correspond to the sum of all elements of that corresponding row in the original input table. At each step, two consecutive elements are gathered and accumulated together from the partial results of the previous step.

In the example of Figure 3.5, only two elements are gathered and accumulated at each step, but operating on more elements can improve performance and lower memory requirements. The ideal number of gathers per pass is hardware-dependent. Small values may sub-utilize the number of available texture fetch units, while big values may stress them, thus penalizing texture-cache performance. Hence the ideal choice is rather empirical and must be refined manually for each particular hardware.

When allocating space for intermediate tables, if the dimensions of the source table happen not to be multiples of the number of gathering samples, they should be rounded-up. Additionally, any attempt to gather elements outside of the boundaries of a table should yield zero value in order to keep the accumulation semantics sound.

In order to compute `sum_all(Q)`, no additional shader effort is required. Such sum can be computed in two stages, first by a vertical reduction, resulting in a single row-vector, and then by an horizontal reduction on such row-vector, resulting in a single value that corresponds to the entire sum of `Q` (the other way around would yield the same result). Moreover, note that the intermediate memory of the parallel reductions used by the Sinkhorn Normalization stage can be shared amongst this reduction as well, as they happen independently.

Another strategy to compute `sum_all(Q)` would be to perform a *2D Parallel Reduction*, a process that resembles texture mip-map generation. This would only increase performance marginally since such a sum is computed at a very low frequency, once per temperature cooling iteration. Besides the effort of implementing and keeping an additional shader, this would also require dedicated additional memory.

Finally, reading-back from GPU is required when comparing the cost of each iteration. Fortunately, such read-back is very small (only one texel). Furthermore, updating the output matrix `S` from the values of `M` does not require a shader, just a simple texel copy. The same can be said for the reassignment of the outlier row before starting each of the Sinkhorn normalization process: a template of the initial outlier row is kept in video memory in a buffer and simply copied over the additional row of `M` when required.

3.7.3 Implementation Details

Our current implementation is OpenGL 1.4 / GLSL 1.0 compliant, requiring few but widely supported OpenGL extensions: framebuffer object, rectangle textures, floating-point textures and shader objects. This makes the algorithm portable to a wide range of GPUs, the lower bound being the commodity GeForce FX series (now 9 generations old).

All textures are stored in 32bit floating-point format (`GL_LUMINANCE32F` or equivalent). Double precision floating-point texture formats are not yet mainstream, and even when available, the hardware may not achieve full performance because not all arithmetic units in commodity graphics GPUs can operate on double precision quantities. This causes the shaders to stall as they race for these units. Half precision floating-point textures, on the other hand, are widely supported and optimized by the graphics hardware and could potentially double the overall performance, but we found that they do not suffice for stable executions of the SoftAssign algorithm.

Textures are bound for read in rectangular texture targets (`GL_TEXTURE_RECTANGLE`

or similar). This is not an enforcement, just a convenience to ease debugging the implementation. Regular 2D texture targets (`GL_TEXTURE_2D`) could be used instead as well.

The OpenGL Shading Language has a built-in function to restrain values within a range called `clamp(val,min,max)`. This is useful to workaround the precision issues when updating `M` as it is hardware accelerated and more efficient than placing a manual conditional logic.

In order to ensure the proper semantics when sampling outside of a texture boundary during the parallel reductions, no special shader control, such as conditionals or extra uniform variables, is required. By simply setting the texture access wrap mode to `GL_CLAMP_TO_BORDER_COLOR`, and specifying `RGBA=(0,0,0,0)` as the border color, is enough to keep the semantics sound.

For parallel reductions, texture filtering can be used to fetch two texels at the same time by sampling at the exact boundary of the corresponding texels, just being careful to multiply the filtered value by two afterwards. Note that some old graphics hardware may not support single precision floating-point texture filtering (GeForce FX Series), even though they may support it for half precision (GeForce 6 Series).

Finally, keep in mind that the hardware has limitations regarding the maximum dimensions for textures. It is possible to split and stitch bigger matrices into smaller textures in order to accommodate all the data, if the limit lies below the required one. Additional control is then required to manage such texture chunks. Also keep in mind that video memory (VRAM) is more scarce than regular RAM. Although most graphics drivers are capable of virtualizing video memory, it is not necessary to do so. Besides, such virtualization is prone to drastically impact the performance.

3.8 Extended Results

The performance comparison between the GPU and CPU implementations is summarized in Table 3.2 and Figure 3.6. Regarding the CPU implementation, for a more fair comparison, we decided to move away from the MATLAB environment and write an optimized multi-threaded implementation in C++ with the Win32 Threads API; the binary was compiled with the Microsoft C Compiler under the Visual C++ 2010 Professional development environment. The GPU implementation is OpenGL 1.4 / GLSL 1.0 compliant.

The hardware configuration used for the performance measurements is an Intel Core2 Quad CPU 2.55GHz with 4GB RAM running Windows 7 Enterprise 32bit, equipped with a GeForce GTX 280 with 1GB VRAM (240 stream processors). All performance results refer to a single temperature cooling iteration, comprised of 10 temperature stabilization iterations, each with 10 Sinkhorn normalization iterations.

Even though the measurements are bound to a specific iteration profile, the performance gracefully scales linearly on the number of iterations. Hence, raising the number of temperature stabilization iterations by a factor of p and the Sinkhorn normalization iterations by a factor of q would yield to a relative increase of $p \times q$ times in the computation time, either in CPU or GPU.

rows	100	100	400	512	900	1024	1200	1600	1800	2048
columns	100	500	1500	2048	3000	4096	5000	6000	7000	8192
CPUx1	0.024	0.114	1.506	2.572	6.485	10.01	14.29	22.73	29.82	39.77
CPUx4	0.022	0.062	0.780	1.479	3.912	5.963	8.532	13.85	17.78	23.62
GPU	0.022	0.023	0.035	0.052	0.114	0.171	0.239	0.374	0.487	0.643

Table 3.2: Performance results for the CPU and GPU implementations of SoftAssign. CPUx1 stands for single-threaded execution, while CPUx4 represents a multi-threaded execution context with 4 threads. All time measurements are expressed in seconds.

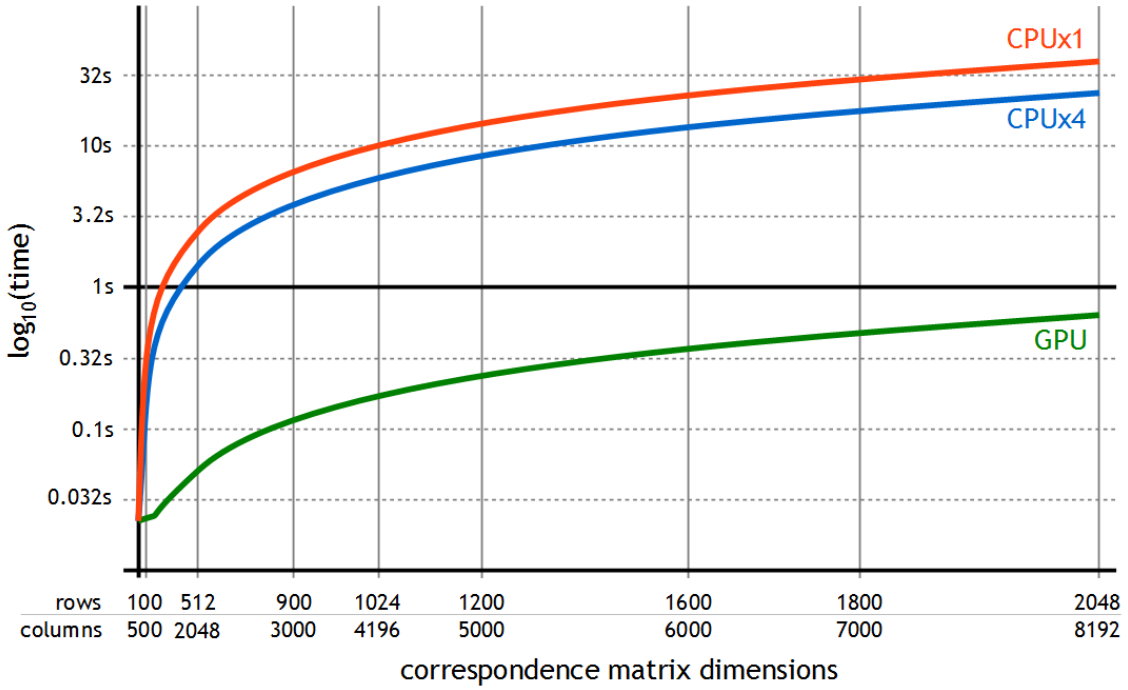


Figure 3.6: Performance chart comparing the CPU and GPU implementations of SoftAssign. For clarity, the chart was plotted with time being expressed in a base-10 logarithmic scale. The samples in the horizontal axis are spaced linearly according to the total number of elements in the distance matrix, that is, $rows \times columns$. Note that the GPU performance is far better than the CPU: the 1s barrier is never reached by the GPU while in the CPU cases this barrier is crossed at very small distance matrix dimensions.

The choice of using four threads is due to the fact that the target CPU, Intel Core2 Quad, is a quad-core microprocessor. Note that the x4 multi-threaded execution is unable to deliver the theoretical 4 times boost in performance when compared to a single-threaded execution. In fact, the x4 multi-threaded performance is very close to a x2 multi-threaded execution. We believe that there are two main reasons for which the implementation is unable to reach peak performance: the first is due the overhead of thread synchronization directives that are essentially managed by the operating system kernel (the thread model was not designed to effective harness micro-parallelism); and the second is the fact that such quad-core processors are arranged in two dies, each holding a dual-core unit that share a common cache.

The GPU implementation, on the other hand, outperforms the 4x multi-threaded CPU implementation by a factor of 35. Compared to a single-thread CPU execution context, the speedup is about 60 times. The only exception is for very small distance matrices: in such cases, the driver overhead congests the actual amount of processing required by the algorithm, and the performance gain is not as significant. However, such small cases are already fast enough to compute anyway and, in practice, neither relevant nor useful at all.

The proposed OpenGL/GLSL-based implementation outperforms Tamaki *et al.*'s CUDA-based SoftAssign [TAMAKI et al. (2010)] by a factor of two in a setup equipped with an identical graphics hardware configuration, a GeForce 8800 GT with 512MB VRAM (112 stream processors). While Tamaki et al. approach takes about 30s to solve a 3000x3000 distance matrix, the proposed approach takes less than 16s.

Finally, a few more examples of photomosaics optimized through the proposed GPU-based implementation of SoftAssign is presented in Figures 3.7 to 3.10. They comprise thematic situations where a given input image is transformed into a photomosaic by using image tiles that have similar semantic context to the one of the original input image. All mosaics were subdivided into 900 patches (30x30) assigned from tile sets composed of 8192 images, thus leading to a 900x8192 distance matrix. The total SoftAssign computing time (for 200 cooling iterations) was less than 1 minute in the GPU execution, while the single-threaded CPU run took about 1 hour and the four-threaded took around 35 minutes.

3.9 Conclusion

Novel strategies to maximize the usage of images for photomosaic synthesis based on a greedy procedural algorithm, simulated annealing and SoftAssign were presented. The maximization is ensured by restricting tiles to be assigned only once to any given patch.

The experimental results show that SoftAssign and SA are effective when the number of available tiles is comparatively small. As more tiles become accessible, SA still remains as the most effective choice, but turns to be impractical due to time constraints; the greedy approach still produces plausible mosaics in such cases.

SoftAssign, on the other hand, not only provides an elegant, deterministic solution for the problem, but also requires much less processing time. The algorithm can be implemented in GPU, providing performance improvements higher than 60 times over optimized CPU implementations.

Such performance improvement is welcome not only to optimize the solution, but also to assist the user in identifying the ideal parameters of the SoftAssign algorithm. There are no principal guidelines on how to tune SoftAssign, thus the search for an optimal parameter set can be very tedious if the user has to wait significant amounts of time between each test setup. A GPU-based implementation of SoftAssign delivers a more immediate feedback to the user.

3.10 Future work

As future work, further investigation is required to determine the impact of more perceptually-aware color similarity metrics to derive the distance matrix and drive the optimization process. The use of different optimization strategies, specially those based on *evolutionary algorithms*, is also a potential and promising target for subsequent analysis.

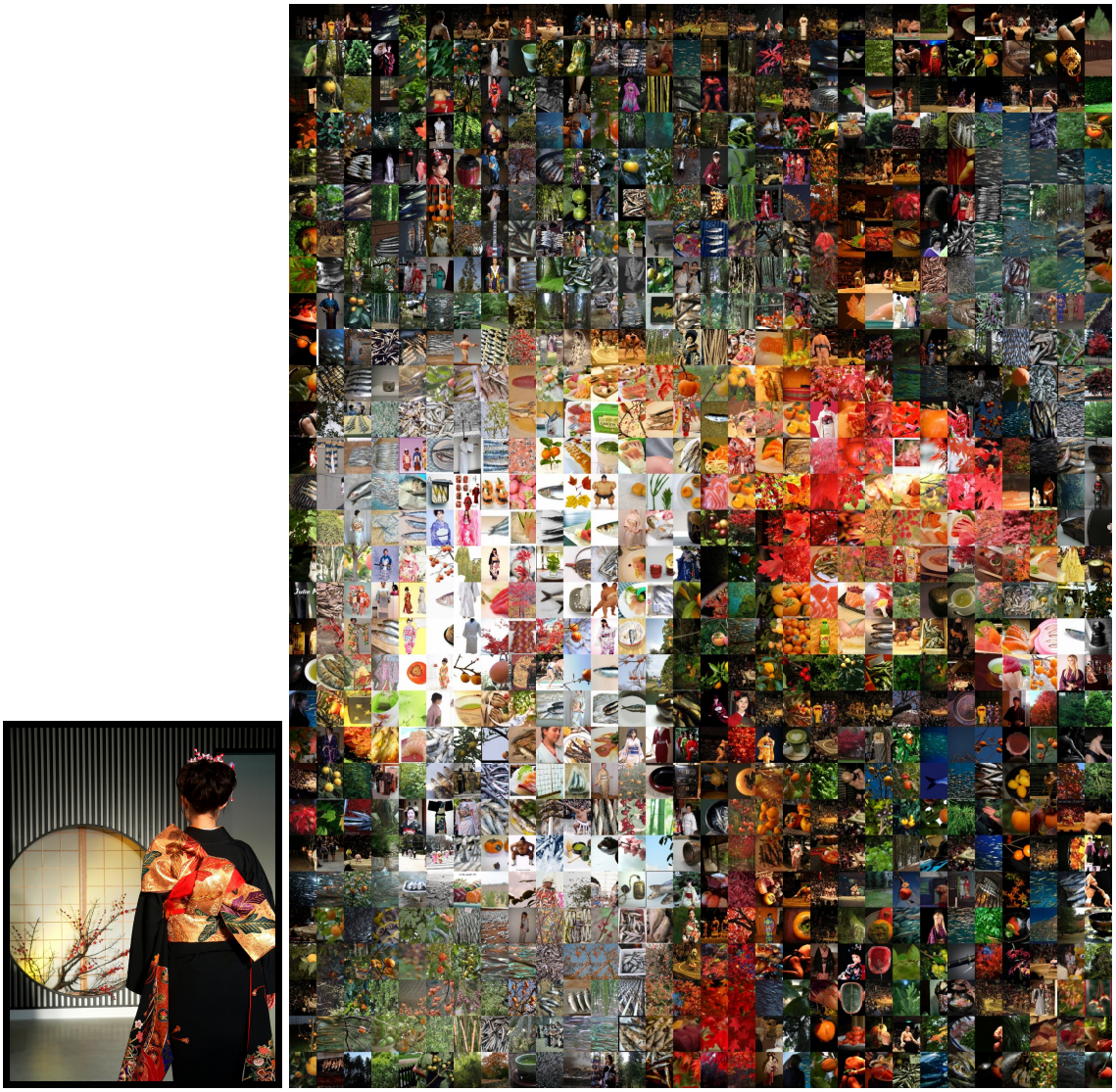


Figure 3.7: A kimono photomosaic made of 30x30 patches selected from a thematic Japanese tile set of 4096 images. The input image is shown miniaturized in the left.
Image and tile set courtesy of `http://www.image-net.org`

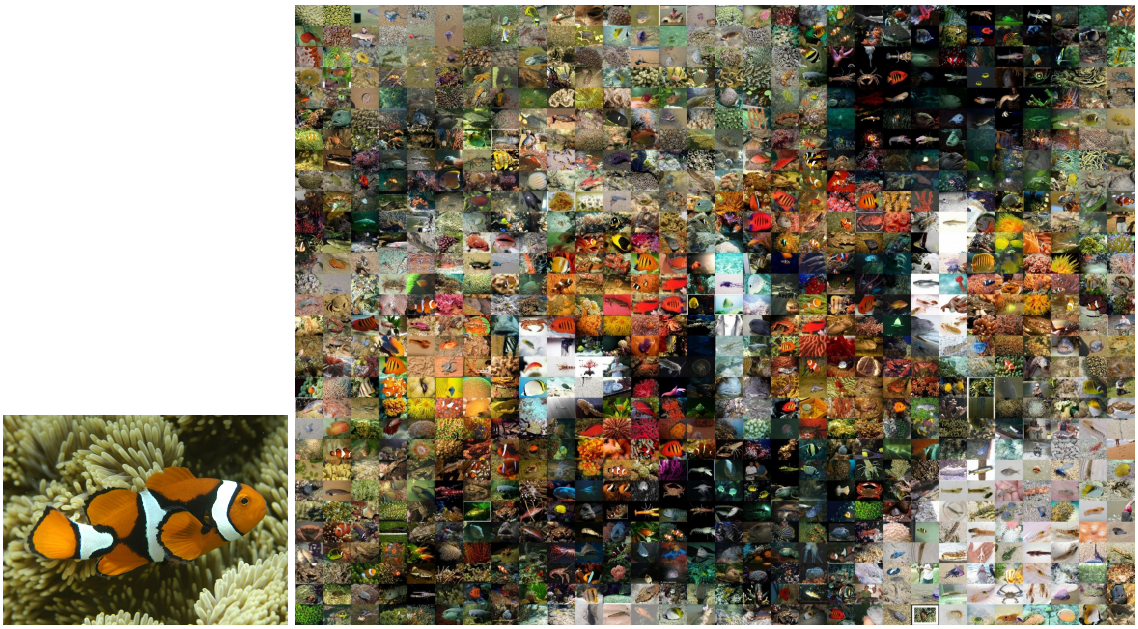


Figure 3.8: A fish image photomosaic made of 30x30 patches selected from a thematic fish tile set of 8192 images. The input image is shown miniaturized in the left.
Image and tile set courtesy of `http://www.image-net.org`

Figure 3.9: A chair image photomosaic made of 30x30 patches selected from a thematic chair tile set of 8192 images. The input image is shown miniaturized in the left.
Image and tile set courtesy of `http://www.image-net.org`

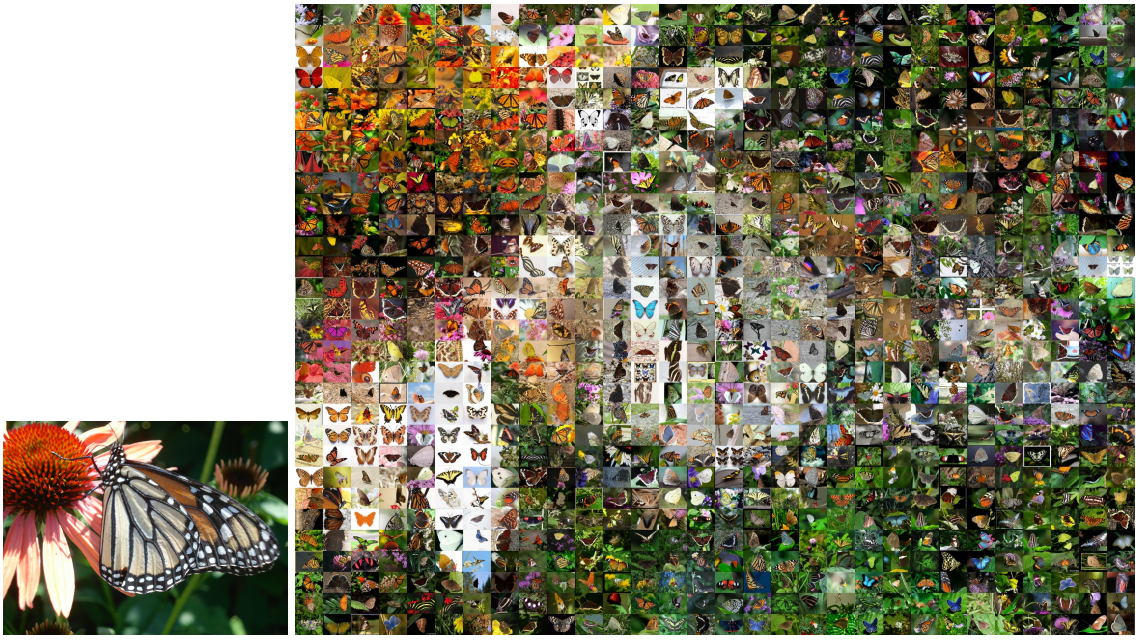


Figure 3.10: A butterfly image photomosaic made of 30x30 patches selected from a thematic butterfly tile set of 4096 images. The input image is shown miniaturized in the left.

Image and tile set courtesy of `http://www.image-net.org`

3.11 Related Publications

SLOMP, M. et al. GPU-based SoftAssign for Maximizing Image Utilization in Photomosaics. **International Journal of Networking and Computing (IJNC)**, Los Alamitos, CA, USA, v.1, p.211–229, July 2011

MIKAMO, M. et al. Maximizing Image Utilization in Photomosaics. In: FIRST INTERNATIONAL CONFERENCE ON NATURAL COMPUTATION (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.275–278

TAMAKI, T. et al. CUDA-based implementations of Softassign and EM-ICP. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR) DEMOS, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010

3.A Appendix A: Photomosaic Optimization Based on Ant Colony Optimization

The "Ants for Japan" Video Photomosaic

[Short Technical Paper for the Evolutionary Art Contest]

Marcos Slomp[1] Bisser Raytchev[2] Toru Tamaki[3] Kazufumi Kaneda[4]
Intelligent Systems and Modeling Laboratory, Department of Information Engineering, Hiroshima University
1-4-1 Kagamiyama, 739-8527, Higashi Hiroshima City, Hiroshima Prefecture, Japan
{[1]marcos-slomp, [2]bisser, [3]tamaki, [4]kin}@hiroshima-u.ac.jp

1. INTRODUCTION

Photomosaics [4] comprise special instances of mosaics. A mosaic is a stylization of an image, consisting of a collection of bulky primitives. A photomosaic is then envisaged as a mosaic whose bulky primitives (patches) are images themselves (tiles). Subjects experience different visual responses based on their relative proximity to a photomosaic. Powerful ideas can be transmitted and intricate artistic effects obtained when photomosaics are used as a form of art [1]. Commercial opportunities can also arise through clever utilization of photomosaics in the hands of skilled marketing personnel [4, 1].

Current photomosaic generation algorithms, however, require huge amounts of images in order to produce attractive results. A common strategy to improve the photomosaic quality is to reduce the size of the bulky primitives by subdividing the input image into very small chunks, or blending the input image on top of the resulting photomosaic; these palliative amends, however, ultimately end up breaking the illusion of the photomosaic. Tuning photomosaics manually is obviously an exhaustive and impractical process. If repetitions are allowed to occur (using the same image at different locations), this may locally bias the photomosaic and introduce undesirable artifacts, rendering parts of the image to look rather displaced and unnatural.

Additionally, some real applications would greatly benefit from using as many of the available tiles as possible (e.g. building a photomosaic of an image of a singer from the images sent by fans, or a photomosaic advertising a company built from images of the products manufactured by the company, etc.). These concerns show that photomosaic generation can be naturally formulated as an optimization process, where the requirements imposed by the nature of the task can be incorporated in the cost function being optimized and the selection of a suitable optimization scheme.

2. THE "ANTS FOR JAPAN" VIDEO PHOTOMOSAIC

Video photomosaics [3], by virtue of possessing an additional dimension (time), can achieve artistic effects which cannot be obtained by the traditional static photomosaics. A whole story could be told, and the dynamics contained in the change of images could be used to an advantage. Also, it would be much more natural and effective to add audio or music to a video photomosaic, than to a static one.

In the present artwork, we actually "evolve" a photomosaic, using an ant-colony-based optimization (ACO) algorithm (more details about ACO are given in the next section). The photomosaics shown as consecutive frames in the video correspond to a subset of solutions of the optimization process obtained by the ants at different iterations of the ACO algorithm. This induces the overall impression that the final photomosaic emerges gradually from an initially chaotic (no predominant structure discernible) collection of images. Our implementation imposes the strong restriction that *no tiles can be reused* in the photomosaic. That is, once a tile-image is assigned to a patch, it can no longer appear in any other patch. Moreover, the patch size used is reasonably large (to allow viewing and appreciation of the individual tile-images), and the available tile-set is quite limited: it consists solely of thematic images related to Japanese culture, customs and geography. The input image was subdivided into 30×30 (900) patches, and the tile-set contained about 8000 images.

The photomosaic generation for this artwork was particularly challenging because of the nature of the input image (see Figure 1), which contains large areas of uniform color (e.g. in the kimono) or uniform patterns (e.g. the wall). The tile-set was not tailored in any particular way to better suit the input image. The resulting photomosaic is "the best" solution found by the ants, based on the tile-set given. Of course, fidelity could be improved with a more carefully selected tile-set, albeit this being a tedious task; the idea is to free the artist of such manual labor and provide the best that can be done with the given resources.

3. METHOD

In order to "evolve" the final photomosaic from an initial "chaotic" state, we have used a modification of the Ant Colony Optimization (ACO) algorithm. Details about the ACO algorithm can be found in [2], here we just briefly summarize the main ideas behind the standard algorithm, and provide some details about the modifications we have made in order to achieve the intended artistic effect.

ACO is a metaheuristic which has been used successfully to solve numerous combinatorial optimization problems like the traveling salesman, vehicle and network routing, quadratic assignment, sequential ordering, etc. ACO is inspired by the behavior of natural ant colonies, where good solutions can emerge from the very simple, or even random, behavior of its individual members, communicating by means of pheromone deposition on the ground, in a cooperative fashion. The ants search for food in a random manner, and when food is found they return to the nest while at the same time deposit pheromone on the ground. This pheromone trail then guides the other ants to the food source.

In the standard ACO algorithm, the problem to be solved is usually modeled using a graph representation, and the ants traverse the graph in search of an optimal solution. In our case, we have a complete bipartite graph, with links (edges) only between pairs of nodes such that one node corresponds to a patch in the source image and the other to a tile-image. The ants traverse the graph, forming a path, and the edges directed from the source patch to the tile-images determine the correspondence which should be used in the photomosaic. Each ant forms its path independently of the others (we have used a colony of 30 ants for our photomosaic). The path of each ant corresponds to a different solution, which is evaluated using a cost function – the cost function we used reflects the visual similarity between the corresponding patches and tiles. Then the ants deposit pheromones on the edges of their path, with the pheromone level being proportional to the quality of the solution. With time, the edges corresponding to good solutions accumulate more pheromones, and this information is shared by the colony (this is their way of communication) when choosing their path during the following iterations: which edge to follow is chosen probabilistically, with edge probabilities determined by the amount of accumulated pheromone and the visual similarity between the images connected by the edge. Additionally, at the end of each iteration a certain amount of pheromone is "evaporated" from all edges, which would discourage the ants from choosing edges which do not lead to promising solution

In order to amplify the effect of gradual evolution and emergence of the photomosaic, at each iteration we have added a gradually decreasing level of "noise" to the solution. This means that a certain number of tile-images (proportional to the noise level) which are selected at the current iteration are randomly replaced by some of the remaining, not yet selected tile-images. This has the additional effect that more of the available source images are being shown (showing as many as possible of the available source images is also one of the objectives of the task).

4. ACKNOWLEDGMENTS

The images used for creating the video photomosaic were obtained from the public image repository at ImageNet[‡]. The music was obtained through a traditional Japanese theatrical performance (*Kabuki*) invitational video from the Japan Foundation[§].

5. REFERENCES

[1] S. Battiato, G. di Blasi, G. M. Farinella, and G. Gallo. Digital mosaic frameworks - an overview. *Computer Graphics Forum*.

[2] M. Dorigo and C. Blum. *Ant Colony Optimization*. MIT Press, July 2004.

[3] A. W. Klein, T. Grant, A. Finkelstein, and M. F. Cohen. Video mosaics. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, NPAR '02, pages 21–ff, New York, NY, USA, 2002. ACM.

[4] R. Silvers. *Photomosaics*. Henry Holt and Co., Inc.

[‡]http://www.image-net.org
[§]http://www.jftor.org

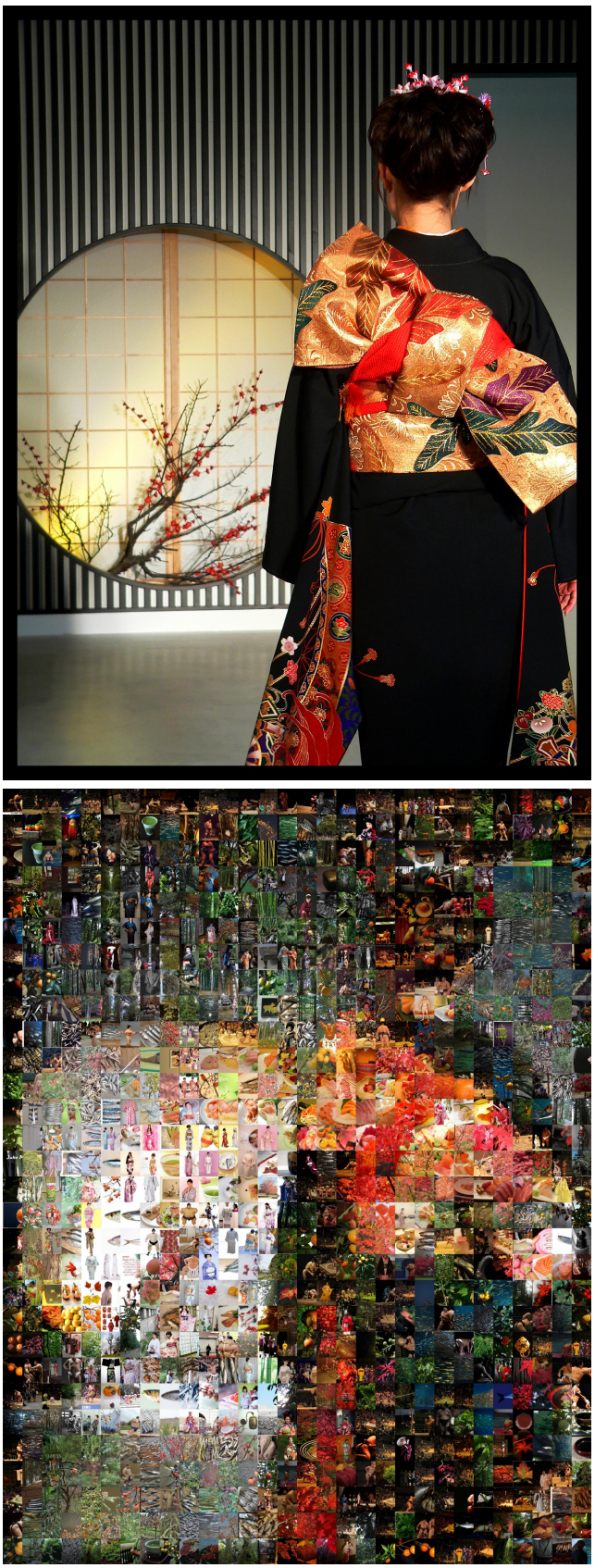


Figure 1: At the top, the reference (input) image; Bottom: synthesized "optimal" photomosaic image.

The "Ants for Japan" Video Photomosaic

[Artistic Statement for the Evolutionary Art Contest]

Marcos Slomp[1] Bisser Raytchev[2] Toru Tamaki[3] Kazufumi Kaneda[4]
Intelligent Systems and Modeling Laboratory, Department of Information Engineering, Hiroshima University
1-4-1 Kagamiyama, 739-8527, Higashi Hiroshima City, Hiroshima Prefecture, Japan
{[1]marcos-slomp, [2]bisser, [3]tamaki, [4]kin}@hiroshima-u.ac.jp

Japan was recently shaken by a catastrophic earthquake followed by a relentless *tsunami*. While much has been said about the material damage and the negative consequences of the tragedy, very little is being said about the gallant efforts managed by the Japanese population towards recovering from the damage. Roads rebuilt in less than a week, schools reformed and reequipped in a few weeks, immediate establishment of decent shelters to protect and reunite affected families, perseverance on "thought-to-be" hopeless rescue attempts...

In some sense, such audacious deeds can be compared to the collaborative behavior of ants: not merely social life-forms that gather food and succumb to the restrictions of the environment, but are indeed remarkable builders that can adapt their work-style to circumvent adverse situations imposed by the habitat, cooperating for the well-being of the entire colony. This artwork comprises a video photomosaic in which the final photomosaic gradually evolves from a "chaotic" state, the entire evolutionary process being simulated with an algorithm inspired by ant-colony behavior. Each photomosaic in the video sequence is made of small images that are related to Japanese culture, customs and geography, and these small images do not repeat within each individual photomosaic.

4 EFFICIENT SUMMED-AREA TABLE AND PREFIX-SUM GENERATION ON THE GPU

4.1 Abstract

Summed-Area Tables offer a powerful data structure for a wide range of tasks, including face recognition, rendering effects and image processing. Summed-Area Tables are the 2D generalization of 1D scans, and two parallel algorithms for scan/SAT generation map well to the GPU: recursive-doubling and binary balanced trees. The balanced tree approach is much faster in general, but was curiously kept aside by the computer graphics community up until recently, despite of its readily availability since long. This Chapter describes both techniques, presenting the challenges for efficient GPU-based implementations and analyzes their performance in different graphics hardwares. As an original contribution, an extension of the binary balanced tree method to higher-order balanced trees is introduced. With this extension, more elements can be reduced and expanded in a same pass, thus substantially improving the performance up to some extent.

4.2 Introduction

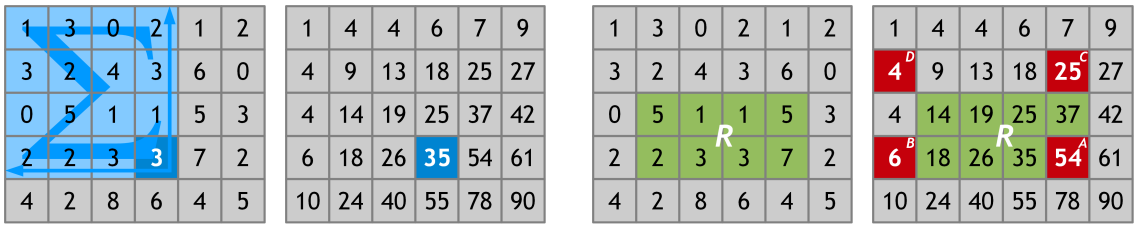
Summed-Area Tables (SAT) were originally introduced as a texture-mapping enhancement over mip-mapping. Although superior in many aspects, precision constraints made SATs inviable for the graphics hardware to follow at that time. Since their conception by Crow [CROW (1984)], SATs were successfully employed on a wide range of tasks ranging from face and object recognition[1] [VIOLA; JONES (2004)], depth-of-field and glossy reflections [HENSLEY et al. (2005)], shadows [LAURITZEN (2007); DÍAZ et al. (2010); SLOMP; TAMAKI; KANEDA (2010)] and tone-mapping [SLOMP; OLIVEIRA (2008)].

A SAT is a *cumulative table*, where each cell corresponds to *the sum of all elements above and to the left of it, inclusive* in the original table, as depicted in Figure 4.1-ab. More formally:

$$SAT(x,y) = \sum_{j=1}^{y} \sum_{i=1}^{x} Table(i,j) \tag{4.1}$$

where $1 \leq x \leq c$ and $1 \leq y \leq r$, with c and r representing the number of columns and rows of the source table, respectively. A SAT therefore has the *same* dimensions of its input table.

The usefulness of SAT comes from the fact that any axis-aligned *rectangular* region of the input table can be *box-filtered*[2] (or integrated) with *only four* lookups on the associated SAT, as depicted in Figure 4.1-cd. This gives the same *constant* filtering complexity $O(1)$ to *any* kernel size.



(a) input table (b) Summed-Area Table (c) filtering region (d) SAT-based filtering

Figure 4.1: A small 6×5 table (a) and its corresponding SAT (b). The highlighted blue cell on the SAT is the sum of all the highlighted blue cells in the input table (all cells up and to the left, inclusive). In order to filter the 8 elements marked in green in the input table (c), only the four red elements A, B, C and D need to be fetched from the SAT (d), a fact that holds true for arbitrary sizes. The filtering result is given by: $\frac{A-B-C+D}{area} = \frac{54-6-25+4}{4*2} = \frac{27}{8} = 3.375$.

From Figure 4.1 it is easy to realize that, if the coordinates of a sampling region R in the original table are given as (x_{min}, y_{min}) and (x_{max}, y_{max}) then the corresponding SAT cells A, B, C and D to be fetched are obtained as follows:

[1] Summed-Area Tables are also referred to as *integral images* in some image processing, computer vision and pattern recognition contexts.

[2] Higher-Order Summed-Area Tables [HECKBERT (1986b)] can extend plain SAT beyond box-filtering, allowing for triangular and spline-based filtering, at the expense of increased constant time overhead and numerical precision issues. Although compelling, a more in-depth discussion on the subject is out of the scope of this thesis since plain SAT are sufficient for the techniques used throughout this document.

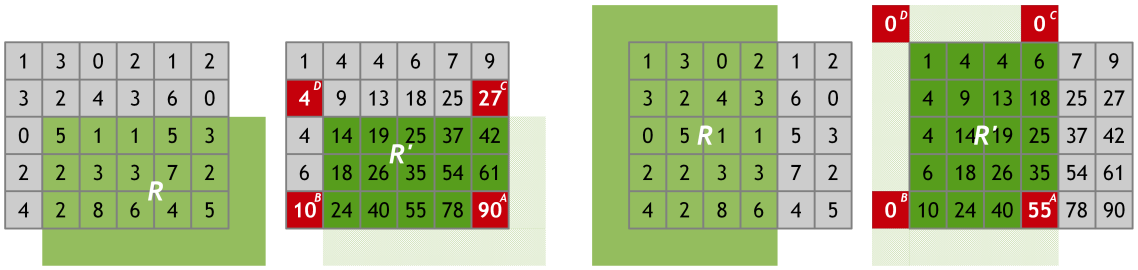
$$\begin{aligned} A(R) &= SAT(x_{max}, y_{max}) \\ B(R) &= SAT(x_{min} - 1, y_{max}) \\ C(R) &= SAT(x_{max}, y_{min} - 1) \\ D(R) &= SAT(x_{min} - 1, y_{min} - 1) \end{aligned} \tag{4.2}$$

Once these particular cells are determined, filtering a region R can be simply performed according to the following expression:

$$Filter(R) = \frac{Sum(R)}{Area(R)} = \frac{A(R) - B(R) - C(R) + D(R)}{(x_{max} - x_{min} + 1)(y_{max} - y_{min} + 1)} \tag{4.3}$$

Fetching cells outside the boundaries of the SAT, however, requires special attention: elements out of the upper or left boundary are assumed to evaluate to *zero* (analogous to the *clamp-to-border-color* mode in OpenGL, with the border color set to zero), while elements out of the bottom or right boundary should be redirected back to the closest element at the respective boundary (analogous to the *clamp-to-edge* mode in OpenGL). These conditions are illustrated in Figure 4.2 and summarized in the formula below:

$$SAT(x, y) = \begin{cases} 0 & \text{if } x < 1 \text{ or } y < 1 \\ SAT(c, y) & \text{if } x > c \\ SAT(x, r) & \text{if } y > r \\ SAT(c, r) & \text{if } x > c \text{ and } y > r \\ SAT(x, y) & \text{otherwise} \end{cases} \tag{4.4}$$



(a) filtering region (b) SAT-based filtering (c) filtering region (d) SAT-based filtering

Figure 4.2: Example of SAT-based filtering for regions that extend outside the boundaries of the table. The areas hatched in light green in (b) and (d) should be discarded from the original areas R in (a) and (c), respectively, in order to evaluate the correct filtering result; the correct area R' is shown in dark green in (b) and (d).

Note that by adjusting the original area of R to the boundary restrictions of the SAT, the effective area of R is prone to changes and should be taken into account prior to evaluating Equation 4.3. This is also shown in Figure 4.2.

Additionally, standard bilinear filtering can also be used to sample the SAT at non-integer cell locations if necessary (the boundary restrictions above must still be respected).

Note that the original table could be entirely discarded: the SAT alone is capable of restoring the original values of the table from which it was built from. However, depending on the application, if the values of the input table are to be used constantly along with

the SAT, it is a good idea to keep the input table at hand. This is specially true if the underlying numerical storage representation is prone to introduce precision errors due to arithmetic operations (i.e., floating-point; more on precision issues in Section 4.6).

All in all, Summed-Area Tables offer an elegant solution to filter rectangular regions of tables/textures in constant time $O(1)$. The challenge is now to find a fast way to compute them. A brute-force algorithm straight from Equation 4.1 is of order $O(n^2)$. A more efficient approach relies on a *purely sequential* algorithm that runs in $O(n)$, as originally suggested by Crow [CROW (1984)], but from which parallelism is impossible to be exploited. The next Sections will discuss strategies for parallel implementations of SAT generation. In the context of this thesis, it is desirable to focus on such parallel algorithms that map well to the GPU and do not rely on any download or upload between main memory and video memory.

4.3 Fast Summed-Area Table Generation on the GPU

Summed-Area Tables can be seen as the 2D equivalent of 1D array prefix-sums. A prefix-sum is a cumulative array, where each element is the sum of all elements to the left of it, inclusive, in the original array. There are two types[3] of prefix-sum: *prescan* and *scan*. A prescan differs from a scan by a leading zero in the array and a missing final accumulation; refer to Figure 4.3 for an example. As a matter of fact, prefix-sums are not limited to the addition operation, but can be generalized to any other binary operation (neutral element is required for prescans). In the context of SAT generation, however, prescans are not particularly useful and just scans *under the addition operation* suffice.

(a) input array (b) prescan (c) scan

Figure 4.3: An example of a prescan (b) and a scan (c), under the addition operation, on some input array (a).

The process of generating a SAT can be broken down into a two stage array scan. First, each row of the input table is independently submitted to a 1D array scan. The resulting table, to be referred to here as a *partial SAT*, is then submitted to another set of 1D scans, this time operating on each of its columns. The resulting table this time is the complete SAT itself. The process is illustrated in Figure 4.4. A more formal derivation can be achieved by isolating the sums of Equation 4.1, as demonstrated below:

$$SAT(x,y) = \sum_{j=1}^{y} \left[\sum_{i=1}^{x} Table(i,j) \right] \tag{4.5}$$

now, since the sum inside the brackets is bound to a specific row j, the bracketed sum consists of a scan operation on the j^{th} row alone (an 1D array); this horizontal 1D scan

[3] A prescan may also be referred to as an *exclusive* prefix-sum, while a scan can be referred to as either an *inclusive* prefix-sum or as an *all*-prefix-sum [BLELLOCH (1990)].

will be referred to here as $\sigma(x, j)$:

$$\sigma(x, j) = \sum_{i=1}^{x} Table(i, j) \tag{4.6}$$

by appropriately pugging $\sigma(x, j)$ back into Equation 4.5 one obtains:

$$SAT(x, y) = \sum_{j=1}^{y} \sigma(x, j) \tag{4.7}$$

but this time the sum is bound to a fixed column x; the result of this sum is evaluated by accumulating all of the values of $\sigma(x, j)$ for $1 \leq j \leq y$, that is, a (vertical) 1D scan operation on the x^{th} column of the set of horizontal row scans of $\sigma(x, j)$.

(a) input table (b) partial SAT (c) partial SAT (d) complete SAT

Figure 4.4: SAT generation as a set of 1D array scans. Each row of the input table (a) is submitted to an 1D array scan, leading to a partial SAT (b). Each column of this partial SAT (c) is then submitted to another 1D array scan, resulting in the complete SAT (d).

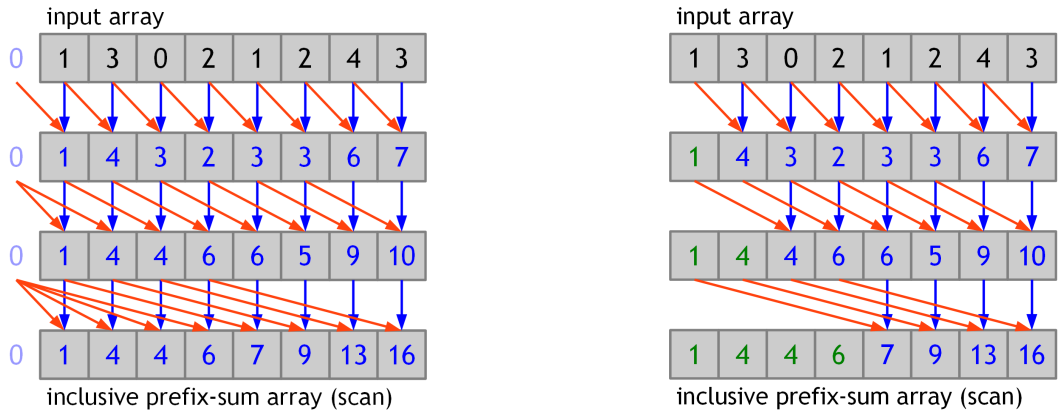
Prefix-sum generation is a straight-forward $O(n)$ procedure using a purely sequential algorithm. Prefix-sums, however, comprise a versatile and fundamental building block for many parallel algorithms. Therefore, efficient methods that harness parallelism from prefix-sum generation also exist. Two of these algorithms are based on multi-pass parallel gathering patterns that map particularly well to the GPU: recursive-doubling [DUBOIS; RODRIGUE (1977)] and balanced-trees [BLELLOCH (1990)].

The approach based on balanced-trees perform less arithmetic operations than the recursive doubling one, but requires twice as much passes. This trade-off, however, quickly starts to pay-off for moderately larger inputs, with balanced-trees being much more work-efficient and faster than recursive-doubling on a GPU-based implementation. As far as the parallel complexity goes, the balanced-tree approach is $O(n/p + log(p))$ while recursive-doubling is $O(n/p\ log(n))$. A complexity analysis on both algorithms is available in the Appendix (Section 4.A), but only considering sequential executions of the algorithms for simplicity. The interested reader is redirected to DUBOIS; RODRIGUE (1977)], [BLELLOCH (1990)] and [HARRIS (2007)], for a more in-depth discussion on how these complexities scale to parallel scenarios.

Until recently the computer graphics community has curiously favored the recursive-doubling approach, despite the attractive performance gains and readily availability since long of the method based on balanced-trees. The following sections introduces both approaches. Only 1D array scans will be detailed, but their extension to SAT generation should be clear from Figure 4.4: all rows/columns can be processed simultaneously at the same pass with exactly the same shader. Algorithmic changes between an horizontal and a vertical scan are trivial and will not be discussed.

4.4 Parallel Scan Generation on the GPU with Recursive Doubling

Relying on the recursive doubling pattern, a parallel gather operation amongst an 1D array with n elements is performed in $log_2(n)$ steps [DUBOIS; RODRIGUE (1977)]. For a scan, each step consists on updating a number of elements by accumulating two values from the array. These updated elements will then be reused in the subsequent step, and so on. An walk-through on the algorithm is shown in Figure 4.5.



(a) update all elements

(b) update necessary only

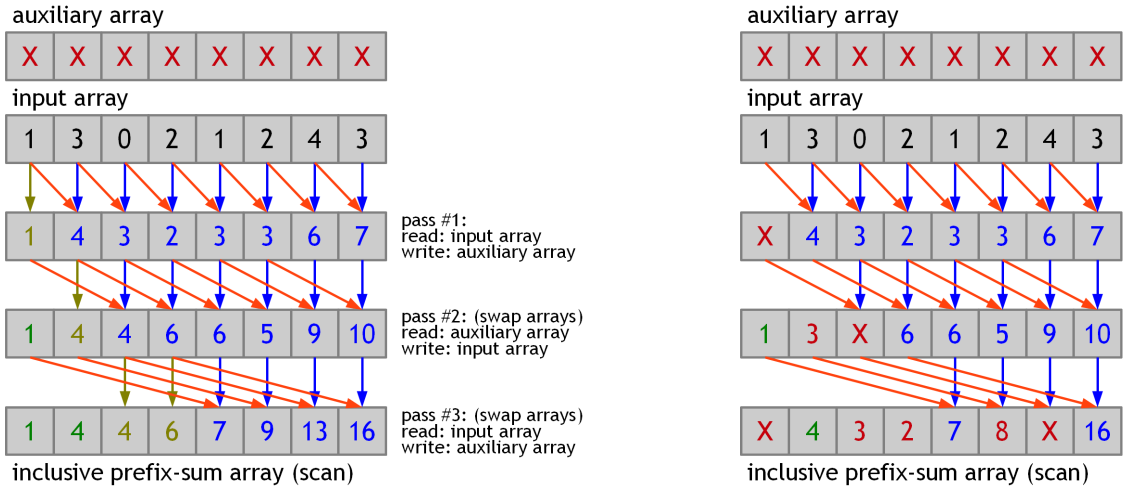
Figure 4.5: An walk-through on scan generation using the recursive-doubling approach. In each step, every element updates itself by accumulating two values: itself (blue arrows) and a neighbor to the left (orange arrows). At each step the offset to the neighbor doubles. In (a) all elements are updated at each step. It can be observed in (a) that the first few elements of each step do not need to be updated again since they would accumulate with a *ghost* neighbor (zero-valued elements in violet) which would not modify their values. The image in (b) exploits this fact and exclude such elements from the update (heading elements marked in green), only updating those who need to be updated (marked in blue).

From Figure 4.5 it is possible to see that everything happens *in-place* within the same input array, that is, no additional memory is required. It is also clear that at each pass $1 \leq i \leq log_2(n)$, the neighbor offset is given by 2^{i-1} and that the first 2^{i-1} elements do not need to be re-updated since they would accumulate with a neighbor element that does not exist in the array (which are assumed to have zero-value).

Such memory-efficient algorithm, however, does not map well to the GPU due to an inherent limitation of current GPU architectures: the impossibility of simultaneously reading and writing from/to the same texture memory and the lack of proper synchronization directives at the global-level. In order to implement the algorithm on the GPU, an additional array of size n has to be used to accumulate the intermediate values; this way, one array is used as a *read-only* resource while the other one is used as a *write-only* target. Once a pass finishes, the arrays are swapped and the process repeats; this pattern is informally known as *ping-pong rendering* by GPGPU practitioners.

A walk-through on a GPU-based implementation of recursive-doubling is depicted in Figure 4.6-a. Note that differently from the *in-place* run of Figure 4.5, each pass i now requires a number of $max(1,\ 2^{i-2})$ elements to be copied onto the write-only array in order to keep the write-only and read-only arrays in perfect synchrony for the subsequent

passes: only the first $max(0,\ 2^{i-2})$ elements are actually completely synchronized and do not need to be copied or updated. Without these copies, after a swap the values of the now read-only array would contain either non-initialized values from the auxiliary array or out-dated values from the input array, as illustrated in Figure 4.6-b.



(a) correct semantics (b) inconsistent execution

Figure 4.6: An walk-through on a GPU-based scan generation using the recursive-doubling approach. At each pass the input and the auxiliary arrays are swapped. An important procedure in this *ping-pong* approach is to keep the values of both arrays consistent for subsequent passes by copying a number of elements from the current read-only array to the write-only array, as illustrated in (a) where these copies are marked in dark-yellow. Without these copies, subsequent passes would operate with (and output) inconsistent values, as shown in (b), with red values indicating the inconsistencies.

Even though the descriptions and examples provided so far dealt with the accumulation of only two elements per pass, the algorithm can be modified to handle any integer number of $k \mid 2 \leq k \leq n$ accumulations per pass. By doing so, the number of passes required reduces to $log_k(n)$. At each pass the first $max(0,\ k^{i-2})$ are already synchronized in both arrays and do not need any special processing, while the following $max(1,\ k^{i-2})$ just have to be copied over; the remaining $n - k^{i-1}$ elements are updated by accumulating themselves (in the read-only array) with $k - 1$ neighbors to the left, each neighbor being spaced by k^{i-1} elements. If an expected neighbor happens to lie outside of the left boundary of the array (a *ghost* neighbor), it is assumed to have zero-value. The technique also works seamlessly when n is not a power or multiple of k.

Despite the fact that values of $k > 2$ can reduce the number of passes significantly, as well as the number of updated elements per pass, the optimal performance is unlikely to be achieved with a large number for k. The reason comes from the fact that more elements have to be retrieved (and accumulated) for each update and, more importantly, that the spacing between neighbors increase exponentially. This memory access pattern is prone of thrashing the cache memory, eventually hurting the cache performance and accuracy due to memory access latency. Such characteristics are particular to each GPU architecture and organization and the optimal value for k has to be determined empirically. From the results of Section 4.7 it was found that $k = 4$ provides a good initial estimation on the inspected hardware profiles.

Note that the GPU-based implementation described in this section consumes the input array in the process. The input vector can be reconstructed from the scan itself, but if the input vector is to be kept intact for other tasks (for performance or precision-related reasons), it should first be copied into another auxiliary buffer, thus increasing the additional memory requirements from n to $2n$.

4.5 Fast Parallel Scan Generation on the GPU Based on Balanced-Trees

Blelloch replaced the recursive-doubling pattern by a more work-efficient one: a binary balanced tree [BLELLOCH (1990)]. There are two stages involved in this binary balanced tree technique: a *reduction* stage (up-sweep) and an *expansion* stage (down-sweep). The input array is submitted to a reduction stage structured like a binary balanced tree, starting from the leaves (the input array itself) up to the root. The nodes of this tree are then used to generate yet another tree, this time spanning from the root down to the leaves, with the resulting leaves corresponding to a prescan of the input array.

A balanced binary trees with n leaves has $log_2(n) + 1$ levels and, therefore, $2n - 1$ nodes in total. Since two trees are generated in the process, a total of $4n - 2$ nodes is produced. However, since the input array itself is already available only $3n - 2$ nodes are actually generated and processed. Hence the $O(n)$ complexity.

Prefix-sum surveys on the literature that mention the balanced-tree approach usually describe it as a method for producing prescans only [BLELLOCH (1990); HARRIS (2007)]. Although prescans can be easily converted into scans in a few different ways (given that the input is still available), this additional computation is not necessary since it is actually possible to modify the plain balanced-tree approach slightly in order to produce a scan directly. This section will focus only on this direct scan generation since it is more useful for SAT generation.

The reduction stage is straight-forward and consists on successively accumulating two nodes in $log_2(n)$ passes. Starting from the input array, each pass produces a set of partial sums on the input array; the last pass produces a single node (the root) comprising the sum of all elements of the input array. This resembles an 1D mip-map reduction, except that averages are not taken. The left side of Figure 4.7 depicts the process.

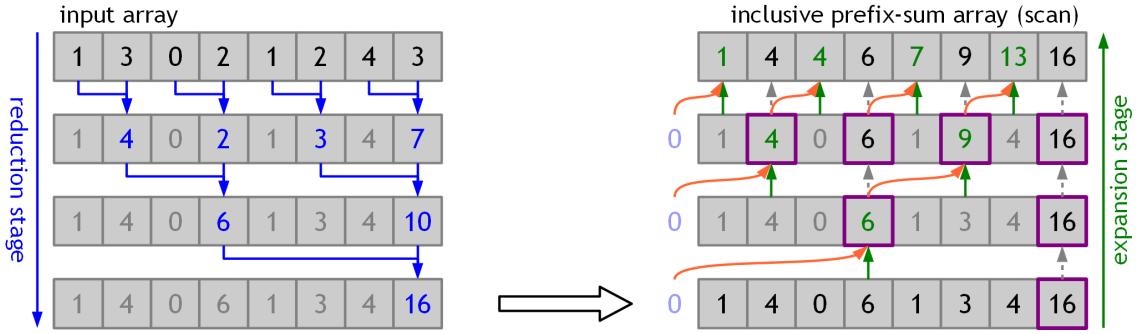


Figure 4.7: An walk-through on scan generation using the balanced-tree approach. The reduction stage is shown in the left and the expansion stage in the right.

The expansion stage is less intuitive and challenging to put into words. The reader is directed to Figure 4.7-right for the explanation to follow. Expansion starts from the root node of the reduction stage, referred to here as the first *generator* node (outlined in magenta). The generator node itself is its own rightmost child (dashed gray arrows). The left child is computed by adding its own value from the reduction tree (green arrows) with the value of its *uncle* generator node immediately to the left (orange arrows). If there is no such *uncle* node, a *ghost* zero-valued uncle is assumed (violet zero-valued nodes). Both children now become generator (parent) nodes (again outlined in magenta) for the next pass and the process repeats. After $log_2(n)$ passes the resulting array will be the scan of the input array.

The process illustrated in Figure 4.7 happens *in-place* and is therefore memory-efficient since all computations are performed successively on the same input array without the need of any auxiliary memory. Unfortunately, a GPU-based implementation, just like with recursive-doubling, would suffer from the impossibility of performing simultaneous read and write operations on the same texture memory, not to mention the lack of global-level synchronization directives. For a GPU-based implementation the intermediate reduction and expansion stages will also require additional memory to store their computations. A depiction of the suggested layout for this extra memory is presented in Figure 4.8.

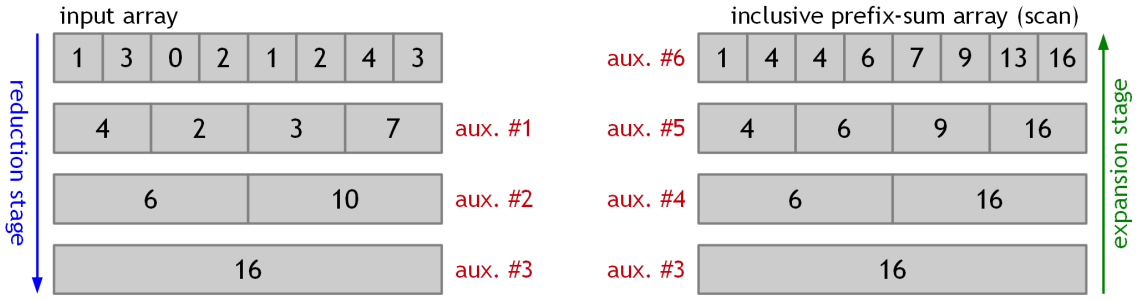


Figure 4.8: Suggested layout for the auxiliary GPU memory during the scan. Simultaneous read/write from/to a same buffer never happens. In order to expand aux.#4 the expanded parent buffer aux.#3 and the reduced *sibling* buffer aux.#2 must be accessed. Similarly, expanding aux.#5 needs access to aux.#4 and aux.#1. The final expansion uses aux.#5 and the input buffer itself. The extra memory amounts to about three times the size of the input.

Even though the amount of necessary auxiliary memory with this layout is substantially large ($\approx 3n$), the memory access patterns becomes more cache-coherent than the ones from the memory-efficient version of Figure 4.7 since the data in each buffer is laid out together in two structures instead of sparsely distributed in a single array. Moreover, this cache-coherence is also another attractive advantage that a GPU-based scan with balanced-trees has over a recursive-doubling one. Also note that the input array is never written over by the algorithm, thus there is no need to accommodate an extra copy of the input in case the application wishes to have it available for other reasons.

Similarly to recursive-doubling, there is no need to limit the computations to two nodes per pass. Reducing and expanding a fixed number of $k \mid 2 \leq k \leq n$ nodes per pass requires just a few modifications and can substantially improve the performance, as well as lower the amount of extra intermediate memory required. The general rules for

an optimal value of k follow the same logic as with recursive-doubling. In the hardware profiles investigated it was found that $k = 4$ provided the overall best performance (see Section 4.7).

A reduction phase with $k > 2$ is straight-forward to implement, but the expansion phase is again more involved. Each generator node will now span k children per pass. To compute the expanded value of any child, the respective child value from the reduction tree is added together with its expanded uncle node, just as when $k = 2$. However, the values of *all* their *reduced siblings* immediately to the *left* have to be added together as well. For example, if $k = 8$ then the expanded value of the 5^{th} child will be the sum of its expanded uncle node with its own respective node in the reduction tree, plus the sum of all of its siblings to the left in the reduction tree, namely, the 1^{st}, 2^{nd}, 3^{rd} and the 4^{th}.

This process works seamlessly even when the array length n is not a power or multiple of k; the only requirement being that any access outside of the left boundary of any of the arrays should evaluate to zero-value. Note that the value of the generator node is no longer propagated. One could propagate it to its k^{th} child (if any), but this is less systematic since special cases need to be accounted in the shader. As a matter of fact, since the root node of the reduction stage is never used in the expansion, this final reduction stage does not need to be computed at all; recall that only the left sibling of the root node is used during the first expansion pass, and this sibling node is nothing but a zero-valued ghost node to be used as the first uncle node of the expansion.

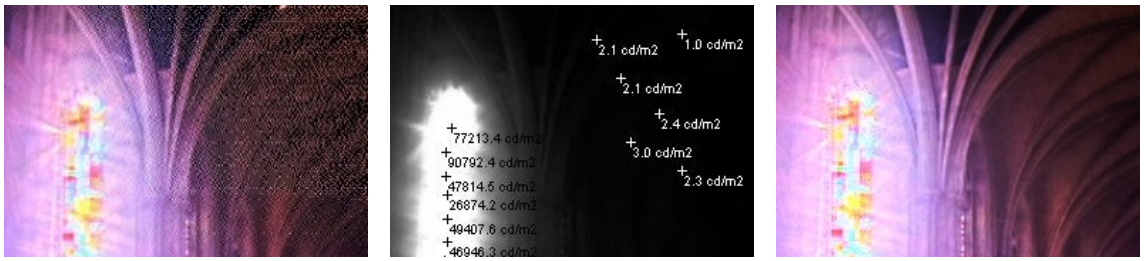
Equipped with such an algorithm, a GeForce GTX 280 is capable of generating a 2048x2048 SAT in about $2ms$. In comparison, recursive-doubling would take nearly $6ms$. In general, the overall speedup is of roughly 3x. Performance results are summarized in Section 4.7.

4.6 A Note on Precision Issues with Summed-Area Tables

Summed-Area Table generation is vulnerable to several precision issues. Luminances, for instance, are positive quantities, which makes the corresponding SATs built from luminance images to grow monotonically. The values in the SAT can quickly reach overflow limits or run out of fractional precision due to ever increasing accumulations. Depending on the magnitude and fractional distribution of the input values involved, as well as the dimensions of the table, these critical situations may be hastily reached.

When such situations happen, high-frequency noise artifacts (*salt-and-pepper*) may appear in the resulting image that was generated through the application of the SAT, thus compromising the image quality. An example is shown in Figure 4.9 in the context of tone mapping (more on this SAT-based tone mapping technique in Chapter A). One way to mitigate this problem is to subtract the average value of the input table from the table itself prior to SAT generation [HENSLEY et al. (2005)]. This simple procedure has two main implications: first, the SAT allots an additional bit of precision, the signal bit, due to introduction of negative quantities, and second, the SAT is no longer monotonic and thus the range of the values within the SAT should reach much lower magnitudes.

After filtering with this non-monotonic SAT, keep in mind that the average input table value should be added back. The average value can be computed using a mip-mapping



(a) plain monotonic SAT (b) luminance magnitude range (c) non-monotonic SAT

Figure 4.9: Summed-Area Tables of luminance images are inherently monotonic and prone to unpleasing noise artifacts (a) if the quantities involved have a wide dynamic range (b). Making the SAT non-monotonic by first subtracting the average luminance mitigates such artifacts.

reduction technique (more on this in Chapter A). The performance overhead incurred is small and well worth for the extra robustness. There is no need for additional memory to store this average-subtracted table: the average can be subtracted onto the original table by using subtractive color blending (which can be achieved in OpenGL through `glBlendEquation(GL_FUNC_REVERSE_SUBTRACT)`). The only extra memory required is the one used to build and store the mip-map levels which would only amount to about $1/3$ of the size occupied by the table.

4.7 Results

This Section analyzes the performance achievements of balanced-trees over recursive-doubling for GPU-based SAT generation. The system configurations and hardware profiles investigated are listed below (shader cores and frequencies retrieved *on-the-fly* during the demo run-time with GPU Caps Viewer v1.14.2):

1. Windows 7 Enterprise 32bit SP1 running on an Intel(R) Core(TM)2 Quad CPU Q9499 2.66GHz with 4GB RAM equipped with a NVIDIA GeForce GTX 280 with 1GB VRAM (240 shader cores at 1107MHz, memory at 1296MHz, WHQL Driver 280.26)

2. Windows 7 Enterprise 32bit SP1 running on an Intel(R) Core(TM)2 Quad CPU Q8200 2.33GHz with 4GB RAM equipped with a NVIDIA GeForce 9800 GT with 512MB VRAM (112 shader cores at 1500MHz, memory at 900MHz, WHQL Driver 280.26)

3. Windows XP Professional x64 Edition SP2 running on an Intel(R) Xeon(R) CPU W3520 2.67GHz with 8GB RAM equipped with an ATI FirePro 3D V3700 with 256MB VRAM (40 shader cores at 800MHz, memory at 950MHz, WHQL Catalyst Driver v8.85.7.1)

The main program was implemented in C++, compiled and linked with Visual C++ Professional 2010. The graphics API of choice was OpenGL and all shaders were implemented in conformance to the feature-set of the OpenGL Shading Language (GLSL) version 1.20.

All performance times in this Section are given in *milliseconds*. Full-frame times were captured with performance counters from the Win32 API and double-checked with the free version of Fraps 3.4.6. Intra-frame performance was profiled using OpenGL Timer Query Objects (`GL_ARB_timer_query`). Performance results were recorded through multiple executions of the program from which outliers were removed and the average was taken.

Summed-Area Table generation times for typical image resolutions using recursive-doubling and balanced-trees are presented in Figures 4.10 and 4.11, respectively. All plotted times are compiled in Table 4.1. The speed-up achieved with the balanced-tree approach is shown in Figure 4.12, from which it can be seen that SAT generation with the balanced-tree outperforms recursive-doubling by a factor of 2.5x≈3x (or 4x on the ATI FirePro 3D V3700) as the image size increases.

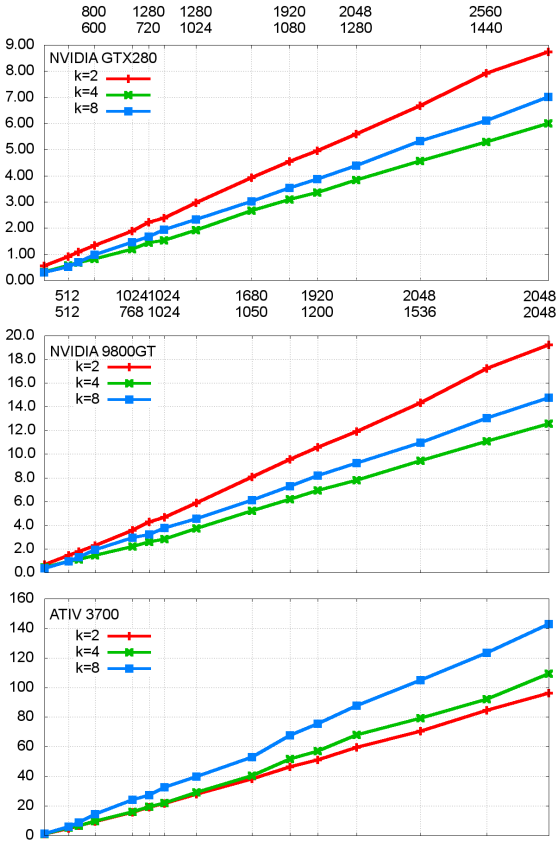


Figure 4.10: Summed-Area Table generation time using recursive-doubling.

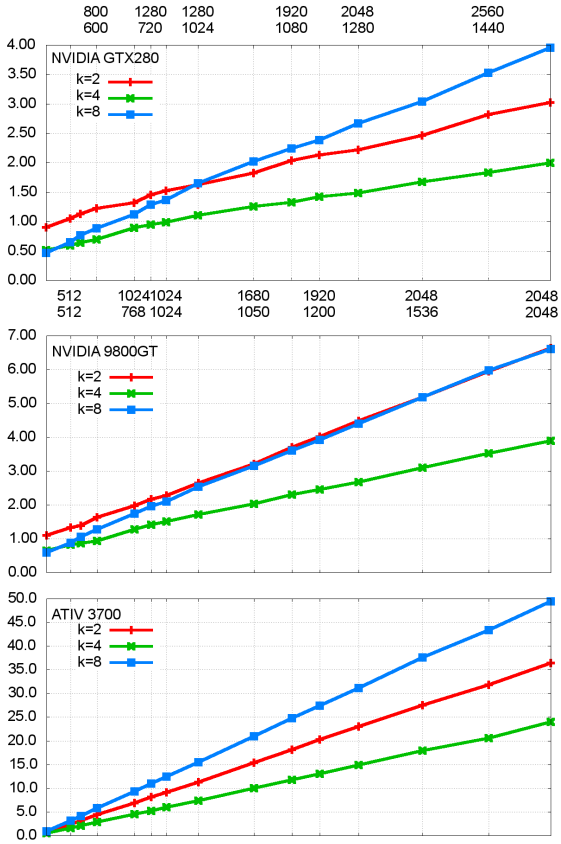


Figure 4.11: Summed-Area Table generation time using balanced-trees.

It must be noted that for small tables the recursive-doubling approach may achieve better performance. This is due to the relative time spent between pass switching and the amount of shader processing actually performed. Every time a new pass is setup there is some implicit GPU driver overhead to prepare the state of the new pass. Recall that the balanced-tree approach requires *twice* as much passes than recursive-doubling and therefore suffers more penalties during such pass switches. For small tables, the time spent on setting-up the passes tend to be *higher* than the actual processing time, and the technique that performs less passes will perform better in general.

The performance impact of switching passes becomes less critical than the actual processing time as the table size increases. Nonetheless, small tables are not as much interesting and either algorithm would still run appropriately fast on them.

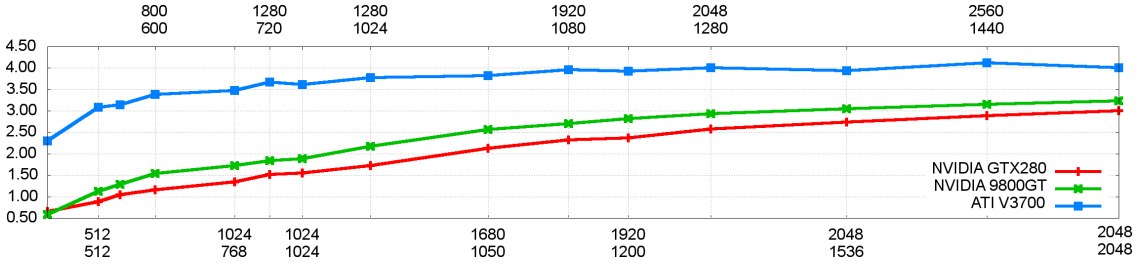


Figure 4.12: Relative speed-up between balanced-trees and recursive-doubling for SAT generation based on the best (fastest k) times recorded for each algorithm for each image size, according to the performance results of Figures 4.10 and 4.11.

Another interesting fact that can be observed is how the performance of the techniques scale on different GPUs based on the capabilities of the hardware. A simple combined ratio between the number of shader cores, shader clock frequency and video memory frequency of two different GPUs can provide an indicator of the theoretical speedup expected from one GPU in comparison to another:

$$\frac{GPU_1}{GPU_2} = \frac{\#cores_1}{\#cores_2} \cdot \frac{clock_1}{clock_2} \cdot \frac{VRAMf_1}{VRAMf_2} \tag{4.8}$$

from which one can realize that the expected speedups between the inspected GPUs are:

$$\begin{aligned} \frac{NVGTX280}{NV9800GT} &= \frac{240}{112} \cdot \frac{1107}{1500} \cdot \frac{1296}{900} \approx 2.3\text{x} \\ \frac{NV9800GT}{ATIV3700} &= \frac{112}{40} \cdot \frac{1500}{800} \cdot \frac{900}{950} \approx 5.0\text{x} \\ \frac{NVGTX280}{ATIV3700} &= \frac{240}{40} \cdot \frac{1107}{800} \cdot \frac{1296}{950} \approx 11.3\text{x} \end{aligned} \tag{4.9}$$

and these theoretical speedups roughly translate to the profiled results of Table 4.1 and Figures 4.10- 4.11. A plot of the profiled speed-up between these GPUs using the balanced tree technique with $k = 4$ is depicted in Figure 4.13. Note that this is a simplistic theoretical estimation and the profiled speedup gaps on smaller table sizes may be due to driver overhead, cache efficiency and other architectural details of the GPUs.

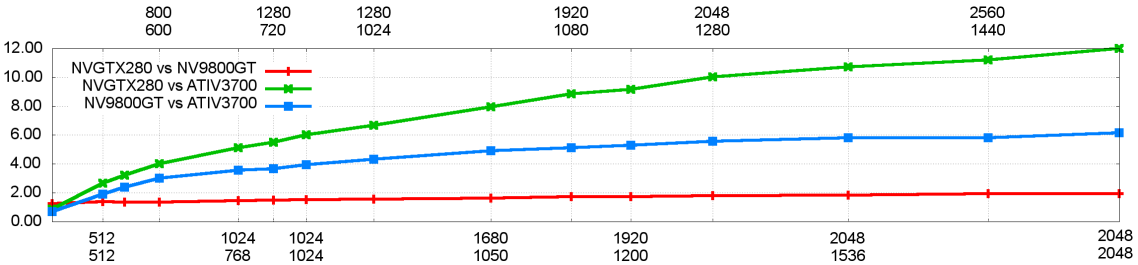


Figure 4.13: Speedup between different GPUs running the balanced tree method ($k = 4$).

	width	height	k	NV GTX 280		NV 9800 GT		ATI V3700	
				RD	BT	RD	BT	RD	BT
1	256	256	2	0.547	0.845	0.675	1.125	1.087	0.695
			4	0.324	0.348	0.414	0.623	1.031	0.449
			6	0.335	0.491	0.442	0.624	1.476	0.546
	65536 pixels		8	0.301	0.442	0.376	0.553	1.393	0.834
2	512	512	2	0.908	1.086	1.406	1.361	4.814	2.375
			4	0.575	0.576	0.909	0.775	5.234	1.563
			6	0.565	0.580	0.878	0.763	6.065	1.896
	262144 pixels		8	0.517	0.604	0.773	0.854	6.099	3.133
3	720	480	2	1.085	1.071	1.770	1.421	6.482	3.121
			4	0.667	0.659	1.130	0.963	6.448	2.056
			6	0.655	0.645	1.066	0.857	7.804	2.449
	345600 pixels		8	0.688	0.716	1.089	1.067	8.702	4.140
4	800	600	2	1.327	1.125	2.336	1.531	9.451	4.381
			4	0.809	0.762	1.405	1.087	9.576	2.797
			6	0.840	0.743	1.360	0.984	11.253	3.395
	480000 pixels		8	0.982	0.646	1.510	1.265	14.478	5.755
5	1024	768	2	1.880	0.986	3.576	1.986	15.806	6.863
			4	1.194	0.747	2.158	1.127	15.913	4.544
			6	1.245	0.550	2.133	1.239	18.310	5.462
	786432 pixels		8	1.448	0.595	2.365	1.696	24.113	9.339
6	1280	720	2	2.224	1.454	4.247	2.178	19.136	8.090
			4	1.434	0.958	2.659	1.457	19.490	5.206
			6	1.417	0.932	2.401	1.485	21.217	6.365
	921600 pixels		8	1.664	1.254	2.714	1.945	27.127	10.953
7	1024	1024	2	2.372	1.525	5.333	2.505	21.475	9.147
			4	1.521	0.997	2.852	1.528	21.813	5.938
			6	1.605	0.977	2.746	1.450	24.625	7.309
	1048576 pixels		8	1.930	1.393	5.046	2.050	32.705	12.431
8	1280	1024	2	2.971	1.604	7.150	2.645	27.846	11.286
			4	1.909	1.081	5.034	1.708	29.119	7.381
			6	1.937	1.069	5.036	1.746	30.560	9.001
	1310720 pixels		8	2.334	1.544	6.240	2.550	39.639	15.445
9	1680	1050	2	3.927	1.841	9.121	3.227	38.111	15.369
			4	2.659	1.242	6.910	2.712	40.271	9.964
			6	2.775	1.258	7.265	3.278	45.270	12.191
	1764000 pixels		8	3.013	2.019	8.264	3.788	53.011	20.916
10	1920	1080	2	4.544	2.073	10.760	3.690	46.433	18.107
			4	3.082	1.305	8.151	3.098	51.601	11.718
			6	3.258	1.411	8.550	3.598	56.400	14.518
	2073600 pixels		8	3.536	2.250	9.767	6.398	67.716	24.708
11	1920	1200	2	4.951	2.077	11.990	4.122	50.909	20.236
			4	3.355	1.418	9.112	3.333	56.965	12.974
			6	3.511	1.520	9.509	4.102	62.554	16.077
	2304000 pixels		8	3.861	2.381	10.760	7.083	75.557	6.805
12	2048	1280	2	5.590	2.159	11.906	4.340	59.458	23.013
			4	3.827	1.462	10.303	3.611	67.791	14.867
			6	4.028	1.740	10.816	4.422	73.091	18.366
	2621440 pixels		8	4.376	2.606	12.250	4.303	87.634	31.133
13	2048	1536	2	6.682	2.492	16.199	5.811	70.588	27.524
			4	4.571	1.653	12.341	4.106	79.338	17.940
			6	5.157	1.882	14.069	5.214	94.266	22.001
	3145728 pixels		8	5.327	3.104	14.721	9.440	104.925	37.528
14	2560	1440	2	7.922	2.805	19.386	5.937	84.384	31.837
			4	5.290	1.821	14.559	4.724	89.137	20.501
			6	6.076	2.245	16.407	5.871	106.202	25.228
	3686400 pixels		8	6.107	3.487	17.224	10.795	117.500	43.309
15	2048	2048	2	8.731	3.112	21.588	7.312	96.113	36.397
			4	5.992	2.007	16.558	5.318	109.207	23.970
			6	6.910	2.306	18.883	6.377	128.502	29.075
	4194304 pixels		8	7.004	3.988	19.740	12.209	142.725	49.442

Table 4.1: Compilation of SAT generation times on different GPUs; **RD** stands for *recursive doubling* and **BT** stands for *balanced tree*. Performance values for both techniques with $k = 2, 4, 6$ and 8 are listed.

4.8 Closing Comments: Limitations, Conclusion and Future Work

Summed-Area Tables comprise a versatile data structure to power a number of applications, ranging from face recognition, rendering and image processing. Summed-Area Tables can be seen as the 2D generalization of 1D *inclusive* prefix-sums (scans). Two parallel patterns for prefix-sum generation map particularly well to GPU-based implementations: recursive-doubling and balanced trees. The balanced tree pattern is more work-efficient than the recursive-doubling pattern, but requires twice as much passes. This trade-off, however, quickly pays-off in favor of balanced trees as the table size increases: in the inspected hardware, speedups of about 2x up to 4x were achieved.

Despite the performance advantages and readily availability of the balanced tree technique since long, the computer graphics community has oddly favored the recursive-doubling pattern up until recently. The first mention of the balanced tree approach in the computer graphics literature is credited to Sengupta et al. [SENGUPTA; LEFOHN; OWENS (2006)]. Quite interestingly, since then several computer graphics researchers that relied on Summed-Area Tables kept using the recursive doubling technique for GPU-based implementations, unaware of the benefits of balanced trees [LAURITZEN (2007); SLOMP; OLIVEIRA (2008); DÍAZ et al. (2010)].

The initial report of Sengupta et al. described direct scan generation using the balanced tree algorithm without the need of deriving it from a prescan [SENGUPTA; LEFOHN; OWENS (2006)]. Curiously, in their subsequent publication, they described only the prescan approach, even though the applications they have analyzed made extensive use of scans (these scans were produced by adding the prescans arrays with their respective original input arrays) [SENGUPTA et al. (2007)].

The main drawback with SAT generation on the GPU using the balanced tree approach is the fact that it requires a considerable amount of intermediate memory ($\approx 3n$) due to the simultaneous read-write texture (global) memory restrictions and the lack of adequate global-level inter-fragment synchronization and communication directives of GPU architectures. As the expected clash between GPU and multi-core CPU architectures comes to a close, such memory access constraints tend to disappear. Current development on general-purpose GPU computing technologies such as CUDA, OpenCL, DirectCompute and C++Amp already started to address these limitations and are paving the road for exciting new prospects. The interoperability overhead between such technologies and regular graphics API is also expected to diminish with future advances.

As a matter of fact, Harris has already investigated implementations of prefix-sums in GPU using the CUDA infrastructure [HARRIS (2007); HARRIS; SENGUPTA; OWENS (2007)]. His implementations exploit several characteristics of NVIDIA GPU architectures and make clever use of local and global CUDA memory. The performance achieved by Harris is on par, and some times faster, than the results profiled with the GLSL-based implementation of this Chapter, but his implementations suffer from some interoperability overhead between CUDA and the graphics API (OpenGL) when applied.

The aforementioned works of Blelloch, Sengupta and Harris describe and use *binary* balanced trees, without mentioning extensions for higher-order balanced trees ($k > 2$). In reality, no work was found in the literature that addressed such higher-order balanced trees, and the extension described in this thesis remains as an original contribution.

4.9 Related Publications

MIKAMO, M. et al. A tone reproduction operator accounting for mesopic vision. In: SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 POSTERS, New York, NY, USA. **Proceedings...** [S.l.: s.n.], 2009. p.41:1–41:1

SLOMP, M.; TAMAKI, T.; KANEDA, K. Screen-Space Ambient Occlusion through Summed-Area Tables. In: FIRST INTERNATIONAL CONFERENCE ON NETWORKING AND COMPUTING (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.1–8. (ICNC '10)

SLOMP, M.; MIKAMO, M.; KANEDA, K. Fast Local Tone Mapping, Summed-Area Tables and Mesopic Vision Simulation. In: MUKAI, N. (Ed.). **Computer Graphics**. [S.l.]: InTech, 2012. p.AAA–BBB

4.A Appendix A: Complexity Analysis

This appendix presents a detailed analysis of the complexity of the two algorithms described in this work to generate prefix sums: recursive doubling and balanced trees. The analysis comprises only the sequential execution, *i.e.*, using a single processor. First, each algorithm is analyzed for the case of arrays of size n, and then they are used to derive the complexity for the summed-area table computation. For simplicity, the ceiling function is omitted, for the cases when $log_k(n) \notin \mathbb{N}$.

For both recursive doubling and balanced tree algorithms, a number k must be chosen and it is assumed to be:

$$2 \leq k \ll n \tag{4.10}$$

The complexity analysis of converting a prescan into a scan and vice versa is not detailed. A shift operation in an array of size n takes $O(n)$ time and the update of a single element in the end of an array is $O(1)$; similarly, an element-wise operation of two arrays of the same size has cost $O(n)$.

4.A.1 Recursive Doubling

The cost of a recursive doubling prefix sum algorithm is $O(n\ log_k(n))$.

The recursive doubling approach requires $log_k(n)$ passes. At each pass $i \geq 1$, a total of $n - k^{i-1}$ elements are updated and, before switching between the arrays, $k^i - k^{i-1}$ elements are copied in order to ensure data stability. The resulting complexity will then be the sum of the updates and copies. Since the number of copies is smaller than the number of updates, the complexity analysis follows just considering the number of updates. At each step, a total of $n - k^{i-1}$ elements are updated. Each of them retrieves k elements from the previous pass and performs $k - 1$ operations. Since the number of operations remains constant for all the elements — and from the assumption on 4.10 — it will be omitted and the analysis follows solely considering the number of updated elements. This statement leads to the following expression:

$$\sum_{i=1}^{log_k(n)} (n - k^{i-1}) \tag{4.11}$$

which can be expressed as:

$$\left[\sum_{i=1}^{log_k(n)} n \right] - \left[\sum_{i=1}^{log_k(n)} k^{i-1} \right] \tag{4.12}$$

and the first sum can be rewritten as:

$$\sum_{i=1}^{log_k(n)} n = n \left[\sum_{i=1}^{log_k(n)} 1 \right] \tag{4.13}$$

in which the expression inside the brackets is an instance of the following summation property:

$$\sum_{i=m}^{n} 1 = n - m + 1 \tag{4.14}$$

and matching the resulting bracketed expression in 4.13 with the property above gives

$$\sum_{i=1}^{log_k(n)} 1 = log_k(n) - 1 + 1 = log_k(n) \tag{4.15}$$

thus plugging it on 4.13 reduces to:

$$\sum_{i=1}^{log_k(n)} n = n\ log_k(n) \tag{4.16}$$

This gives a hint about the complexity of the algorithm. The derivation could be stopped here since the parcels on the second sum of 4.12 does not heavily rely on n (just on a much lower $log_k(n)$) and will not have strength enough to nullify any of the terms of 4.16. However, for a more formal conclusion, the derivation will continue.

The second sum of 4.12 can be rearranged as:

$$\sum_{i=1}^{log_k(n)} k^{i-1} = \sum_{i=1}^{log_k(n)} k^i k^{-1} = k^{-1} \left[\sum_{i=1}^{log_k(n)} k^i \right] = \frac{1}{k} \left[\sum_{i=1}^{log_k(n)} k^i \right] \tag{4.17}$$

where the expression inside the brackets is an instance of a geometric series, defined as:

$$\sum_{i=m}^{n} r^i = \frac{r^{n+1} - r^m}{r - 1} \tag{4.18}$$

where r is the ratio of the progression, and, in 4.17, this ratio is k. Matching the geometric series definition above with 4.17 gives:

$$\frac{1}{k} \cdot \frac{k^{log_k(n)+1} - k^1}{k - 1} = \frac{1}{k} \cdot \frac{k^{log_k(n)} k^1 - k}{k - 1} = \frac{1}{k} \cdot \frac{nk - k}{k - 1} = \frac{1}{k} \cdot \frac{k(n-1)}{k - 1} = \frac{n - 1}{k - 1} \tag{4.19}$$

and finally, by replacing 4.16 and 4.19 in 4.12 gives:

$$\sum_{i=1}^{log_k(n)} (n - k^{i-1}) = n\ log_k(n) - \frac{n - 1}{k - 1} \tag{4.20}$$

which, given the assumption of 4.10, leads to the conclusion that the updates are $O(n\ log_k(n))$.

Now, to finish the analysis, one must add the cost of the copies and updates, $O(n) + O(n\ log_k(n))$, which is $O(n\ log_k(n))$.

4.A.2 Balanced Tree

The cost of a balanced tree prefix sum algorithm is $O(n)$.

The balanced-tree approach requires the building of two trees: one for the reduction stage and another for the expansion one. For a given number of leaves n, the number of cells of a single tree is given by the summation expression below:

$$\sum_{i=0}^{log_k(n)} \frac{n}{k^i} \tag{4.21}$$

but recall that two of them must be generated, so:

$$2\left[\sum_{i=0}^{log_k(n)} \frac{n}{k^i}\right] \tag{4.22}$$

and also recall that the leaves of the reduction tree do not need to be generated, since they comprise the input array of size n - and are already computed - they can be removed from the expression above as follows:

$$2\left[\sum_{i=0}^{log_k(n)} \frac{n}{k^i}\right] - n \tag{4.23}$$

The number of operations in the reduction and expansion stages are not the same: the reduction tree reads k elements and performs $k-1$ operations while in the expansion tree the number of readings depends on the index of the children nodes. From a given child index $1 \leq d \leq k$, a total of d fetches and $d-1$ operations are performed, thus leading to the worst case when $d = k$. Assuming the worst case always, the number of operations performed at each node in both trees will be $k-1$ and remains constant. Relying on 4.10, the analysis then follows solely considering the total of elements produced.

Recall the expression inside the brackets in 4.23. It can be rearranged as follows:

$$\sum_{i=0}^{log_k(n)} \frac{n}{k^i} = n\left[\sum_{i=0}^{log_k(n)} \frac{1}{k^i}\right] = n\left[\sum_{i=0}^{log_k(n)} (k^i)^{-1}\right] \tag{4.24}$$

and since $(k^i)^{-1} = (k^{-1})^i$, it can be conveniently expressed as:

$$\sum_{i=0}^{log_k(n)} (k^{-1})^i \tag{4.25}$$

which is an instance of a geometric series, defined in 4.18, where r is the ratio of the progression, and, in 4.25, this ratio is $k-1$. Matching the expression above with the geometric series definition produces:

$$\sum_{i=0}^{log_k(n)} (k^{-1})^i = \frac{(k^{-1})^{log_k(n)} - (k^{-1})^0}{k^{-1} - 1} = \frac{k^{-1}k^{-log_k(n)+1} - 1}{k^{-1} - 1} \tag{4.26}$$

and since $k^{-log_k(n)} = n^{-1}$, the previous statement simplifies to:

$$\frac{k^{-1}n^{-1} - 1}{k^{-1} - 1} \tag{4.27}$$

for which the occurrences of 1 in the expression above can be replaced by kk^{-1}, since:

$$kk^{-1} = \frac{k}{k} = 1 \tag{4.28}$$

leading to the following:

$$\frac{k^{-1}n^{-1} - kk^{-1}}{k^{-1} - kk^{-1}} = \frac{k^{-1}(n^{-1} - k)}{k^{-1}(1 - k)} = \frac{n^{-1} - k}{1 - k} \tag{4.29}$$

which is equivalent to

$$\left(\frac{1}{n} - k\right)\left(\frac{1}{1-k}\right) = \left(\frac{1 - kn}{n}\right)\left(\frac{1}{1-k}\right) = \frac{1 - kn}{n(1-k)} \tag{4.30}$$

and can be plugged back on 4.24, simplifying it:

$$n\left[\sum_{i=0}^{log_k(n)} (k^i)^{-1}\right] = n\left(\frac{1 - kn}{n(1-k)}\right) = \frac{1 - kn}{1 - k} = \frac{kn - 1}{k - 1} \tag{4.31}$$

thus finally being replaced in 4.23:

$$2\left(\frac{kn - 1}{k - 1}\right) - n = \frac{2kn - 2}{k - 1} - n \tag{4.32}$$

and expressing the above statement in the same denominator gives:

$$\frac{2kn - 2 - n(k-1)}{k-1} = \frac{2kn - 2 - kn + n}{k-1} = \frac{kn + n - 2}{k-1} = \frac{n(k+1) - 2}{k-1} \tag{4.33}$$

which, from the assumption of 4.10, is $O(n)$.

4.A.3 SAT Generation Algorithms

A summed-area table is obtained running multiple instances of prefix sums on each of its rows and then on each of its resulting columns. Given a prefix sum algorithm $A_{scan}(n)$ for 1D arrays of length n, the complexity of an algorithm to generate a summed-area table with dimensions $w \times h$ can be expressed as:

$$h \cdot O(A_{scan}(w)) + w \cdot O(A_{scan}(h)) \tag{4.34}$$

and is also useful to keep in mind that:

$$N = w \times h \tag{4.35}$$

being N the total number of cells of the SAT.

Here follows the complexity analysis for the summed-area table case using the algorithms presented in this appendix:

- *A SAT can be generated using the recursive doubling algorithm in $O(N\ log_k(N))$*:

  plugging the recursive doubling algorithm on 4.34 gives:

  $$h \cdot (w\ log_k(w)) + w \cdot (h\ log_k(h)) = h \cdot w\ log_k(w) + w \cdot h\ log_k(h) \tag{4.36}$$

  thus, from 4.35, the expression above can be written as:

  $$N\ log_k(w) + N\ log_k(h) = N\ (log_k(w) + log_k(h)) \tag{4.37}$$

  and from the following property of the logarithms:

  $$log_k(a) + log_k(b) = log_k(a \cdot b) \tag{4.38}$$

  the former expression can be conveniently expressed as:

  $$N\ log_k(w \cdot h) \tag{4.39}$$

  and again, from 4.35, it reduces to:

  $$N\ log_k(N) \tag{4.40}$$

  which is $O(N\ log_k(N))$.

- *A SAT can be generated using the balanced tree algorithm in $O(N)$*:

  plugging the balanced tree algorithm on 4.34 gives:

  $$h \cdot w + w \cdot h \tag{4.41}$$

  thus, from 4.35, the expression above can be written as:

  $$2N \tag{4.42}$$

  which is $O(N)$.

5 SCREEN-SPACE AMBIENT OCCLUSION THROUGH SUMMED AREA TABLES

5.1 Abstract

There is an increasing demand for high quality real time graphics nowadays. Shadows play an important role to the realism of computer-generated images, enhancing depth, curvature and localization senses. Due to their global nature, shadows introduce overwhelming complexity to rendering algorithms. Recently, screen-space ambient occlusion techniques started to flourish, and are now the de facto standard for real-time dynamic shadow synthesis. A few issues remain, though, such as the sampling quality and noise artifacts. The contributions of this work are two-folded: a variation of screen-space ambient occlusion that uses Summed-Area Tables, yielding to satisfactory results yet performing better than previous attempts, and serves as a new application to the arsenal of Summed-Area Tables.

5.2 Introduction

Computer graphics is playing a major role in contemporary society, being effectively applied on a wide range of subjects: digital entertainment, architectural and product design, engineering and urban planning, medical data visualization and diagnosis, geoprocessing, publicity, to cite a few. Even mundane tasks are unpractical without some sort of graphical feedback (ATMs, web surfing, mobile phones, etc).

Realistic image synthesis is the ultimate goal of computer graphics, and shadows comprise a crucial role to the aesthetics of computer-generated images. Shadows not only enhance realism, but also amplify depth, curvature and localization awareness. In essence, shadows embrace portions of a scene where light can not reach (directly) due to surrounding geometrical interference. This settles shadows in a *global* context: not only light and material properties are relevant, but all objects of the scene as well, which can potentially occlude each other.

Due to this global nature of shadows, typical global illumination algorithms such as ray-tracing [WHITTED (1980b)], radiosity [GORAL et al. (1984)], stochastic path-tracing [KAJIYA (1986)] and photon-mapping [JENSEN (1996)] are inherently capable of producing realistic shadows. However, the high computational costs associated to them compromise their use on large scale scenarios even for offline rendering, not to mention real-time rendering. Amidst, artists often require *quick*[1] rendering feedback tools in order to tweak scenes without incurring into prohibitive productivity penalties.

Exploiting the fact that shadows do not require to be realistic to feel pleasing [TABELLION; LAMORLETTE (2004)], several specialized algorithms were conceived, providing cheap, yet plausible alternatives. One of the most effective techniques devised is *ambient occlusion* (AO) [ZHUKOV; IONES; KRONING (1998)]. The key concept behind it is to estimate light *accessibility* looming from ambient or soft outdoor lighting based solely on enclosing geometric assets. Ambient Occlusion provides important depth and curvature cues that would not otherwise be clear with only direct lighting, as depicted in Figure 5.1. Practical applications of AO are illustrated in Figure 5.2.

Computing Ambient Occlusion takes a fraction of the time of a full radiosity solution, but it is still a computationally demanding operation nonetheless. Up until recently, real-time approximations were only viable through the use of preprocessed static *ambient-map* textures (similar to light-maps), being inherently bound to static meshes.

With the advent of modern programmable graphics hardware, it was natural that some of these methods were adapted to GPU, and AO is no exception. Perhaps the most successful attempt is screen-space ambient occlusion (SSAO) which gauges the light accessibility of pixels by sampling the depth-buffer [SHANMUGAM; ARIKAN (2007); MITTRING (2007); FOX; COMPTON (2008)]. Acceptable performance rates require few samples per pixel, compromising the visibility integral and introducing unpleasant noise artifacts. Although low-pass filters may amend noise, avoiding depth discontinuities and the use of multi-scale filters are costly to evaluate.

The present paper proposes a modification to the core SSAO algorithm that improves sampling quality through the use of a depth-based Summed-Area Table (SAT). Such more

[1] *Quick* in this context could be up to the order of a few seconds or even minutes.

integral sampling strategy minimizes the occurrence of noise and eliminates the need of palliative post-processing addends. The proposed method produces credible results while performing better than previous attempts. The contribution is two-folded, since it represents yet another application for Summed-Area Tables.

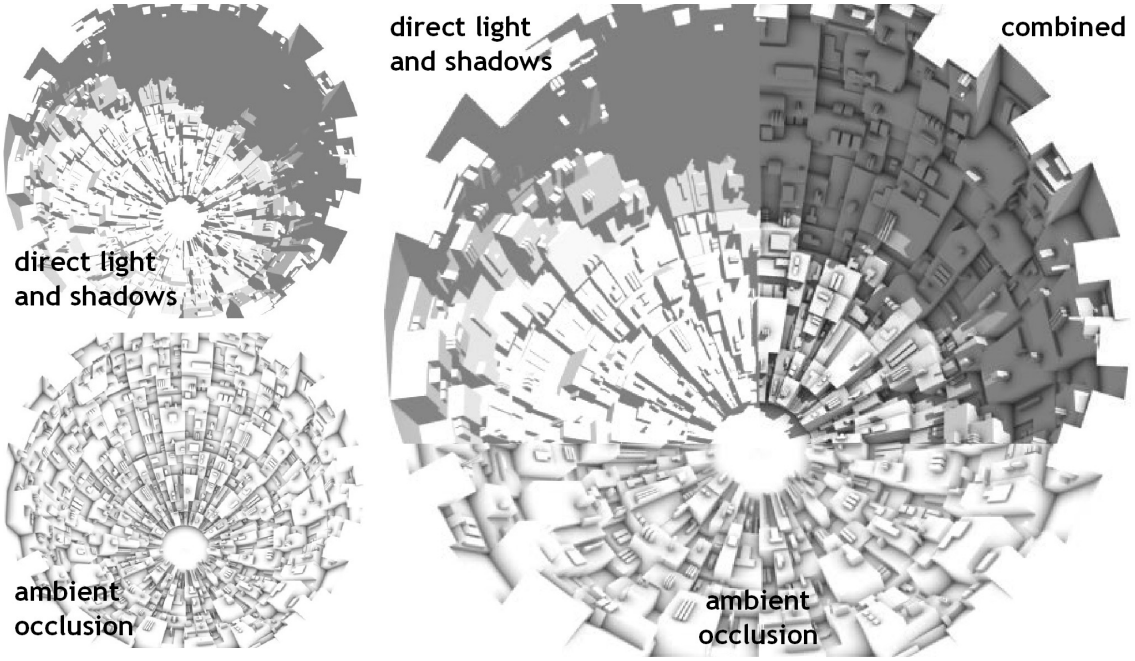


Figure 5.1: Comparison between sharp shadows generated from standard Shadow Maps and soft shadows produced by Ambient Occlusion.

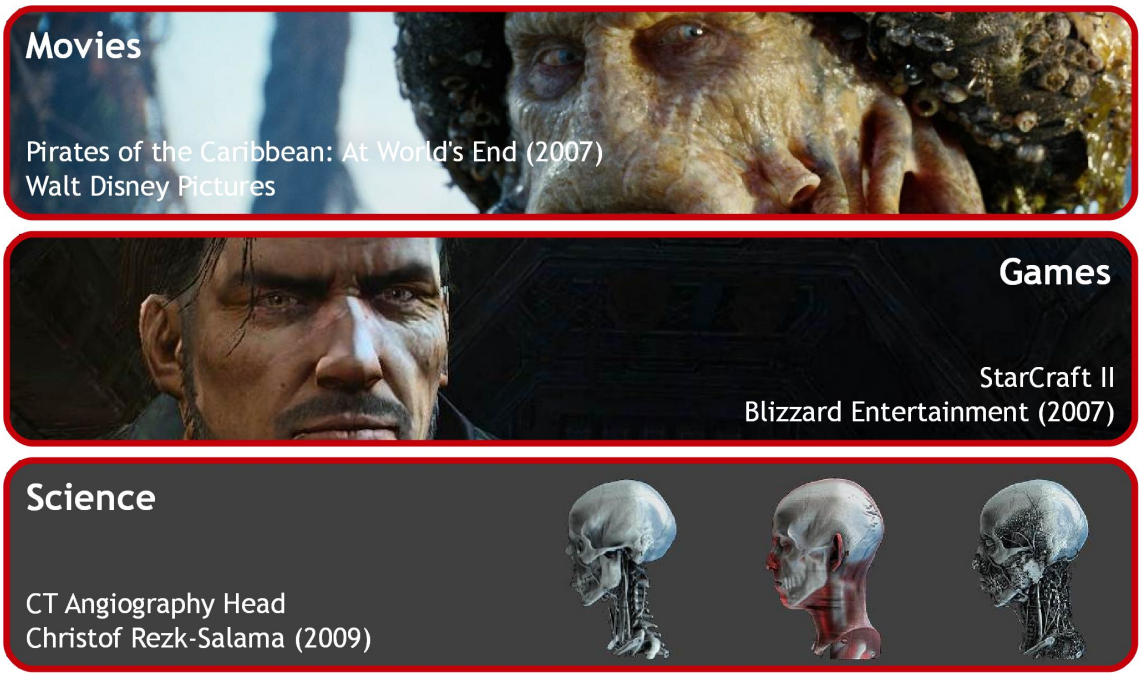


Figure 5.2: Ambient Occlusion techniques have been successfully employed in a wide range of fields, from data visualization to professional games and film production.

5.3 Related Work

Ambient occlusion alone is a quite extensive research topic, despite being relatively new. To keep this reviewing section manageable, related subjects such as classical global illumination, image-based lighting, and other shadow generation algorithms may be referenced, but not discussed.

5.3.1 Ambient Occlusion (AO)

Ambient Occlusion is attributed to ZHUKOV; IONES; KRONING (1998), even though the principles track back a few years [MILLER (1994)]. The method was only popularized later through efforts of film-industry individuals [LANDIS (2002); CHRISTENSEN (2002); PHARR; GREEN (2004)]. Since then, AO has attracted the attention of several graphics researchers and practitioners.

Ambient Occlusion defines the concept of *accessibility* of light at a given surface point [MILLER (1994)]. This can be think as the ratio of the number of directions from which light can arrive at the surface point (that is, *unblocked* directions) over the total number of directions of the visibility hemisphere (see Figure 5.3), formalized as:

$$A(x,\vec{n}) = \frac{1}{\pi}\int_{\Omega} \overline{V}(x,\vec{\omega})\ (\vec{n}\cdot\vec{\omega})\ d\vec{\omega} \tag{5.1}$$

with x and $\vec{n}$ being the position and normal vector of the corresponding surface point, respectively, and Ω being the set of all directions of the visible *hemisphere* with respect to $\vec{n}$. The term $(\vec{n}\cdot\vec{\omega})$ is the geometric attenuation factor (Lambert's cosine term[2]); other literature may refer to this term as $max(0,\vec{n}\cdot\vec{\omega})$ to prevent negative contributions coming from directions below the visibility hemisphere, but since the integration domain Ω in this context is explicitly defined as the visibility hemisphere around $\vec{n}$, the outcome of $(\vec{n}\cdot\vec{\omega})$ will never be negative). Finally, the visibility function $\overline{V}(x,\vec{\omega})$ is simply:

$$\overline{V}(x,\vec{\omega}) = \begin{cases} 0 & \text{if } \vec{\omega} \text{ is blocked by nearby geometry} \\ 1 & \text{otherwise} \end{cases} \tag{5.2}$$

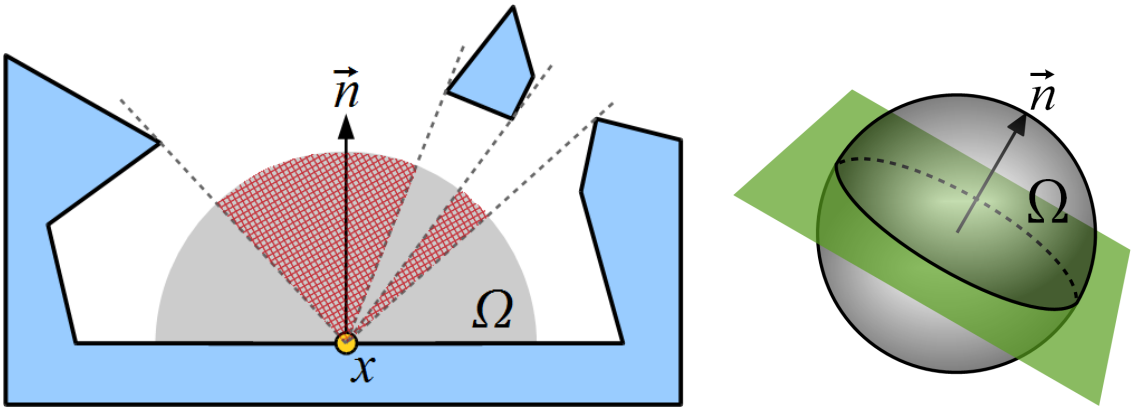
The term $1/\pi$ of Equation 5.1 is a normalization factor to keep the resulting accessibility in the closed range $[0,1]$; the outcome of the integral is a quantity in the range $[0,\pi]$, that is, a projected area in the base (disc) of the corresponding unit hemisphere of area 2π and therefore should be normalized by the total area of the unit disk, that is, π. The projected area is used in this context due to the cosine attenuation factor $\vec{n}\cdot\vec{\omega}$ embedded into the integrand.

Light and material assets are incorporated later, being modulated by the accessibility, as in the shading function below:

$$S(x,\vec{n}) = \rho(x)\cdot L(x)\cdot A(x,\vec{n}) \tag{5.3}$$

where $S(x,\vec{n})$ is the shading function, $\rho(x)$ and $L(x)$ being the *diffuse*[2] material properties

[2] Ambient occlusion was primarily designed for diffuse light transport.



(a) ambient occlusion (2D slice of a scene) (b) visibility hemisphere in 3D

Figure 5.3: Ambient occlusion overview (a): the accessibility of a given point x with normal vector $\vec{n}$ is the ratio between all non-blocked directions (hatched in red) over the total number of directions of a visibility hemisphere Ω around $\vec{n}$ (gray). The image in (b) depicts a visibility hemisphere Ω centered around a normal vector $\vec{n}$ in 3D.

and the amount of light arriving at that particular surface point, respectively.

Combined with image-based lighting [DEBEVEC (2002)], AO produces convincing images at a fraction of the cost of global illumination solutions. Accessibility can be precomputed and stored in vertex attributes or textures; although this permits real-time performance, it is restricted to static geometry. Seeking to relax such constraint, a great deal of research was done, from which some are worth highlighting.

Mesh animation can be performed by blending the precomputed accessibilities of consecutive frames of animation [KONTKANEN; AILA (2006); KIRK; ARIKAN (2007)]. Key-framed deformations are possible, but not arbitrarily. Besides scalability issues (large memory footprints and preprocessing time), external occlusions to the mesh are ignored.

Ambient Occlusion Fields [KONTKANEN; LAINE (2005); MALMER et al. (2007)] evaluates the occlusion potential of objects at specific directions and store them as radial functions onto a small cube-map associated to them. This way, objects can query occlusion from nearby objects based on their relative position and orientation. Objects can move freely, but each object must have only rigid, non-deformable meshes; self-occlusion is handled through traditional AO.

An interesting probabilistic-based AO technique focuses on the rendering of trees (and vegetation, at some extent) [HEGEMAN et al. (2006)]. Although it allows some degree of dynamicity and have little to none precomputation, it is only suitable to individual trees without other nearby geometric occlusions.

Dynamic Ambient Occlusion [BUNNELL (2005)] is a much more robust technique. Instead of operating on polygons, meshes are approximated through a set of disks that can receive, emit, reflect and transmit light. They are placed and combined into a hierarchy during preprocessing. Accessibility is recomputed every frame, traversing the hierarchy for every disk. Additional passes can account for indirect illumination. Although almost entirely GPU-based, managing such hierarchies in video memory is not scalable since the hardware does not perform well when random memory access is extensively required.

5.3.2 Screen-Space Ambient Occlusion

Screen-Space Ambient Occlusion (SSAO) is an attempt to alleviate the aforementioned scalability and rigidity issues via depth-buffer [SHANMUGAM; ARIKAN (2007); FOX; COMPTON (2008); MITTRING (2007)]. The key observation is that the depth-buffer gives an approximation of the geometric neighborhood at each pixel and comes for *free* since it is one of the fundamental stages of the rendering pipeline implemented by the hardware. This limits accessibility calculations to what is seen on the*screen*, but decouples geometric complexity, eliminating mesh constraints and precomputations. SSAO is currently being employed in several cutting-edge 3D engines [Ogre Team; Leadwerks Team; MITTRING (2007); FILION; MCNAUGHTON (2008b)].

The SSAO has its roots on the depth enhancement technique of LUFT; COLDITZ; DEUSSEN (2006). This algorithm performs a low pass filter on the depth-buffer and subtracts it from itself. The difference is then used to improve depth cues, similarly to mainstream image sharpening [GONZALEZ; WOODS (2001)]. Despite the fact of performing exceedingly fast in GPU (for small profiles), it tends to introduce unusual darkening and halo artifacts at deep depth discontinuities.

A more sophisticated SSAO approach was presented by SHANMUGAM; ARIKAN (2007). In this method, occluders are categorized into small nearby elements, which are randomly sampled from the depth-buffer (around the interest point), and larger distant ones, handled in a separated pass through a raster-based deferred accumulation. The small elements are represented as spheres with radii proportional to the corresponding pixels' depths, while the larger ones are circular *billboards* (screen-aligned quadrilaterals with a circular pattern mapped on them). Accessibility is evaluated through the projected solid angle of the elements involved, a rather expensive computation. A similar, but more physically accurate approach was introduced simultaneously by SLOAN et al. (2007).

The efforts of MITTRING (2007) coined the term SSAO in the graphics community. Armed with a simplistic approach, random samples are still taken from the depth-buffer but no costly solid angle calculations are involved. Instead, the number of samples is increased and accessibility is determined simply through the depth difference (and ratio) between the inspected pixel and each sampled pixel around it. Geometric attenuation is possible if normal vector information is available in a G-buffer [DEERING et al. (1988); POLICARPO; FONSECA (2004)] (an example of a deferred G-buffer is shown in Figure 5.4. Hundreds of samples are required for a high-quality output, drastically affecting performance. In practice, however, only a small number (between 8 and 32) is taken for performance reasons, which then incurs in noise artifacts. Due to the low-frequency of AO, a subsequent low-pass filter can reduce such high-frequency flickering; additionally, accessibility can be evaluated and sampled from a smaller depth-buffer since AO tend to vary smoothly. The variants proposed by FOX; COMPTON (2008); FILION; MCNAUGHTON (2008b) only modify how samples are randomized every frame.

The horizon-based SSAO [BAVOIL; SAINZ; DIMITROV (2008)] attempts to improve the accessibility integral evaluation to enhance quality and reduce noise, with a trade-off in performance. It uses ray-marching to determine the horizon's edges, and then integrates over the connected area, as in MAX (1988). Unfortunately, the technique is still biased to the sampling efficacy and although noise is reduced, it is still apparent (further low-pass filtering is still a must).

A similar approach to the one described in this paper was proposed by DÍAZ et al. (2010). They have also used depth-based SAT to estimate the accessibility at each surface point in screen-space, but have not fully exploited the behavior of Equation 5.1 to refine and improve the results. Therefore, their technique is equivalent to the *minimalist approach* of this paper, but lacks the improvements of the *normal-guided sampling* and *depth refinement*, properly introduced in Section 5.4.

Finally, the next logical step of SSAO is to evolve into screen-space global illumination. Indirect illumination [RITSCHEL; GROSCH; SEIDEL (2009)] and self-reflections [ZHAO; YANG (2009)] were already put into proof and, although limited, they pave the road to exciting real-time graphics research opportunities.

5.4 Technique Explained

As with any other screen-space ambient occlusion technique, the proposed method also requires access to the depth-buffer of the scene. In a more simplistic form, a (eye-space) normal buffer is not a requirement, but having one allows for some relevant enhancements, as will be reinforced later. Therefore, the availability of basic deferred shading capabilities is assumed POLICARPO; FONSECA (2004). This particular setup pattern is common to most of the SSAO techniques previously briefed. A typical deferred G-buffer is illustrated in Figure 5.4.

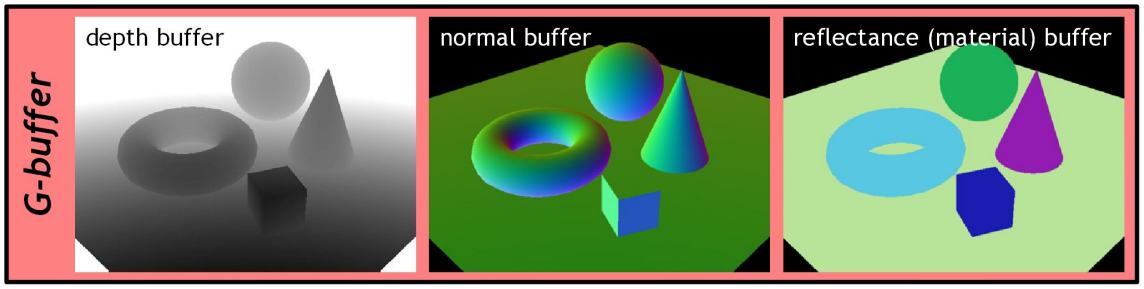


Figure 5.4: An example of a typical G-buffer produced by a deferred shading pipeline, consisting of the depth-buffer, a normal buffer and a material/reflectance buffer.

Recall from Equation 5.3 that lighting and material are handled separately. For that reason, there is no need to allocate buffers into the deferred module to store them. However, as the primitive count of the scene grows considerably, it is a good idea to have some of the shading information (like the material reflectance) encoded in the G-buffer as well, since re-emitting the same geometry on a later shading step may result in a bottleneck.

Once the depth-buffer is ready, what follows is to generate its associated Summed-Area Table. Efficient SAT generation methods on the GPU were already discussed on Chapter 4. The last step is then to evaluate the accessibility of each pixel using the SAT and the G-buffer. A total of three strategies to extract accessibility information from this setup will be introduced in the Subsections to follow, each representing an approximation of Equation 5.1. Accessibility calculations are performed along with the final rendering (shading) step, matching the requirements of Equation 5.3 thus producing shaded pixels that are ready to be displayed. Refer to Figure 5.5 for visual comparison between the three strategies to be presented.

(a) minimalist approach (Subsection 5.4.1)

(b) normal-guided approach (Subsection 5.4.2)

(c) normal-guided with two-level depth refinement (Subsection 5.4.3)

Figure 5.5: Visual comparison of the three methods for SAT-based screen-space ambient occlusion introduced in this Section.

5.4.1 The minimalist approach

A square-shaped region is established around each pixel, from which depth is averaged through the SAT, as illustrated in Figures 5.6 and 5.7. If the *filtered* depth is higher than current pixel's depth[3], no occlusion happens, and the pixel is said to have maximum accessibility. If the resulting depth is lower, this means that there are pixels in that region that can potentially occlude the current one. The absolute difference between them gives a measurement of relative occlusion which, when subtracted from 1, determines the light accessibility at that point.

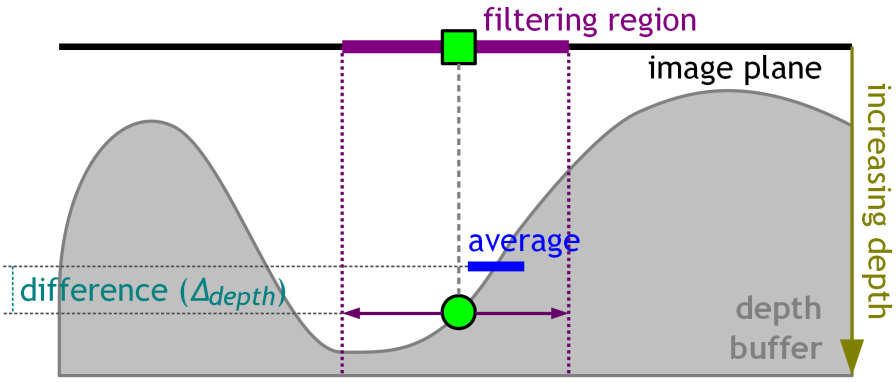


Figure 5.6: The minimalist approach: the amount of occlusion over a pixel (in green) can be approximated through the difference of the average depth around that pixel (in magenta) and the pixel depth itself.

The area of the sampled region could be fixed for all pixels, or proportional to the current pixel depth. The rationale is that pixels lying deep in the scene have less projected visibility hemisphere influence, thus having smaller sampling regions; moreover pixels that lie at the *infinity* (*non-rastered*) will have filtered depths identical to their own depths, leading to maximum accessibility and avoiding depth discontinuities (see Figure 5.7).

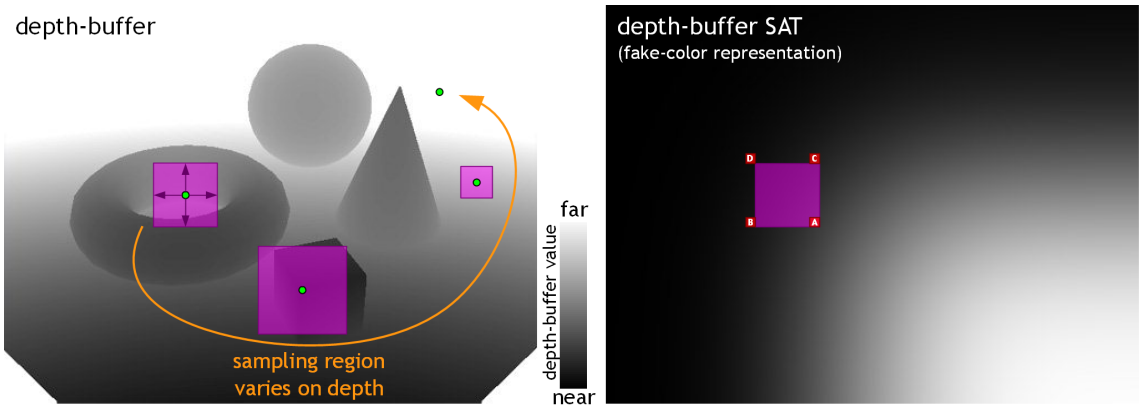


Figure 5.7: A square-shaped area (in magenta) is selected around each inspected pixel (in green); the size of the area depends on the current pixel's depth. This area is then filtered using the SAT and the difference between the filtered depth and the actual depth of the inspected pixel is used to determine the amount of occlusion.

[3]The larger the depth value of a pixel is, the further the pixel is to the image (near) plane.

The occlusion factor alone may not be of significant magnitude to a proper darkening. User-defined attributes can be used to magnify it further. The depth of the current pixel can also be used to estimate further darkening contribution. No general guideline was found to define these attributes appropriately and empirical testing is required.

This approach is very similar to LUFT; COLDITZ; DEUSSEN (2006), with two potential advantages: fast multi-scale filtering (rectangular-shaped box-filter profiles only due to inherent SAT restrictions) and less depth discontinuity issues. An example of this minimalist approach is shown in Figure 5.5-a.

5.4.2 Normal-guided sampling

This approach extends the minimalist one by using orientation information in order to search for better sampling subspaces. The idea is to offset the filtering region along the normal to focus the sampling onto more relevant regions. The rationale is that geometry located around the normal direction will have higher occlusion influence than the ones at glancing angles, as verified by the geometric attenuation factor of Equation 5.1.

The length of the *orthographically projected* normal vector (discarding the eye-space z-component), along with the current depth determines how far the filtering area is shifted. This offset happens along the direction of the projected normal vector. This way, surfaces orthogonal to the viewing direction will have filtering subspaces targeted away from the pixel center and, as before, the deeper a pixel is the less the area needs to shift due to reduced projected visibility hemisphere influence. Refer to Figures 5.8 and 5.9 for an illustrative example.

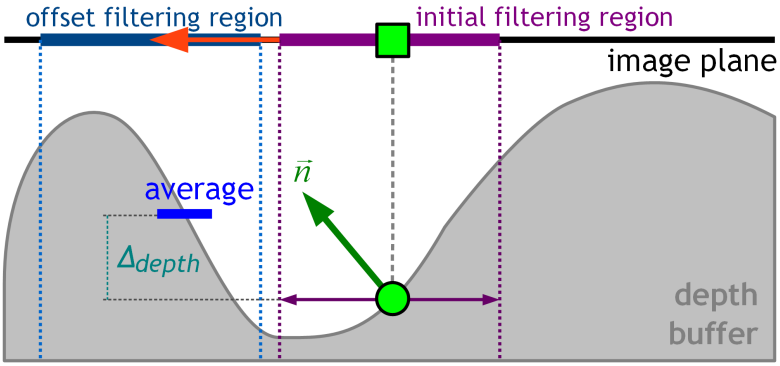


Figure 5.8: Surface orientation can redirect filtering onto a more influential regions. The initial filtering subspace (in magenta) is translated along the direction of the projected normal (orange arrow), resulting in a new filtering area (dark blue).

Although normal-surrounding sampling is present in most SSAO techniques [SHANMUGAM; ARIKAN (2007); FILION; MCNAUGHTON (2008b); BAVOIL; SAINZ; DIMITROV (2008)], they are limited to a small number of samples for performance reasons, with distant pixels being allowed to sample even less. A SAT-based filtering, however, combines all pixels of a subspace without prohibitive increasingly costs, no matter how large the subspace is or where the pixel is. Such tight integration reduces noise, which is a substantial trait observed on other SSAO methods.

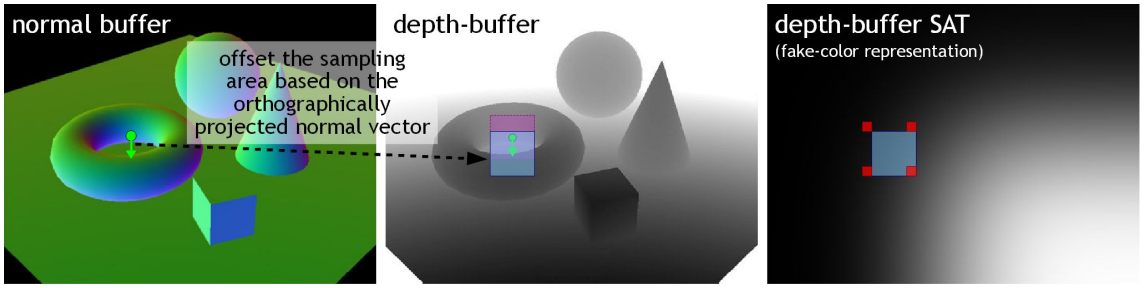


Figure 5.9: The normal-buffer is sampled along with the depth-buffer to determine an offset direction and length (light-green arrow). This offset then relocates the initial sampling area (in magenta) to another location (in light-blue) where the average depth is computed using the associated Summed-Area Table.

The subspace estimated here is more restrictive than the one from the horizon-based technique [BAVOIL; SAINZ; DIMITROV (2008)], in which samples can be procured within a larger field. Even with such sophistication, the local relevance of the samples is prone to increase high-frequency noise since the sample search is randomized. The present approach, however, trades off sampling subspace coverage for a more deterministic, integral methodology that is capable of suppressing flickering up to some reasonable extent. See Figure 5.5-b for an example of this normal-guided technique just explained.

5.4.3 Depth refinement

This is the last approach to be introduced, which can enhance visual quality of both previous attempts. The modification is rather simple: once depth is averaged, the resulting depth is subtracted from the depth of the pixel central to the filtering region, and if the absolute difference lies below some threshold, further subdivision can prevent situations where opposing depth subregions nullify each other (empirical observation suggest thresholds around 15% of the depth of the central pixel). The square-shaped region is refined into four smaller squared-shape areas, similar to a *quad-tree* query. Accessibilities are computed for each subregion, each having contribution proportional to the corresponding area. An example is depicted in Figures 5.10 and 5.11.

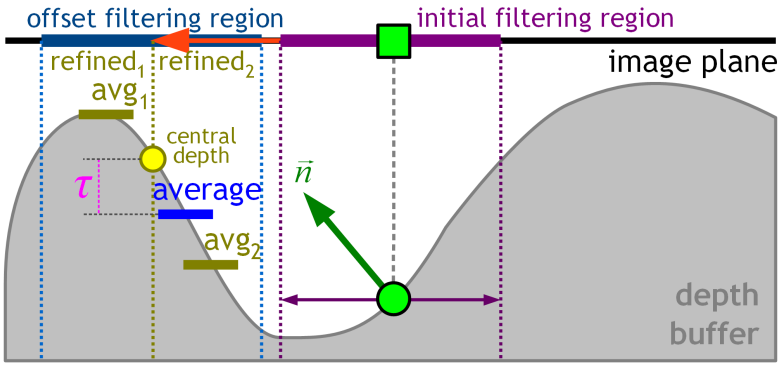


Figure 5.10: Depth refinement can avoid pitfalls such as the one produced when averaged depths cancel each other influences. Once filtered, if the absolute difference between the filtered depth and the pixel central to the region is below some threshold τ, then a depth trap may be in place and subdivision is triggered, seeking to identify more potentially occluding subregions within it.

Each resulting subspace is prone to further subdivisions, yet not recommended to go deeper than two levels. Each SAT filter requires only four texture fetches, but since "recursive" subdivision leads to an exponential behavior, memory bandwidth and latency can become a bottleneck. The amount of SAT samples taken at each subdivision can be largely reduced if one keeps track of the upper-level samples, as depicted in Figure 5.12. This way one can query one level of the quad-tree subdivision with only 9 SAT fetches instead of 20 naïve fetches. Although faster, storing and managing these parent elements for several chained subdivisions is difficult and can quickly consume all of the the fast-access registers and local memory of each shading unit. The process of unrolling the recursion due to the lack of recursion support in GPU is also another complicating factor to consider in the implementation.

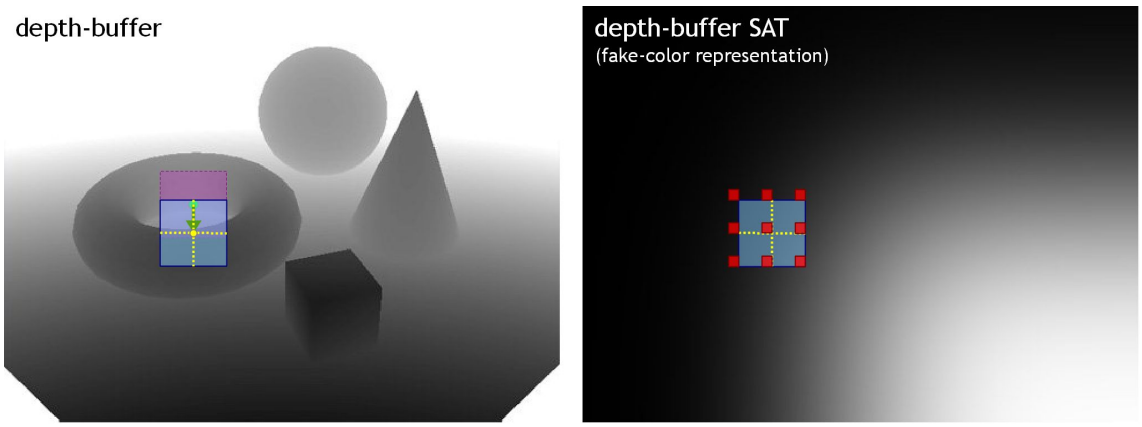


Figure 5.11: Once the initial sampling region is offset by the normal vector, subdivision can take place. Each subdivision step splits the parent area into 4 equally-sized subregions, analogous to a quad-tree subdivision.

The accessibility computation only considers subregions whose average depth are higher than the parent's average depth. The final average is calculated based on a normalized weighted average of the passing subdivided averages, the weight being the area of the subdivided quadrilateral:

$$\Delta_{depth} = \frac{\sum_i \Delta^i_{depth}\ Area_i\ T_i}{\sum_i Area_i\ T_i} \tag{5.4}$$

where T_i is a binary membership function that results in 1 if the average depth of the subdivided area Δ^i_{depth} is larger than the parent's average depth, or zero otherwise. Once the final average depth Δ_{depth} is obtained, accessibility calculation follows using the same difference calculation of the aforementioned minimalist and normal-guided strategies.

Although applicable to the minimalist approach, the overhead of such refinement would nullify the performance gains originally intended for the minimalist method. It is also reasonable to think of an analogous procedure to emulate bilateral filtering, thus minimizing issues regarding depth discontinuities. The proper methodology, however, remains subject of further research.

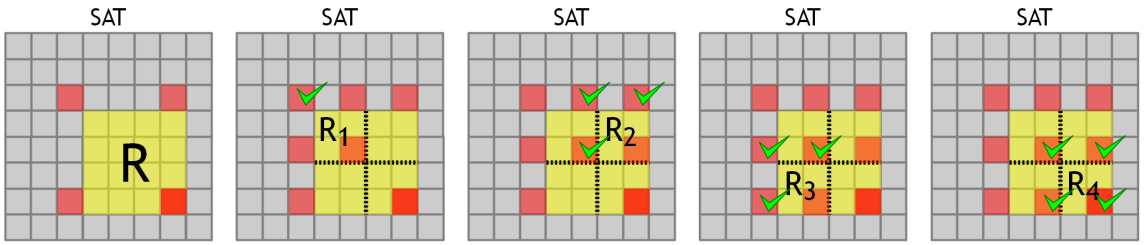


Figure 5.12: Efficient SAT region sub-filtering. Starting from some large region R, the four cells required to filter such area are retrieved. From this point, in order to filter the subregion R_1 one would only need to fetch 3 cells, since the upper-leftmost is already known from its parent region. As for R_2, only one cell must be fetched, since the other cells are already known from its parent and sibling R_1; similarly for R_3. The last subregion R_4 does not require any additional look-ups.

Overall, the presented depth refinement method emulates, at some very localized extent, the occluder hierarchy traversal of BUNNELL (2005) but without the need of maintaining any geometry-related data structure since it is inferred systematically from screen-space depth information. For an example of the depth-refinement method in practice, refer to Figure 5.5-c.

5.5 Implementation Details

This section provides guidelines to implement the proposed algorithm; all hints listed here reflect on simplifications to data representation and manipulation of particular stages without incurring into noticeable quality losses.

Depth-buffers typically hold normalized values withing the range $[0, 1]$. This makes half-float (16bits) texture formats ideal candidates for compact depth-information storage. This design choice does not only reduces memory requirements, but also halves the bandwidth usage through the texture memory interface, potentially accelerating both SAT generation and accessibility calculations by a factor of two.

Caution must be taken when generation the SAT using half-floats on high-resolution depth buffers, since the limited precision and numerical range can introduce noise artifacts due to overflow or accumulated precision errors. The same non-monotonic SAT generation method described in Section 4.6 is recommended to be applied in this context as well.

Quarter-sized depth-buffers are recommended (half of the screen resolution in both horizontal and vertical directions). In fact, most SSAO techniques employ such strategy MITTRING (2007); FILION; MCNAUGHTON (2008b); FOX; COMPTON (2008). Recall that ambient occlusion tend to vary smoothly, on which relevant frequencies (diffuse, low-frequency ones) can be accommodated into smaller scales, at some extent, without noticeable perceptual differences. This heuristic alone can boost performance up to a factor of four times. The SAT-related precision problems pointed above diminish even more since the input table size shrinks (downscaled depth-buffer).

Inverting the depth-buffer values before the SAT generation stage may mitigate precision issues. This ensures that a scene containing large portions where no fragments were produced (thus having zero depth when inverted) will not *disturb* the SAT accumulations. Consider doing such value-inversion while down sampling the depth-buffer as suggested in the last paragraph.

5.6 Results

The results presented and discussed in this section were produced under the following desktop computer profile:

- Operating System: Windows 7 Enterprise Edition 32bit
- CPU: Intel Core2 Quad 32bit CPU Q9400 2.66GHz with 4GB RAM
- GPU: NVIDIA GeForce GTX 280 with 1024MB VRAM (240 shader cores)
- Driver: WHQL certified NVIDIA Display Driver 191.07

The prototype was based on the existing Direct3D 10 framework provided by the NVIDIA SDK sample on Horizon-Based SSAO [NVIDIA Corporation (2009)].

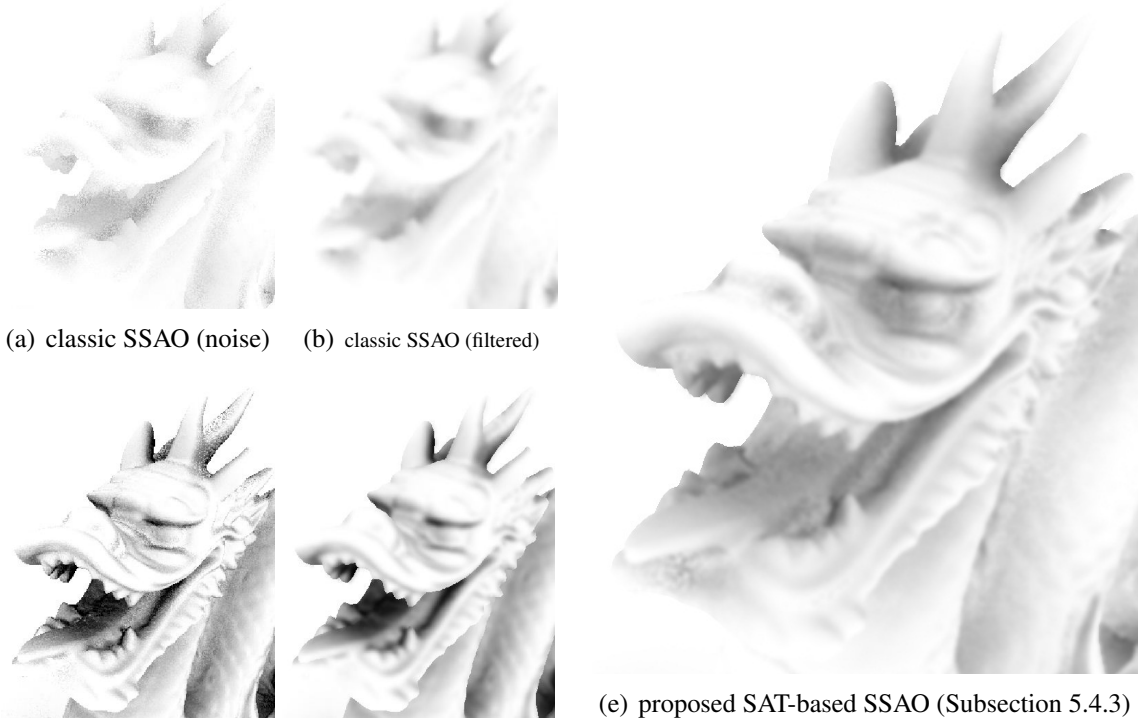
A comparative plate of images is provided in Figure 5.13. The results present in this plate correspond solely to the accessibility values calculated for each pixel. Shading results are available in Figure 5.15, but they allure exclusively to the proposed SAT-based approach. Figure 5.14 also provides some additional examples of the SAT-based method.

The overall qualitative gain of the present method over existing ones is of subjective matter, and the same can be said about other techniques related to ambient occlusion: AO is not designed to provide a physically accurate output, but to generate believable results in a very short span of time.

The horizon-based SSAO may have a better visibility hemisphere coverage, but samples are still procured in a random fashion, which limits the technique to the local relevance of the obtained samples thus being prone to high-frequency noise artifacts. The presented method attempts to reduce such flickering in a more methodical level, integrating all possible samples of a subspace, even though this happens onto a more restrictive coverage field.

Even though some noise can still be noticeable in the images from Figure 5.13 (the SAT-based image inclusive), high-frequency noise is filtered, leaving only smooth transitions. Such low-frequency variations tend to disappear once shading is incorporated on top of ambient occlusion, as shown in Figure 5.15.

It is worth pointing that each SSAO technique have its own empirical set of parameters that can be modified. A particular setup in method may alter the output to something that resembles the default behavior of another one. That being said, even if a result feels too washed or too strong in one of the images of Figure 5.13, parameters could be modified to match a desired effect or an expected contrast.



(a) classic SSAO (noise) (b) classic SSAO (filtered)

(c) horizon SSAO (noise) (d) horizon SSAO (filtered)

(e) proposed SAT-based SSAO (Subsection 5.4.3)

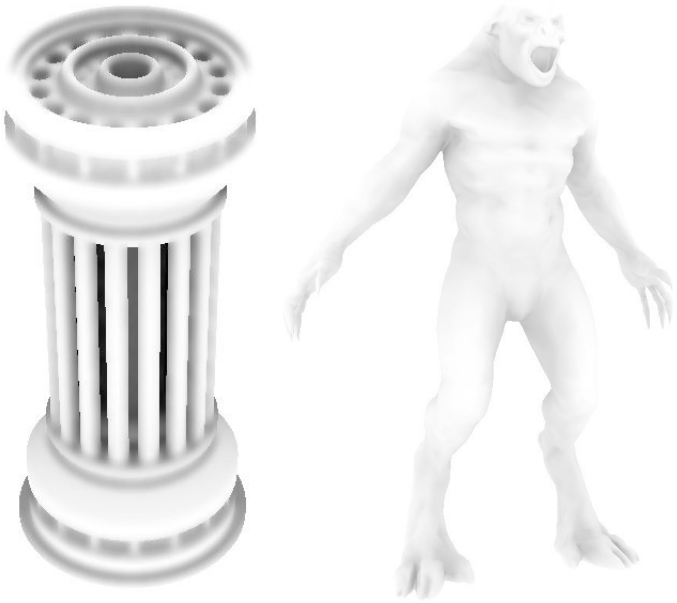
Figure 5.13: Comparative results between traditional SSAO [MITTRING (2007)] (left, upper row) and horizon-based SSAO [BAVOIL; SAINZ; DIMITROV (2008)] (left, bottom row) before and after the low-pass filter is applied. The right-most image is the result obtained with the proposed SAT-based SSAO technique (normal-guided with depth refinement). The SAT-based approach intrinsically performs such filtering along with accessibility calculation, without incurring into any additional performance penalty.

The real strength of the proposed technique is performance. The following table summarizes the overall performance of relevant techniques compared to the proposed one, for a 1024x1024 pixel image (thus with a down sampled depth-buffer of 512x512) under their respective default/suggested settings.

	Traditional: [MITTRING (2007)] [FILION et al. (2008b)] 16 samples; 15x15 blur	Horizon-based: [BAVOIL et al. (2008)] 16 rays; 8 steps; 15x15 blur	SAT-based: normal-guided and depth refined
Speed	130 FPS (≈7.7ms)	90 FPS (≈11ms)	220 FPS (≈4.5ms)

Table 5.1: Performance of different screen-space ambient occlusion techniques.

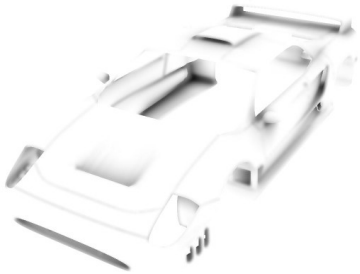
This performance gain is attributed solely to the GPU-friendly characteristics of Summed-Area Table generation methods: mainly, stream-based data-flow and coherent memory access, which enhances the hit-ratio of the texture cache. All results of the proposed method in this document make use of the recursive doubling approach for SAT generation, with four accumulations per pass. Under the hardware profile employed, the corresponding SAT of a 1024x1024 single-channel 32bit floating-point texture (`DXGI_FORMAT_R32_FLOAT`) takes less than 1ms to be computed.



(a) SAT-based SSAO (normal-guided with depth-refinement)

(b) SAT-based SSAO with image-based lighting (IBL)

(c) Phong shading only



(d) SAT-based SSAO only

(e) Phong and SSAO combined

Figure 5.14: Additional examples of the proposed SAT-based SSAO using the depth refinement strategy along with the normal-guided one (Subsection 5.4.3).

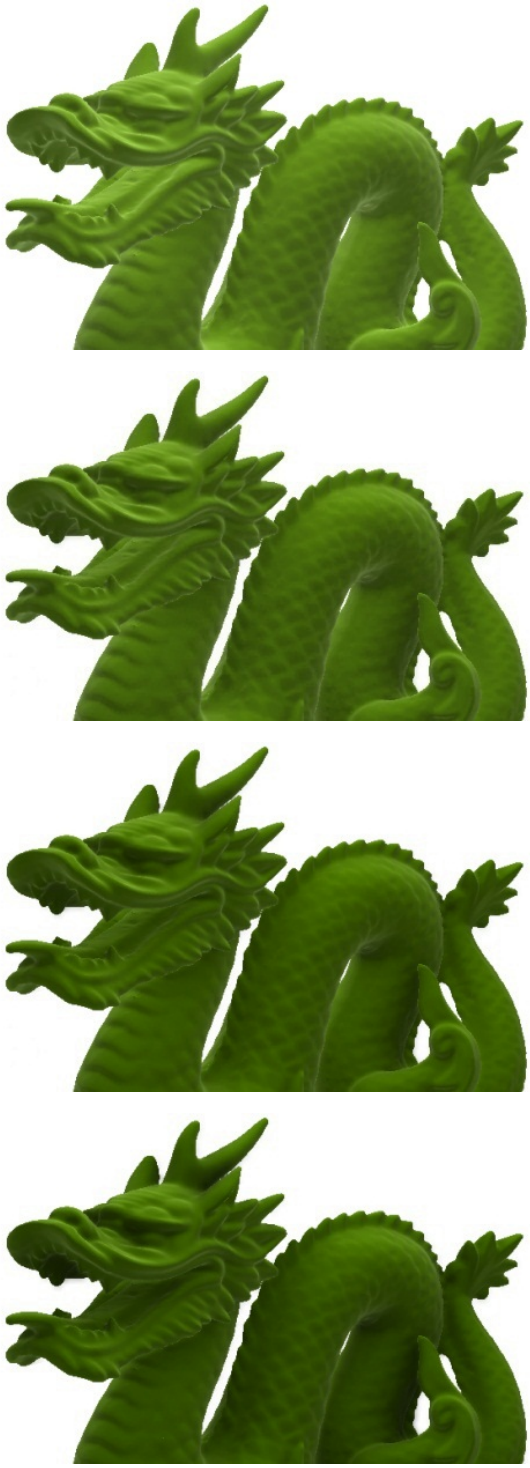


Figure 5.15: The top image represents Phong shading only [PHONG (1975)], without ambient occlusion information. The subsequent images modulate shading with the minimalist, normal-guided and normal-guided with depth refinement approaches, respectively. Note the increasing introduction of darkening features around the mouth, crest and tail of the dragon, progressively enhancing the overall depth perception. (Minor contrast adjustments were applied to these images in order to highlight shadowing details in printed media.)

Finally, the graph depicted in Figure 5.16 shows how the traditional, horizon-based and SAT-based SSAO methods scale to different target resolution settings, using the same algorithmic setup from Table 5.1.

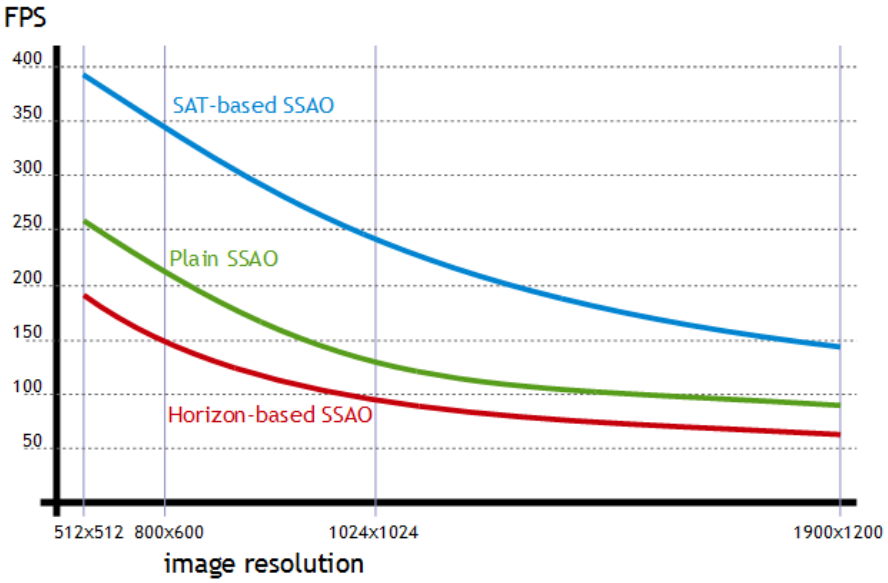


Figure 5.16: Performance scalability under increasing image resolutions. Performance measurements account for all stages, from deferred buffer generation, depth-buffer downsampling, accessibility calculations to shading at last.

5.7 Limitations

The proposed SAT-based strategies for SSAO share the same limitations of DÍAZ et al. (2010): the quality and "'intensity"' of the ambient occlusion shadowing is dependent on the overall scene depth (distance between the near and far planes) and geometric complexity contained in the objects involved. The proper parameter setting is difficult to be determined analytically and empirical experimentation is necessary for each particular scene; an artist tailoring a particular scene should be able to determine the proper parameters and thresholds with some experimentation.

5.8 Conclusion

A modification to the plain screen-space ambient occlusion was introduced. The proposed method provides a more integral sampling strategy, through the use of Summed-Area Tables, which reduces noise and eliminates additional post-processing addends. The proposed method incorporates on-the-fly shadow generation as another potential application to the versatility currently provided by Summed-Area Tables. A total of three strategies were described, with two of them (normal-guided and depth refinement) improving on techniques already available in the literature.

Results have convincing image quality while performance is significantly improved when compared to previous *non*-SAT attempts. Although other methods have superior

horizon coverage, the proposed one trades off this for an improved sampling integration; this is plausible since ambient occlusion itself is not physically accurate thus determining which one feels more *realistic* is of subjective matter. Design guidelines were discussed in order to assist further implementations and maximize pipeline performance. Limitations were also made explicit and justified within their respective context.

Although the limitations might prevent the technique of being used in final production shaders, the performance boost achieved can be helpful to provide faster ambient occlusion effects during modeling or in asset management tools such as visualizers and scene editors.

5.9 Related Publications

SLOMP, M.; TAMAKI, T.; KANEDA, K. Screen-Space Ambient Occlusion through Summed-Area Tables. In: FIRST INTERNATIONAL CONFERENCE ON NETWORKING AND COMPUTING (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.1–8. (ICNC '10)

6 CONCLUSION

This thesis presented GPU-accelerated algorithms for raindrop rendering, ambient occlusion, SoftAssign-based photomosaic optimization, summed-area table generation and mesopic vision simulation. All techniques, save for the photomosaic optimization, can sustain real-time frame rates even in low-end graphics hardware. Even for the case of photomosaic optimization much more reasonable times were achieved, reducing the computing time from hours to a few minutes.

The raindrop rendering technique introduced addresses the rendering of spherical raindrops. Contrary to conventional wisdom, meteorological and physical studies have shown that in typical rainy conditions the shape of the vast majority of the raindrops is in fact spherical. From such observation a particle-based technique was devised, rendering raindrops as screen-aligned quadrilateral sprites. Each sprite samples from a precomputed vector mask and transforms these vectors in run-time based on the viewing attitude, using a fragment shader. Standard, highly-optimized features of the graphics hardware were exploited, such as mip-mapping for efficient interpolation between vector masks at different distances. The proposed technique is substantially faster than previous attempts, animating and rendering millions of raindrops in real-time. Although several streak-based rain rendering techniques already exist, the presented technique is primarily intended for situations where raindrops move in a slower pace, in which streaks are unable to reproduce the specular richness of raindrops. These situations are becoming increasingly popular and include temporal effects such as slow-motion, instant-replays and paused simulations, or slow-moving raindrops on surfaces or windows.

Photomosaic optimization not only improve the overall quality of photomosaics but also greatly reduces the amount of necessary tiles for a faithful reproduction. The three optimization strategies studied – greedy search, simulated annealing and SoftAssign – can be tailored to impede tiles of appearing repeatedly, thus preventing the photomosaic of becoming locally biased. The SoftAssign approach was found to offer the best trade-off between quality and performance. The proposed GPGPU implementation of SoftAssign mapped on the graphics pipeline achieves speedups of 30x up to 60x when compared to an optimized CPU implementation, essentially turning long hours of computation into a few minutes. This "traditional" GPGPU implementation maps particularly well to the GPU, exploiting texture filtering to accelerate computations; as a result performance advantages when faced against a CUDA-based GPGPU implementation are up to the order of 2x. This performance boost is a welcomed feature since identifying the ideal parameter set for SoftAssign requires experimentation, a laborious, tedious task with little guidelines.

Another topic covered by this thesis is prefix-sum and summed-area table generation on the GPU. An improved version of the binary balanced tree algorithm was introduced, generalizing the process to k-ary balanced trees. This improvement can provide significant performance gains, but the optimal value for k depends on several characteristics of the underlying hardware. This requires some experimentation, but once the ideal parameter is found it can be generally used for that particular hardware without any special cases. Along with this proposed improvement, another goal of the study is to popularize the balanced tree technique since the graphics community has been oddly unaware of it, favoring recursive-doubling instead. As demonstrated, the balanced tree approach is much more work-efficient and attractive on the performance spectrum, with gains up to 3x. The downside, at least for a GPGPU implementation on current generation hardware, is the amount of intermediate memory required. The proposed extension to k-ary balanced trees helps to reduce the necessary intermediate memory.

In the Appendix A, an efficient per-pixel approach to reproduce the Purkinje's blue-shift effect under mesopic vision conditions was introduced. The filter decouples chrominance adjustments from luminance compression, thus allowing the filter to be used in conjunction with existing tone mapping operators – given that the operator is able to provide local absolute measurements of luminance. The blue-shift is simulated by suppressing the responses coming from red intensities according to local luminosity conditions. These shifts are evaluated based on psychophysical observations and recent physiological evidence, and can operate on traditional HDR imagery. The filter can be implemented entirely on the GPU and introduces a mostly negligible performance overhead to the complete tone-mapping process.

Finally, a modification to the plain screen-space ambient occlusion was introduced. The proposed method uses Summed-Area Tables to provide a more integral sampling of a pixel's depth neighborhood. As a result, noise is greatly reduced, thus eliminating the need of post-processing low-pass filtering. Three strategies were described, with two of them, normal-guided and depth-refined, improving on SAT-based SSAO techniques already present in the literature. Results have convincing image quality and attractive performance, but the technique is limited to the depth range of the scene presented on the screen. As the distance between near and far planes increases, the lower will be the ambient shadows produced at a finer geometric level. This limitation may prevent the method of being used in final production stage, but the performance is still alluring for fast ambient occlusion effects during modeling or asset management stages.

COMPLETE LIST OF PUBLICATIONS

SLOMP, M. et al. Photorealistic real-time rendering of spherical raindrops with hierarchical reflective and refractive maps. **Journal of Computer Animation and Virtual Worlds (CAVW)**, [S.l.], v.22, p.393–404, August 2011

SLOMP, M. et al. GPU-based SoftAssign for Maximizing Image Utilization in Photomosaics. **International Journal of Networking and Computing (IJNC)**, Los Alamitos, CA, USA, v.1, p.211–229, July 2011

SLOMP, M.; MIKAMO, M.; KANEDA, K. Fast Local Tone Mapping, Summed-Area Tables and Mesopic Vision Simulation. In: MUKAI, N. (Ed.). **Computer Graphics**. [S.l.]: InTech, 2012. p.AAA–BBB

鴨道弘, Marcos Slomp, 玉木徹, 金田和文: 「薄明視における視覚特性を考慮したトーンリプロダクション手法」, 画像ラボ, vol.22, no.6, pp.52-58 (2011 06).

三鴨道弘, Marcos Slomp, 玉木徹, 金田和文: 「薄明視における視覚特性を考慮したトーンリプロダクション」映像情報メディア学会誌, Vol.64, No.9, pp.1372-1378 (2010).

SLOMP, M.; TAMAKI, T.; KANEDA, K. Screen-Space Ambient Occlusion through Summed-Area Tables. In: FIRST INTERNATIONAL CONFERENCE ON NETWORKING AND COMPUTING (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.1–8. (ICNC '10)

MIKAMO, M. et al. Maximizing Image Utilization in Photomosaics. In: FIRST INTERNATIONAL CONFERENCE ON NATURAL COMPUTATION (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.275–278

SLOMP, M. et al. Photorealistic Real-time Rendering of Spherical Raindrops with Hierarchical Reflective and Refractive Maps. In: ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES (I3D), New York, NY, USA. **Proceedings...** ACM, 2010

TAMAKI, T. et al. CUDA-based implementations of Softassign and EM-ICP. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR) DEMOS, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010

MIKAMO, M. et al. A tone reproduction operator accounting for mesopic vision. In: SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 POSTERS, New York, NY, USA. **Proceedings...** [S.l.: s.n.], 2009. p.41:1–41:1

Marcos Slomp, Matthew W. Johnson, 玉木徹, 金田和文: "Photorealistic Real-time Rendering of Spherical Raindrops with Hierarchical Reflective and Refractive Maps", Visual Computing/グラフィックとCAD合同シンポジウム2009 予稿集, 旭川市勤労者福祉総合センター, 北海道(2009 06).

三鴨道弘, 島田洋輔, Marcos Slomp, 玉木徹, 金田和文: 「薄明視における人間の視覚特性を考慮したトーンリプロダクション手法」, 画像電子学会第243回研究会講演予稿, 08-05-27, pp.155-161, 鹿児島大学, 鹿児島(2009 03).

TECHNICAL ACKNOWLEDGMENTS

I would like to thank the open source initiatives of **DevIL**[†] (*Developer's Image Library*, former OpenIL), **GLEW**[‡] (*The OpenGL Extension Wrangler Library*), **boost**[§], **Scilab**[¶], **LibreOffice**[‖], **gnuplot**[**], **MiKTeX**[††] and **TeXnicCenter**[‡‡] for their generosity and excellence in their open-source contributions that accelerated the development of the research projects contained in this thesis, as well as the writing of this document.

Special thanks to: Microsoft for providing Windows 7 Enterprise and Visual Studio 2010 Professional free of charge for students; NVIDIA and AMD/ATI for graphics SDK, examples, debugging and profiling tools; Beepa, oZone3D, RealTech-VR and Graphics Remedy for providing quality freeware software such as Fraps, GPU Caps Viewer, OpenGL Extensions Viewer, gDEBugger.

I would also like to sincerely thank the Munsell Color Science Laboratory, Industry Graphics, Gregory Ward, Mark Fairchild (Mark Fairchild's HDR Photographic Survey), Erik Reinhard, Paul Debevec, Jack Tumblin, Yuanzhen Li, Dani Lischinski, Laurence Meylan and many others for generously making their HDR imagery data-set freely available for researchers. Many thanks to the Stanford Computer Graphics Laboratory for the 3D Scanning Repository and for the anonymous contributors of the TurboSquid free 3D model database.

This research was supported by Japan's Ministry of Education, Culture, Sports, Science and Technology (Monbukagakusho/MEXT).

[†] http://openil.sourceforge.net
[‡] http://glew.sourceforge.net
[§] http://www.boost.org
[¶] http://www.scilab.org/
[‖] http://www.libreoffice.org
[**] http://www.gnuplot.info
[††] http://miktex.org
[‡‡] http://www.texniccenter.org

A FAST AND ROBUST SPATIALLY-VARYING MESOPIC VISION SIMULATION FOR TONE MAPPING OPERATORS

A.1 Abstract

High Dynamic Range (HDR) Imaging is receiving special attention recently due to its more accurate representation of real-world light intensities. HDR has been successfully employed in digital photographs, films and in real-time rendered images. HDR capabilities are even being embedded in layman devices such as smart-phones. In order to display HDR imagery on commodity display devices, tone mapping through either a global or local operator is necessary. Local operators can preserve much more contrastive details at the cost of a much more elevated computational cost; real-time performance is challenging to be achieved with local operators, even when aided by the parallel power of modern programmable graphics hardware (GPU). Moreover, most tone reproduction operators focus solely on luminance compression, ignoring other important chromatic characteristics of the Human Visual System (HVS) such as color-shifts under mesopic vision. This Chapter describes a technique to reproduce mesopic vision in a spatially-varying fashion. The proposed filter builds upon a previous spatially-uniform mesopic vision filter and an approximation of the local photographic operator to evaluate mesopic color shifts in a per-pixel basis. The filter only incurs in a very small performance overhead to the overall tone mapping process and sustains real-time frame rates in current graphics hardware and current high-definition screen resolutions. Chrominance alteration happens orthogonally to luminance compression thus being adaptable to other tone mapping operators as well.

A.2 Introduction

High dynamic range (HDR) imaging is becoming an increasingly popular practice in computer graphics, bringing unprecedented levels of realism to computer-generated imagery and rich detail preservation to photographs and films. HDR imagery can embrace more accurately the wide range of light intensities found in real scenes than its counterpart, low dynamic range (LDR), which are tailored to the limited intensities of display devices. Simply put in computer terminology, think of HDR as a *very large*, *continuous* range of intensities encoded in a floating-point representation (but not necessarily), while LDR translates to *coarse*, *quantized* ranges, usually encoded as 8-bit integers and thus limited to 256 *discrete* intensities.

An important term in HDR imaging is the *dynamic range* or *contrast ratio* of a scene, representing the distance between the lowest and highest intensity values. Luminance in the real world typically covers 14 orders of magnitude (dynamic range of $10^{14} : 1$), ranging from direct sunlight (10^5 up to $10^8 cd/m^2$) to shallow starlight (10^{-3} down to $10^{-6} cd/m^2$), while typical image formats and commodity display devices can cope with only 2 up to 4 orders of magnitude (maximum contrast ratio of $10^4 : 1$) [LEDDA et al. (2005)].

A challenging task on HDR imaging consists in the proper presentation of the large range of intensities within HDR imagery in the much narrower range supported by display devices while still preserving contrastive details. This process involves intelligent luminance (dynamic range) compression techniques, referred to as *tone reproduction operators* or, more commonly, *tone mapping operators* (TMO). An example is depicted in Figure A.1.

(a) No Tone-Mapping (clamped) (b) Global Operator (c) Local Operator

Figure A.1: A comparison between the global and the local variants of the photographic operator of Reinhard et al. (2002). Local operators are capable of preserving more contrastive detail.

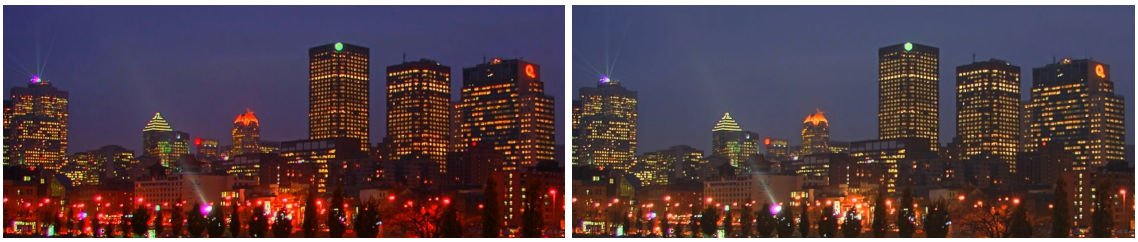
The process of tone mapping shares similarities with the light adaptation mechanism performed by the Human Visual System (HVS), which is also unable to instantly cope with the wide range of luminosity present in the real world. Even though the HVS is only capable of handling a small range of about 4 or 5 orders of magnitude at any given time, it is capable of dynamically and gradually shift the perceptible range up or down, appropriately, in order to better enclose the luminosity range of the observed scene [LEDDA; SANTOS; CHALMERS (2004)]. Display devices, on the other hand, are much more restrictive, since there is no way to dynamically improve or alter their inherently fixed dynamic range capabilities.

Tone mapping operators can be classified as either *global* (spatially-uniform) or *local* (spatially-varying). Global operators process all pixels uniformly with the same parameters, while local operators attempt to find an optimal set of parameters for each pixel individually, often considering a variable-size neighborhood around every pixel; in other words, the amount of luminance compression is locally adapted to the different regions of the image. Determining such local vicinities is not straight-forward and may introduce strong haloing artifacts if not handled carefully. Although all local tone-mapping operators are prone to this kind of artifacts, good operators try to minimize their occurrence. Overall, local operators retain superior contrast preservation over global operators, but at much higher computational costs. Figure A.1 illustrates the visual differences between each class of operator.

Due to the prohibitive performance overhead of local operators, current HDR-based real-time applications rely on either global operators or in more simplistic exposure control techniques. Such exposure mechanisms can be faster than global operators, but are not as automatic, often requiring extensive manual intervention from artists to tailor the parameters for each scene and vantage point; worse yet, if objects or light sources are modified, this entire tedious process would have to be repeated.

Apart from contrast preservation and performance issues, most tone-mapping operators focus exclusively on luminance compression, ignoring chromatic assets. The HVS, however, alters color perception according to the overall level of luminosity, categorized as *photopic*, *mesopic* and *scotopic*. The transition between these ranges is held by the HVS, stimulating visual cells – cones and rods – according to the overall lighting conditions. Cones are less numerous and less responsive to light than rods, but are sensitive to colors, quickly adapt to abrupt lighting transitions and provide sharper visual acuity than rods.

At photopic conditions (think of an outdoor scene at daylight) color perception is accurate since *only* cones are being stimulated. When the lighting conditions turn to scotopic (think of starlight), colors can no longer be discerned because cones become completely inhibited while rods become fully active. *Mesopic vision* is a transitory range in-between where rods and cones are both stimulated simultaneously. At this stage colors can still be perceived, albeit in a distorted fashion: the responses from red intensities tend to fade faster, thus producing a peculiar *blue-shift* phenomenon known as Purkinje effect [MINNAERT (1954)].



(a) without mesopic vision simulation (b) with mesopic vision simulation (proposed)

Figure A.2: An evening urban scene (a) without mesopic vision simulation and (b) with the mesopic vision strategy later described in Section A.5. As it can be seen the sky changes from purple to a more blueish tone, while distant artificial lights shift from red to orange and yellow.

Moonlit scenes, for example, present a blueish appearance even though the light being reflected by the moon from the sun is not anywhere close to blue in nature, but it is actually *redder* than sunlight [KHAN; PATTANAIK (2004); HULST (1957)]. Besides this overall blueish appearance at extreme mesopic vision conditions, the same phenomenon also causes otherwise red features to appear in a much darker tone, or in an *orangeish* or *yellowish* tone; similarly, purple tonalities tend to be noticed in dark blue colorations. Refer to Figure A.2 for a depiction of a scene with and without the proposed mesopic vision filter.

The explanation of such effect comes from the fact that in mesopic conditions rods respond better to short wavelengths (blue) stimuli than long and medium wavelengths (red, yellow, green). As the overall luminosity conditions dim, but *before* rods completely take over the visual system (scotopic vision), color perception shifts towards the currently most sensible rod's wavelengths, that is, around blue.

Mesopic vision reproduction for computer-generated images has immediate application on artistic assets and architectural lighting design. Perhaps an even more relevant application is on road engineering and signalization planning, by reproducing the overall experience of drivers subjected to adverse lighting conditions.

This Chapter focuses on a novel, fast and universal perceptually-based method to reproduce color shifts under mesopic vision conditions. The method builds upon initial investigations of MIKAMO et al. (2009), who suggested the use of perceptual metrics for mesopic vision reproduction derived from the psychophysical experiments performed by IKEDA; ASHIZAWA (1991). The proposed filter extends the former in a *spatially-varying* fashion, that is, performing mesopic color-shifts in a per-pixel basis. This spatially-varying characteristic allows for much more plausible results when strong highlights or light sources are present in the scene. The filter uses the local variant of the photographic tone mapping operator of REINHARD et al. (2002) as its foundation to evaluate local average luminosity for each pixel. The photographic operator has many attractivenesses, but the two main reasons on why it is employed here are: a) its real-time performance when approximated by Summed-Area Tables [SLOMP; OLIVEIRA (2008)]; and b) its perceptually-driven approach for identifying local luminosities through the evaluation of a brightness perception model [BLOMMAERT; MARTENS (1990)]. This does not mean that the proposed filter is bound exclusively to the photographic operator; the filter can also be adapted to suit other tone mapping operators since the chromatic adjustments happen *decoupled* from luminance compression. The filter imposes a tiny performance overhead to the overall tone-mapping process, hence being friendly to real-time applications.

A.3 Related Work

A proper sound review on tone-reproduction operators would consume several pages of this document, since the available literature is vast and rich. Therefore, this background review section will focus on pertinent research related to *real-time local tone-mapping* as well as *mesopic vision simulation*. The curious reader is referred to REINHARD et al. (2010) for an extensive survey on tone-mapping and HDR imaging techniques.

A.3.1 Real-time Local Tone-mapping

Local operators, due to the varying-size filtering and halo-avoidance requirements, impose great challenge for faithful real-time implementations, even with parallel power of modern programmable graphics hardware (GPU). From all existing TMOs, the photographic operator introduced by REINHARD et al. (2002) has received special attention from researchers and practitioners, mainly due to its simplicity (few parameters), automaticity (no user intervention), robustness (extreme dynamic ranges) and perceptually-driven approach (local adaption is guided by an HVS-based brightness perception model). There are two variants of the operator, a global and a local one. The photographic operator is reviewed in Section A.3.3.

The local variant of the photographic operator, at its core, makes use of differences of Gaussian-filtered (DoG) luminance images at various scales. Unfortunately, convolving an image with variable-size Gaussian kernels is a computationally demanding operation. Several attempts were made to accelerate this filtering to provide real-time frame rates.

GOODNIGHT et al. (2003) investigated the implementation of local operators on GPU. Their best result was with the photographic operator, where they implemented the 2D Gaussian convolution using separable 1D kernels in a two-stage approach. Their implementation is by no means naïve, but makes clever use of the efficient 4-component vector dot product instruction provided by GPU architectures. This reduces the number of required filtering passes and thus achieves better performance. Despite all their optimization efforts, the technique only runs at interactive rates when using a small subset of the originally required adaptation scales (i.e., the first few Gaussian-filtered luminance images at small convolution profile sizes; see Section A.3.3). A limited number of adaptation zones gradually causes the operator to fall-back to the global variant case, thus sacrificing important contrastive details (see Figure A.3-b).

KRAWCZYK; MYSZKOWSKI; SEIDEL (2005) proposed an approximation for the local photographic operator on the GPU. The Gaussian-filtered luminance images are downsampled to $1/4$, $1/16$, and $1/64$ of their original size. Convolutions are then performed using smaller approximate Gaussian kernels of fixed size (always 7 pixels wide), with intermediate filtered results being reused. The blurred images are then upsampled back to the original size prior to evaluating the DoG model. This strategy significantly speeds up the operator, but not without inherent limitations: there is excessive blurring across high contrast edges being caused by the downsampling-upsampling process, potentially introducing noticeable halo artifacts (see Figure A.3-c). Besides these shortcomings, their main contributions concentrate on reproducing perceptual effects such as temporal luminosity adaptation, glare and loss of vision acuity.

SLOMP; OLIVEIRA (2008) replaced the expensive variable-size Gaussian convolutions of the operator with box-filtering powered by Summed-Area Tables (SAT) [CROW (1984)]. Summed-Area Tables allow arbitrary rectangular portions of an image to be efficiently box-filtered in constant time $O(1)$. Although box-filtering provides a very crude approximation of Gaussian-filtering, within the context of the photographic operator results are nearly indistinguishable from the original operator, as demonstrated in Figure A.3-d. It is worth mentioning that box-weighted kernels were also used in the context of tone mapping by PATTANAIK; YEE (2002) in their bilateral-filtering algorithm. The process of SAT generation can be efficiently implemented on the GPU, thus

not only mitigating the occurrence of halos, but also substantially accelerating the operator, even when compared against the fast method of KRAWCZYK; MYSZKOWSKI; SEIDEL (2005). A review on SAT and their generation is provided in Sections 4.2, 4.3 and 4.5.

(a) REINHARD et al. (2002) (b) GOODNIGHT et al. (2003) (c) KRAWCZYK et al. (2005) (d) SLOMP; OLIVEIRA (2008)

Figure A.3: Comparison between different implementations of the local photographic operator. Note that words vanished from the book in (b), and halos appeared around the lamp in (c).

A.3.2 Mesopic Vision Simulation

Despite luminance compression, many tone mapping operators have concentrated efforts on reproducing perceptual effects recurrent from the HVS, most notably: temporal luminosity adaption, scotopic vision simulation, loss of visual acuity, and glare. The literature on these topics is broad; refer to REINHARD et al. (2010) for examples of each category. However, and quite surprising, very little has been done on reproducing mesopic vision; below is a summary of the most remarkable research related to mesopic vision in tone reproduction.

DURAND; DORSEY (2000) have specialized the rod-cone interaction model of FERWERDA et al. (1996) to better suit night scenes, adding support for chromatic adaptation and color shifts, among other effects. Even though the method does not explicitly address mesopic vision, the underlying framework can be tunned to handle specific *subranges* of mesopic vision with some degree of fidelity, but not without some user interaction. The technique is fast and runs at interactive rates since it inherits the same *global* characteristics of FERWERDA et al. (1996).

KHAN; PATTANAIK (2004) have also proposed a blue-shift filter based on rod-cone interaction. Their technique is designed primarily for moonlit scenes, without the intervention of artificial light sources. Overall, the method tends to introduce very strong blue hues on the scene, almost completely mitigating other color tones. This is due to their hypothesis that *only* short wavelength cones (blue) would respond to light in a naturally moonlit scene. The technique itself does not rely on HDR imagery and therefore can not be classified as a TMO.

MIKAMO et al. (2009) presented an efficient and universal mesopic vision filter for existing tone mapping operators. Based on the overall average luminance of a scene, the blue-shift is evaluated by suppressing the response of red intensities. The amount of color shift is determined by a global coefficient evaluated through a normalized equivalent lightness curve. The equivalent lightness curve employed comes from the psychophysical experiments of IKEDA; ASHIZAWA (1991) and covers the *entire* mesopic vision range systematically. The filter is universal since chrominance adjustments are decoupled from luminance compression, thus being suitable for any existing tone mapping operator. The filter introduces only a small performance overhead to the entire tone mapping pipeline. Traditional HDR imagery can be targeted without the need of more sophisticated – and less available – HDR image formats. The only drawback is the spatially-uniform nature of the filter which is prone of producing unconvincing results when light sources or strong highlights are present in the scene. This filter is reviewed in Section A.5; more specifically, in Section A.5.4.

KIRK; O'BRIEN (2011) have proposed a color shift model for mesopic vision, building upon the biological model and fitting experiments of CAO et al. (2008). Their model can be combined with existing TMO since chrominance adaptation is decoupled from luminance compression. The chrominance adaption strategy itself is spatially-uniform (global), but nonetheless computationally demanding and incompatible with interactive frame rates. The overhead comes from the fact that *spectral* information is required; in other words, the input images must be capable of approximating the continuous distribution of energy at every pixel using some higher dimensional representation. Moreover, the method assumes that the spectral sensitivity of the camera is known; if not provided by the manufacturer, a calibration procedure is required to estimate the sensitivity. Traditional HDR imagery can also be targeted, although not without first estimating the unknown spectral distribution.

The mesopic vision filter to be introduced in this Chapter extends the filter of MIKAMO et al. (2009) in a spatially-varying fashion. All of the aforementioned advantages of the original global filter are kept. The proposed filters makes use of the local variant of the photographic operator to efficiently estimate local area luminances and thus evaluates color-shifts in a per-pixel basis. Even though the results for the filter in this Chapter are based on the photographic operator, the filter can be applied to other operators as well – that is, given that the operator is able to provide local measurements of luminance. The performance overhead incurred to the entire tone mapping process is most likely negligible. The proposed spatially-varying filter is described in Section A.5; more specifically, in Section A.5.5.

A.3.3 Review of the Photographic Tone Reproduction Operator

REINHARD et al. (2002) digital photographic operator uses a photographic technique called Zone Systems [ADAMS (1983)] as a conceptual framework to manage luminance compression. The goal is to map the *key-value* (subjective predominant intensity) of a scene to the *middle-gray* tone of the printing medium (*middle-intensity* of the display device in this case) and then linearly rescaling the remaining intensities accordingly. This can be intuitively thought as setting an exposure range in a digital camera. An overview of the operator is shown in Figure A.4.

Given an HDR image, a good estimation for its key-value is the *geometric average* (*log-average*) of its luminances, which is less susceptible to small outliers than plain arithmetic average:

$$\widetilde{L} = exp\left(\frac{1}{N}\sum_{x,y} log(L(x,y)+\delta)\right) \tag{A.1}$$

where N is the number of pixels in the image, $L(x,y)$ is the luminance at the pixel with coordinates (x,y), and δ is a small constant (i.e., $\delta = 0.00001$) to prevent the undefined $log(0)$.

Each pixel luminance is then *scaled* based on the Zone System printing zones, with the estimated key-value $\widetilde{L}$ being mapped to the middle-grey range:

$$L_r(x,y) = L(x,y)\frac{\alpha}{\widetilde{L}} \tag{A.2}$$

where $\alpha = 0.18$ for average-key scenes (akin to automatic exposure control systems present in digital cameras [GOODNIGHT et al. (2003)]). For high-key and low-key scenes, the parameter α has to be tweaked, but an automatic estimation strategy is described by REINHARD (2003).

For the global-variant of the operator, each relative luminance $L_r(x,y)$ is mapped to a normalized displayable range $L_d(x,y) \in [0,1)$ as follows:

$$L_d(x,y) = \frac{L_r(x,y)}{1+L_r(x,y)} \tag{A.3}$$

This global operator is prone to conceal contrastive detail. A better approach is to locally adapt the contrast of each region, individually, similar to photographic *dodging-and-burning*:

$$L_d(x,y) = \frac{L_r(x,y)}{1+L_r^{s_{max}}(x,y)} \tag{A.4}$$

where $L_r^{s_{max}}(x,y)$ is the Gaussian-weighted average of the largest isoluminant region s_{max} around each isolated pixel where no substantial luminance variation occur. The term $L_r^{s_{max}}(x,y)$ can be more intuitively thought as a measurement of *local area luminance* around a pixel. It is imperative to judiciously determine these isoluminant regions because otherwise strong haloing artifacts are prone to appear around high-contrast edges of the image.

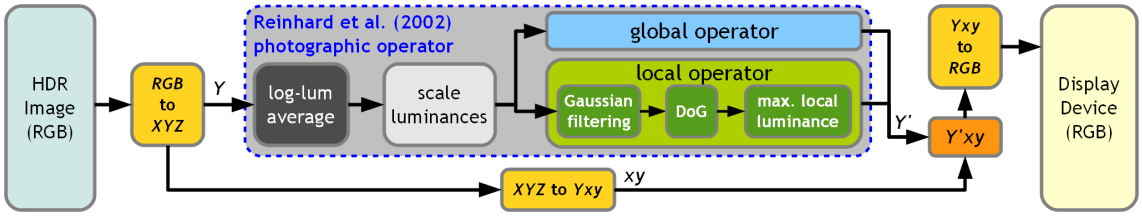


Figure A.4: Overview of the photographic tone mapping operator of REINHARD et al. (2002). The input Y is the HDR luminance ($L(x,y)$) and the output Y' is the compressed luminance ($L_d(x,y)$).

The dodging-and-burning approach of the local photographic operator uses *differences of Gaussian*-filtered (DoG) portions of the scaled luminance image $L_r(x, y)$ at increasing sizes to iteratively search for these optimal isoluminant regions, according to the following expression:

$$V_s(x, y) = \frac{L_r^s(x, y) - L_r^{s+1}(x, y)}{2^\phi \alpha / s^2 + L_r^s(x, y)} \tag{A.5}$$

where ϕ is a sharpening factor, and defaults to $\phi = 8$. The term $L_r^s(x, y)$ corresponds to a center-surround Gaussian-blurred image, formally:

$$L_r^s(x, y) = L_r(x, y) \otimes Gaussian_s(x, y) \tag{A.6}$$

where the operator $\otimes$ denotes the kernel convolution operation and $Gaussian_s(x, y)$ is a Gaussian convolution profile of some scale s centered at pixel coordinates (x, y). The choice for this DoG-based model is not arbitrary and closely follows the human brightness perception model and psychophysical experiments of BLOMMAERT; MARTENS (1990).

Finally, the largest isoluminant scale s_{max} is found by thresholding the Gaussian differences $V_s(x, y)$ obtained from Equation A.5 against the following expression:

$$s_{max} : |V_{s_{max}}(x, y)| < \epsilon \tag{A.7}$$

where $\epsilon = 0.05$ proved to be a good thresholding choice through empirical experimentation, according to REINHARD et al. (2002).

Starting from the initial scaled luminance image $L_r(x, y)$ of Equation A.2, subsequent blurred images $L_r^s(x, y)$ are produced according to Equation A.6 with a kernel about 1.6 times larger than the previous one. This particular scaling factor makes the DoG model resemble a Laplacian of Gaussian filter (refer to REINHARD et al. (2002) for additional details), but can be slightly modified in order to better fit the center-surround convolution profiles. As the differences $V_s(x, y)$ are computed according to Equation A.5, they are thresholded against Equation A.7, stopping as soon as the condition fails. The largest scale is selected if the threshold condition is never reached. In the end, the estimated local area luminance $L_r^{s_{max}}(x, y)$ is plugged back into Equation A.4 for tone mapping.

In general, a total of eight scales (and therefore a total of seven DoG) is sufficient for most situations, but more or less scales can be employed depending on the dynamic range of the input image. The suggested kernel *length* (*not radius*) in pixels, at both horizontal and vertical directions, of the first eight center-surround profiles are: $1, 3, 5, 7, 11, 17, 27$ and 41.

Color information can be removed prior to luminance compression and inserted back afterwards by using the Yxy deviation of the $CIE\ XYZ$ color space [HOFFMANN (2000)]. The Yxy color space is capable of separating luminance and chrominance components. This decolorization and recoloring process is also depicted in the diagram of Figure A.4. Once promoted from RGB to the Yxy color space, the Y component of the Yxy triplet represents the luminance, and directly binds to the luminance function $L(x, y)$. At the end, the compressed luminance Y' (that is, $L_d(x, y)$) simply replaces the original uncompressed luminance Y in the original Yxy triplet, with the chrominance components xy kept intact: $Y'xy$.

A.4 Photographic Local Tone Mapping with Summed-Area Tables

The approximation proposed by SLOMP; OLIVEIRA (2008) to the local photographic operator of REINHARD et al. (2002) suggests the replacement of the costly variable-size Gaussian-filtering by box-filtering. This means that the Equation A.6 of Section A.3.3 gets replaced by:

$$L_r^s(x,y) \approx L_r(x,y) \otimes Box_s(x,y) \tag{A.8}$$

Box-filtering can be efficiently performed through Summed-Area Tables at a fraction of the cost of Gaussian convolutions, requiring only four SAT lookups for any kernel scale s. The input table is, in this case, L_r from Equation A.2, and the corresponding SAT will be referred to as $SAT[L_r]$. Equation A.8 is then rewritten as:

$$L_r^s(x,y) \approx SAT[L_r]_s(x,y) \tag{A.9}$$

where $SAT[L_r]_s(x,y)$ box-filters a square-shape region of L_r centered around the pixel location (x,y) at some scale s using only the contents of the $SAT[L_r]$ itself; in other words, the four pertinent cells of the SAT are fetched and filtering follows as depicted in Figure 4.1-cd.

The set of differences $V_s(x,y)$ from Equation A.5 are performed without any structural alteration, the only change being that $L_r^s(x,y)$ and $L_r^{s+1}(x,y)$ now amount to box-filtered portions of the scaled luminance image instead of the Gaussian-filtered regions. An overview of the modified local photographic operator is shown in Figure A.5.

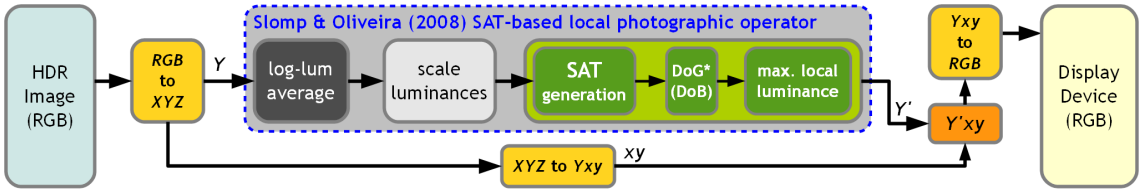


Figure A.5: Overview of the SAT-based local photographic operator of SLOMP; OLIVEIRA (2008).

Visually, results produced with this box-filtering approximation proved to be comparable to the original operator (see Figure A.3-ad). Quantitative analysis using the S-CIELAB metric of ZHANG; WANDELL (1997) was also evaluated in the paper by SLOMP; OLIVEIRA (2008). Typically, the same originally devised filtering scales (listed at the end of Section A.3.3) and threshold $\epsilon = 0.05$ from Equation A.7 can be used. However, since box filters weight the contributions equally, they are more prone to noticeable halos than Gaussian filters of the same scale, which gradually reduces the weights towards the limits of the profiles. If such artifacts become apparent, the authors suggest reducing the threshold ϵ down to 0.0025.

A.4.1 Generating the Scaled Luminance Image on the GPU

Prior to SAT generation, the scaled luminance image L_r should be computed. This process is fairly straight-forward and can be mapped entirely to the GPU. Initially, the

input HDR image is placed into a *float-precision* RGB texture (in the case of synthesized 3D scenes, the entire scene is rendered in such texture target), as depicted in Figure A.6-a.

Following that the RGB texture is rendered into a *luminance-only* (single-channeled) float-precision texture using a full texture-aligned quad. At this stage, each RGB pixel is converted to the XYZ color space, but only the *logarithm* of the *luminance* component Y (plus some δ) is stored in this new texture. This corresponds precisely to the internal summation component of Equation A.1. The summation and average can be evaluated with a full mip-map reduction of this log-luminance image: the single texel of the last mip-map level will be the key-value $\widetilde{L}$ from Equation A.1, *except* for the *exponentiation*. These steps are shown in Figure A.6-bc.

(a) input HDR RGB image (b) log-luminance (c) mipmap (d) scaled luminance L_r

Figure A.6: Efficient generation of the scaled luminance image L_r on the GPU. The logarithm of the luminance (b) of the input image (a) is submitted to a mip-mapping reduction stage (c) and then used to produce the scaled luminance image (d).

Modern graphics hardware and API expose functionalities for automatic and efficient mip-mapping generation (i.e., `glGenerateMipMaps()` in OpenGL). If not supported, the process can be easily emulated on a multi-pass shader approach. Even when available, however, it might be worth implementing a shader-based mip-mapping and take advantage of bilinear filtering capabilities of the GPU, given that the hardware supports bilinear filtering on float-precision textures. By doing so, 16 texels instead of only 4 can be accumulated and averaged per pass. The intermediate mip-map levels are not relevant in this context; only the last one is useful. Besides performance enhancements this also reduces memory consumption.

A source of issues to keep in mind is the use of *non-power-of-two* (NPOT) textures. Current hardware and API have support for NPOT textures, but the implementation may fail to run in older configurations. Padding to the closest POT texture is a robust and popular solution, but at the cost of additional memory. Even with support to NPOT textures, accommodating the NPOT textures into padded POT textures can incur in extra performance gains since the graphics hardware is particularly tailored to handle POT textures more efficiently. If padded textures are used, caution must be taken during the mip-mapping generation to discard the contribution of pixels that lie in the padded region.

Once the mip-map reduction is performed there are essentially two ways of exporting the key-value $\widetilde{L}$ to other stages. The first one is to download the single texel of the last mip-map level from GPU memory to CPU memory, apply the exponentiation function in the CPU, and bind it as an uniform shader parameter to subsequent stages. The second one is to repeatedly sample this texel as needed through shader-level texture access routines

and apply the exponentiation in the shader once the routine returns. The latter is more attractive since reading-back from graphics memory is a synchronous process likely to stall the whole pipeline. One can also apply a dedicated shader to evaluate the exponentiation along with the final reduction.

Finally, a final full texture pass is performed and targeted to yet another luminance-only float-precision texture, this time evaluating Equation A.2. The term $L(x, y)$ is obtained by once again sampling the RGB texture and converting the pixels to the XYZ color space. The key-value $\widetilde{L}$ is accessible through the last log-luminance mip-map level, remembering to take the exponentiation once the texel is fetched. The parameter α is an application-controlled uniform shader parameter. In fact, the base log-luminance texture can be used here as the render target, since it is no longer required by the operator, thus alleviating memory requirements. Refer to Figure A.6-d for a depiction of the resulting scaled luminance image.

At this point, the scaled luminance texture L_r can be either used directly on the global operator or submitted to an efficient SAT generation stage (see Chapter 4.2) from which the resulting SAT can be used to approximate the local operator, as described earlier in Section A.4.

A.5 Mesopic Vision Simulation

This Section reviews the spatially-uniform mesopic filter of MIKAMO et al. (2009) and introduces the proposed spatially-varying extension. These two filters are described in Sections A.5.4 and|A.5.5, respectively.

In order to reproduce Purkinje's blue-shift effect, the mesopic filter of MIKAMO et al. (2009) requires some quantitative estimation on how individual color responses tend to change under mesopic vision conditions. Following that, it is important to be able of reproducing such changes in some HVS-compatible and perceptually-uniform fashion. Therefore, before the proposed mesopic vision operators are properly introduced in Sections A.5.3- A.5.5, a discussion on psychophysical subjective luminosity perception and opponent-color spaces will be presented in Sections A.5.1 and A.5.2.

A.5.1 Equivalent Lightness Curve

IKEDA; ASHIZAWA (1991) have performed a number of subjective luminosity perception experiments under various lighting conditions. In these experiments, subjects were exposed to a special isoluminant room where different glossy colored cards[1] were presented to them. Once adapted to the different levels of luminosity, the subjects were asked to match the brightness of the colored cards against particular shades of gray distributed in a scale[4]. From the data analysis, several curves of *equivalent lightness* were plotted, depicting how the experienced responses of different colors varied with respect to the isoluminant room conditions. The results can be compiled into a single equivalent lightness response chart, as shown in Figure A.7.

As can be observed in the curves, as the overall luminosity decreases, red intensities

[1] Standard, highly calibrated color/gray chips and scale produced by Japan Color Research Institute.

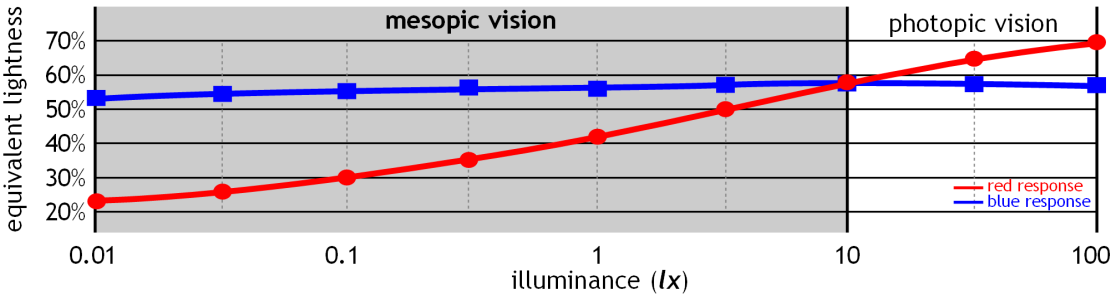


Figure A.7: Equivalent lightness curve for red and blue according to the experiments of IKEDA; ASHIZAWA (1991). These curves summarize the experienced relative brightness from several colored cards against gray-scale patterns in different isoluminant environment conditions by several test subjects.

produce much lower lightness responses than blue intensities, which only slightly varies. This behavior implicitly adheres to the expected characteristics of the Purkinje's effect, that is, the tendency of the HVS to favor blue tonalities at low lighting conditions. Since blue responses are barely affected, finding adequate lowering factors mainly for red responses should be a sufficient approximation, which is the key idea behind the proposed mesopic filters.

The red response curve of Figure A.7 can be approximated by the expression below:

$$E(I) = \frac{70}{1 + (10/I)^{0.383}} + 22 \tag{A.10}$$

The range of interest is the mesopic vision range, that is to say, between $0.01\ lx$ and $10\ lx$. If the equivalent lightness of some overall luminosity $E(\lambda)$ is normalized against the equivalent lightness of the triggering luminance of the mesopic vision range $E(10)$, then the result will yield a coefficient ρ that indicates, in relative terms, how much the response coming from red intensities at such lighting conditions lowers with respect to the starting range $E(10)$:

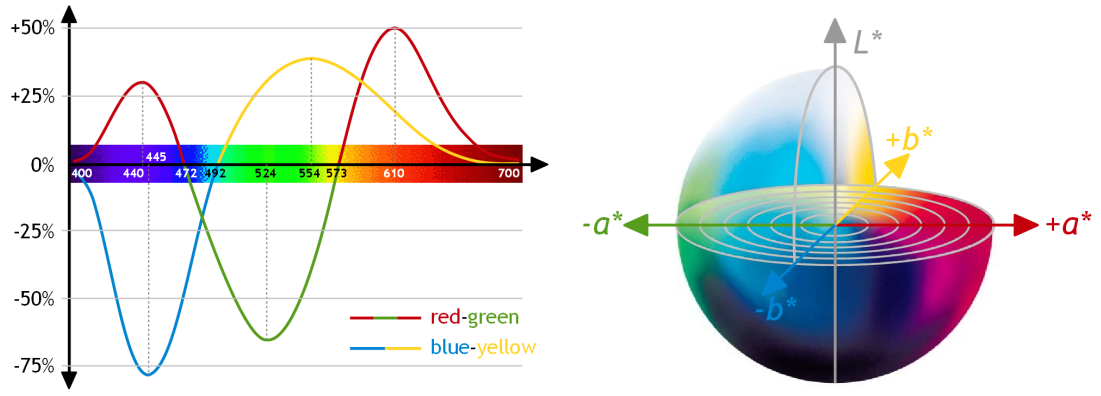
$$\rho(\lambda) = \frac{E(\lambda)}{E(10)} \tag{A.11}$$

In other words, if $\rho(\lambda) \geq 1$, the overall luminosity offers photopic conditions and chrominances *do not* need to be altered. However, if $\rho(\lambda) < 1$ then the overall luminosity lies in the mesopic vision range, and $\rho(\lambda)$ gracefully provides a normalized relative measurement of how much red components should be scaled down in order to reproduce the expected experienced response. These two conditions can be handled uniformly by *clamping* $\rho(\lambda)$ to $[0, 1]$.

A.5.2 Opponent-Color Theory and the L*a*b* Color Space

Recent physiological experiments by CAO et al. (2008) have shown that mesopic color-shifts happen in the opponent-color systems of the Human Visual System, and that these shifts change linearly with the input stimuli. The initial hypothesis behind opponent-color theory in the HVS was proffered by German physiologist Karl Ewald Konstantin Hering and later validated by HURVICH; JAMESON (1955). The key concept is that particular pairs of colors tend to nullify each other responses in the HVS and thus can not to be noticed simultaneously.

Color perception in the HVS is guided by the joint activity of two independent opponent systems: red versus green and yellow versus blue. A plot of these opponent response curves is shown in Figure A.8-a, based on the experiments of HURVICH; JAMESON (1955). Interestingly, luminance perception is also controlled by another opponent system: white versus black.



(a) HVS opponent-colors: wavelength × response

(b) L*a*b* opponent-color space

Figure A.8: Opponent chromatic response of the HVS (a), and the L*a*b* color space (b).

Physiologically speaking, these systems are dictated by opponent neurons which appropriately produce a chain of excitatory and inhibitory responses between the two components of each individual system. Numerically this models to positive feedbacks at some wavelengths being interfered concurrently by negative impulses at their counterparts.

Several color spaces were established based upon opponent-color schemes. The $CIE\ L^*a^*b^*$ (Figure A.8-b) is remarkably one of the best known and widely used of such color spaces since it gauges color distances in a compatible perceptually-uniform (linear) fashion with respect to the Human Visual System. The L^* component represents a *relative* measurement of luminance (thus can not be expressed in cd/m^2), while a^* relates to a red-green opponent system and b^* to a blue-yellow opponent system. For simplicity, throughout the remaining of this Section the $CIE\ L^*a^*b^*$ color space will be shortly referred to as Lab.

The spatially-uniform mesopic vision filter of MIKAMO et al. (2009) – to be introduced in the following Subsection – make use of the Lab color space. Consequently, the proposed spatially-varying filter also makes use of the same color space. The choice for an opponent-color system to apply the color-shifts is supported by the physiological observations of CAO et al. (2008); the Lab space itself was selected because it offers the desired perceptually linear characteristics. Even though external light stimuli is directly *received* by red-green-blue cone-shaped sensors in the retina, the experienced *perceived* color comes from the activity of opponent-color systems wired to these primaries.

A.5.3 Overview of the Mesopic Vision Reproduction Operator

The mesopic vision filter for digital images of MIKAMO et al. (2009) is derived directly from the lightness response curve specialized for mesopic vision of Equation A.10 and from the perceptually-uniform Lab color space. Given a pixel represented in some

color space (i.e., RGB), the process begin by promoting the pixel to the Lab color space.

A Lab pixel holding a *positive* quantity at its a component is actually holding some red intensity (negative a means green, see Figure A.8). Now, if the overall luminosity condition suggests mesopic vision (that is, $\rho(\lambda) < 1$) then this red (positive-a) component should modulated by the normalized coefficient $\rho(\lambda)$ obtained from Equation A.11:

$$a' = a\ \rho(\lambda) \tag{A.12}$$

and this modified quantity a' replaces the original a in the Lab triplet, yielding to $La'b$. This can then be converted back to the initial color space (i.e., RGB) or to any other color-space for presentation purposes.

In order to integrate these chrominance adjustments with HDR luminance compression, another color space must be used as an intermediate, preferentially one that can decouple luminance from chrominance. The Lab space itself is one of such spaces, but since the component L comprises only a relative measurement of luminance, this may cause incompatibilities with tone mapping operators that require proportionally equivalent absolute quantities. The Yxy deviation of the $CIE\ XYZ$ color space is up to this task [HOFFMANN (2000)].

The general algorithm for the filter can be summarized in the following steps:

1. obtain a measurement of the overall luminosity – λ – (Sections A.5.4 and A.5.5)
2. compute the red response coefficient for this luminosity – $\rho(\lambda)$ – (Equation A.11)
3. transform the original HDR pixels to the Lab color space
4. perform the blue-shift by altering the red (positive-a) component – a' – (Equation A.12)
5. transform the modified $La'b$ pixels to the Yxy color space
6. compress the HDR pixel luminance Y using some TMO, yielding to Y'
7. replace the HDR pixel luminance Y by the compressed pixel luminance Y'
8. transform the modified Yxy pixels to the color space of the display device

MIKAMO et al. (2009) evaluated the average isoluminant λ for the step 1 above in a spatially-uniform (per-scene, global) fashion, as will be described in Section A.5.4. The proposed spatially-varying extension computes an individual average isoluminant $\lambda(x, y)$ *for each pixel* of the image, and will be introduced in Section A.5.5.

A.5.4 Spatially-Uniform Mesopic Vision Reproduction Operator

MIKAMO et al. (2009) observed that one way to determine the overall luminosity of a scene is through the log-average luminance (as reviewed in Section A.3.3). However, although this log-average is a suitable candidate, the authors have found that this log-average has the tendency of placing relatively bright images very low into the mesopic

range scale. For that reason, MIKAMO et al. (2009) opted for a simple arithmetic average of the luminances and realized that this produced more plausible indications for the global mesopic scale. This arithmetic average is also more compatible with the equivalent lightness curve of Section A.5.1 which is plotted based on *absolute* luminosity quantities.

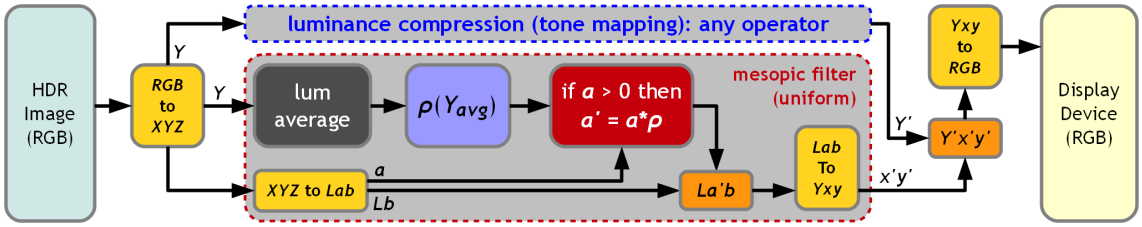


Figure A.9: Overview of the spatially-uniform mesopic vision reproduction filter.

Computing the arithmetic average of the luminances is straight-forward on GPU by using the same mip-map reduction strategy described in Section A.4.1. It is possible to produce this average in advance along with the log-average by reformatting the base log-luminance texture with an additional channel to hold the absolute luminance (`GL_LUMINANCE_ALPHA32F`).

Once evaluated, the average serves as a measurement for the overall luminosity of the entire scene and the algorithm follows as listed in Section A.5.3, applying the *same* response coefficient $\rho(Y_{avg})$ to all pixels, uniformly. The process is illustrated in Figure A.9. An example of this spatially-uniform mesopic filter is shown in Figure A.10-b. Refer to Section A.6 for more examples.

(a) without mesopic simulation (b) spatially-uniform mesopic (c) spatially-varying mesopic

Figure A.10: Comparison between an image without mesopic simulation (a), with global mesopic filter (b) and per-pixel mesopic filter (c). All red intensities shift toward orange/yellow in (b), while only those not sufficiently bright enough change in (c) like the light reflex in the leftmost wall. Also note that purple tones shifted towards a more blueish hue in the mesopic images.

A.5.5 Spatially-Varying Mesopic Vision Reproduction Operator

The main disadvantage of the spatially-uniform filter of MIKAMO et al. (2009) becomes clear when strong red-hued light sources are present in the scene, as is the case of Figure A.10. When the overall luminosity suggests mesopic vision, the intensity coming

from such light sources will be inadvertently suppressed, regardless of their own local intensity; even worse, the higher the intensity, the larger the shift will be. Red light traffic semaphores, neon lights and rear car lights, for example, still hold perceptually strong red intensities which are not noticed as yellow by an external observer, even at the dimmest surrounding lighting conditions. An even more extreme example would be a digital alarm clock equipped with red LEDs: even in complete darkness the LEDs are still perceived as red. This leads to the design of a variant filter that is capable of reproducing mesopic vision in a local, spatially-varying fashion.

In order to counter-act the effects described above, per-pixel local area luminance must be inspected. The good news is that such local measurement is already available from the local variant of the photographic tone mapping operator ($L_r^{s_{max}}(x,y)$). The bad news is that this quantity is based on the relative *scaled* luminance image L_r of Equation A.2, and thus incompatible with the *absolute* scale of the equivalent lightness curve from Section A.5.1.

Fortunately, the scale can be nullified with the inverse function of the scaled luminance L_r:

$$L_r^{-1}(x,y) = L_r(x,y)\frac{\widetilde{L}}{\alpha} = L(x,y) \tag{A.13}$$

The formula above can be generalized to filtered versions of $L_r(x,y)$ at any scale s:

$$L_r^s(x,y)\frac{\widetilde{L}}{\alpha} = L^s(x,y) \tag{A.14}$$

Hence, plugging the *scaled* local area luminance $L_r^{s_{max}}(x,y)$ in the expression above yields to $L^{s_{max}}(x,y)$, which is the *absolute* local area luminance. This quantity is now compatible with the equivalent lightness curve, and it is now possible to determine individual response coefficients $\rho(L^{s_{max}}(x,y))$ for each pixel (x,y), thus enabling localized mesopic adjustments.

An overview of this spatially-varying mesopic filter is depicted in Figure A.11. The application of the filter on a real image in shown in Figure A.10-c; more examples are provided in Section A.6.

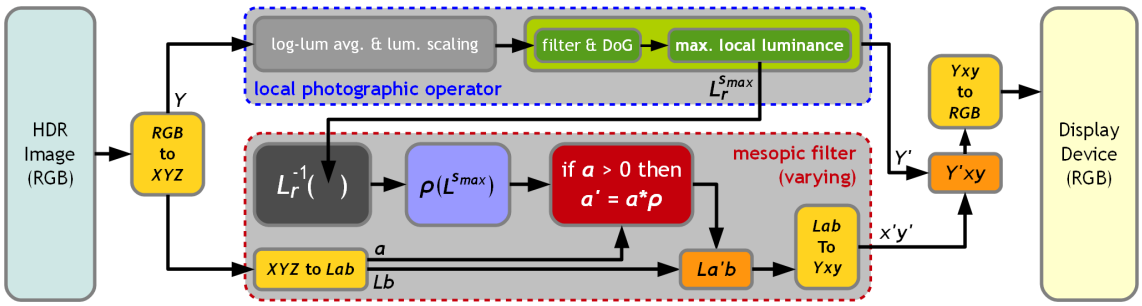


Figure A.11: Overview of the spatially-varying mesopic vision reproduction filter.

The spatially-varying mesopic filter is structurally simpler than the spatially-uniform one since no additional data need to be assembled. The filter is, however, strongly attached to the framework provided by the photographic operator. Although no other local

TMO was explored, most local operators perform estimations of local pixel averages and hence should be able to offer such information and feed it through the filter pipeline of Figure A.11.

An advantage of using the local photographic operator over other local operators for the localized mesopic reproduction is the fact that per-pixel local area luminances are searched through an HVS-based brightness perception model [BLOMMAERT; MARTENS (1990)]. This means that the chromatic adjustments follow some perceptual guidelines, while other operators may not rely at all on HVS features and thus become less suitable for a proper mesopic reproduction.

A.6 Results

This Section analyzes the performance impact of both mesopic filters, spatially-uniform and spatially-varying, on a tone mapping pipeline. Additional examples of the filters described in Section A.5 are also available in Figures A.16 and A.17.

The system configurations and hardware profiles investigated are listed below:

1. Windows 7 Enterprise 32bit SP1 running on an Intel(R) Core(TM)2 Quad CPU Q9499 2.66GHz with 4GB RAM equipped with a NVIDIA GeForce GTX 280 with 1GB VRAM (240 shader cores at 1107MHz, memory at 1296MHz, WHQL Driver 280.26)

2. Windows 7 Enterprise 32bit SP1 running on an Intel(R) Core(TM)2 Quad CPU Q8200 2.33GHz with 4GB RAM equipped with a NVIDIA GeForce 9800 GT with 512MB VRAM (112 shader cores at 1500MHz, memory at 900MHz, WHQL Driver 280.26)

3. Windows XP Professional x64 Edition SP2 running on an Intel(R) Xeon(R) CPU W3520 2.67GHz with 8GB RAM equipped with an ATI FirePro 3D V3700 with 256MB VRAM (40 shader cores at 800MHz, memory at 950MHz, WHQL Catalyst Driver v8.85.7.1)

The main program was implemented in C++, compiled and linked with Visual C++ Professional 2010. The graphics API of choice was OpenGL and all shaders were implemented in conformance to the feature-set of the OpenGL Shading Language (GLSL) version 1.20. A diagram depicting all the stages of the program for the display of a complete frame on the screen is presented in Figure A.12.

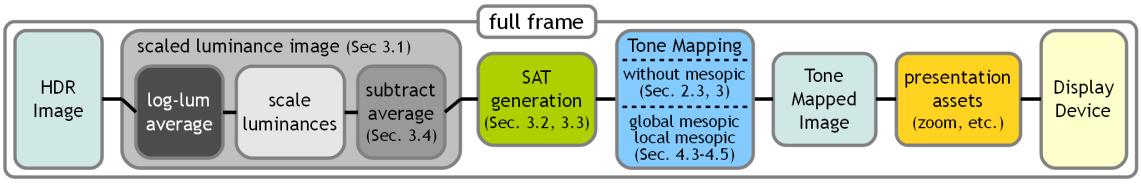


Figure A.12: Overview of all the stages implemented in the demo program for the display of a complete frame.

All performance times in this Section are given in *milliseconds*. Full-frame times were captured with performance counters from the Win32 API and double-checked with the free version of Fraps 3.4.6. Intra-frame performance was profiled using OpenGL Timer Query Objects (`GL_ARB_timer_query`). Performance results were recorded through multiple executions of the program from which outliers were removed and the average was taken.

Performance times for the tone mapping stage alone (i.e., only for the blue-box of Figure A.12) with and without the mesopic simulation are shown in Figure A.13. The relative overhead introduced to the tone mapping operator alone by the spatially-varying mesopic filter is shown in Figure A.15. Complete frame times are shown in Figure A.14, accounting for all the stages depicted in Figure A.12 with the spatially-varying mesopic

filter activated. These times are all explicitly listed in Table A.1, save for time spent on presentation assets and for the SAT generation time which was already inspected in Section 4.7; speaking of SAT generation, the full frame performance was profiled using the balanced tree approach with $k = 4$.

From Figure A.15 one can see that the overhead introduced by the spatially-varying mesopic filter tends to amount to about 16% up to 19% of the original execution time of the operator without the filter (8% for the ATI FirePro 3D V3700). Most of this overhead is coming from the non-linear Yxy to Lab color conversions (and vice-versa) inside the tone mapping shader. Note that this overhead is being measured relative to the tone mapping stage alone; putting it on a full-frame scale the overhead is most likely negligible.

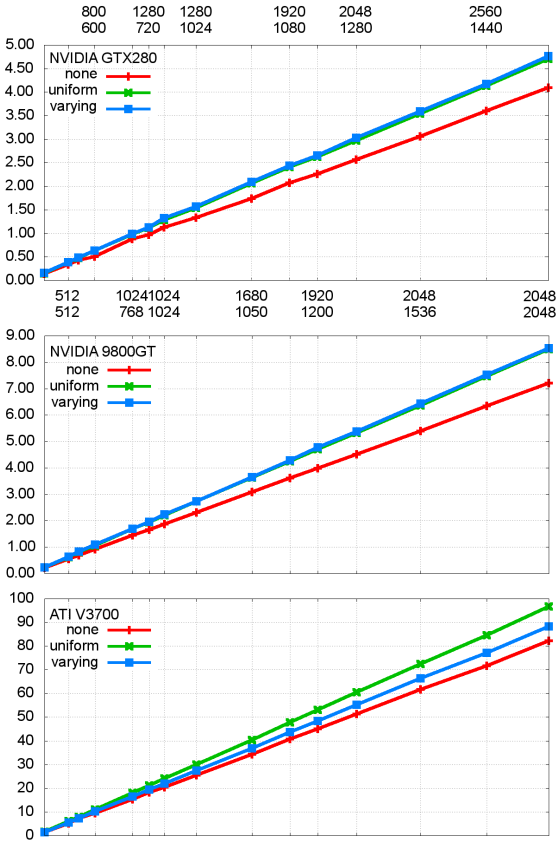


Figure A.13: Performance of the tone mapping stage alone, with and without the mesopic filters.

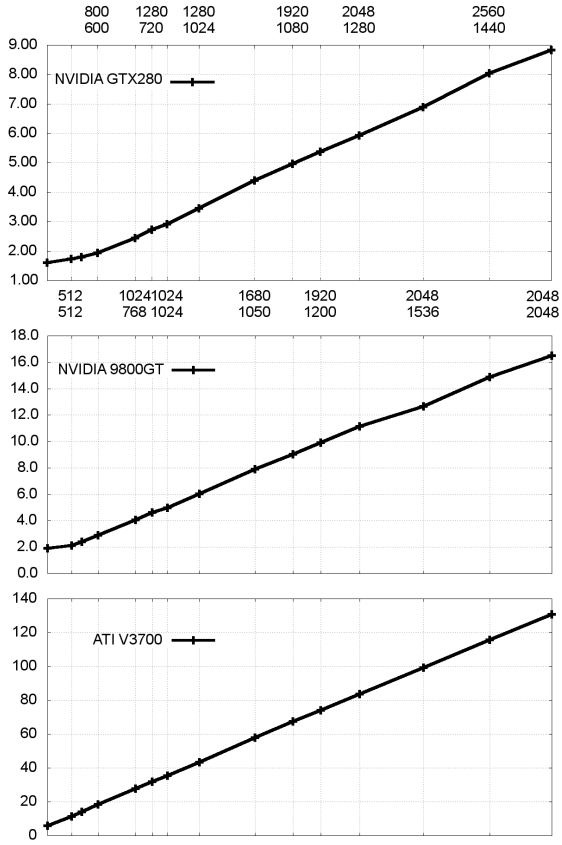


Figure A.14: Performance times accounting for all the stages presented in Figure A.12 required for a full-frame display.

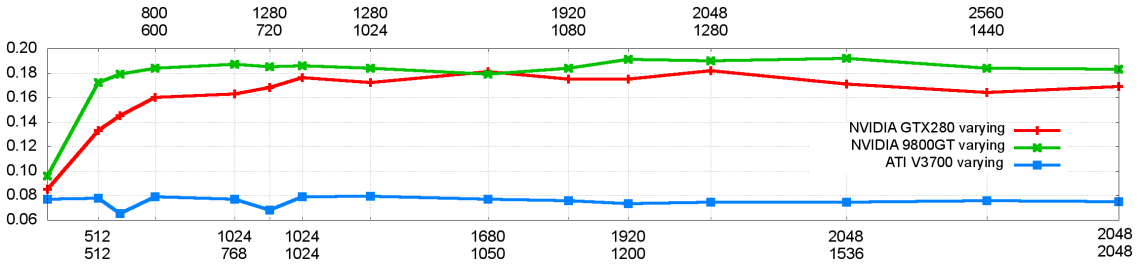


Figure A.15: Overhead of the spatially-varying mesopic filter to the tone mapping stage.

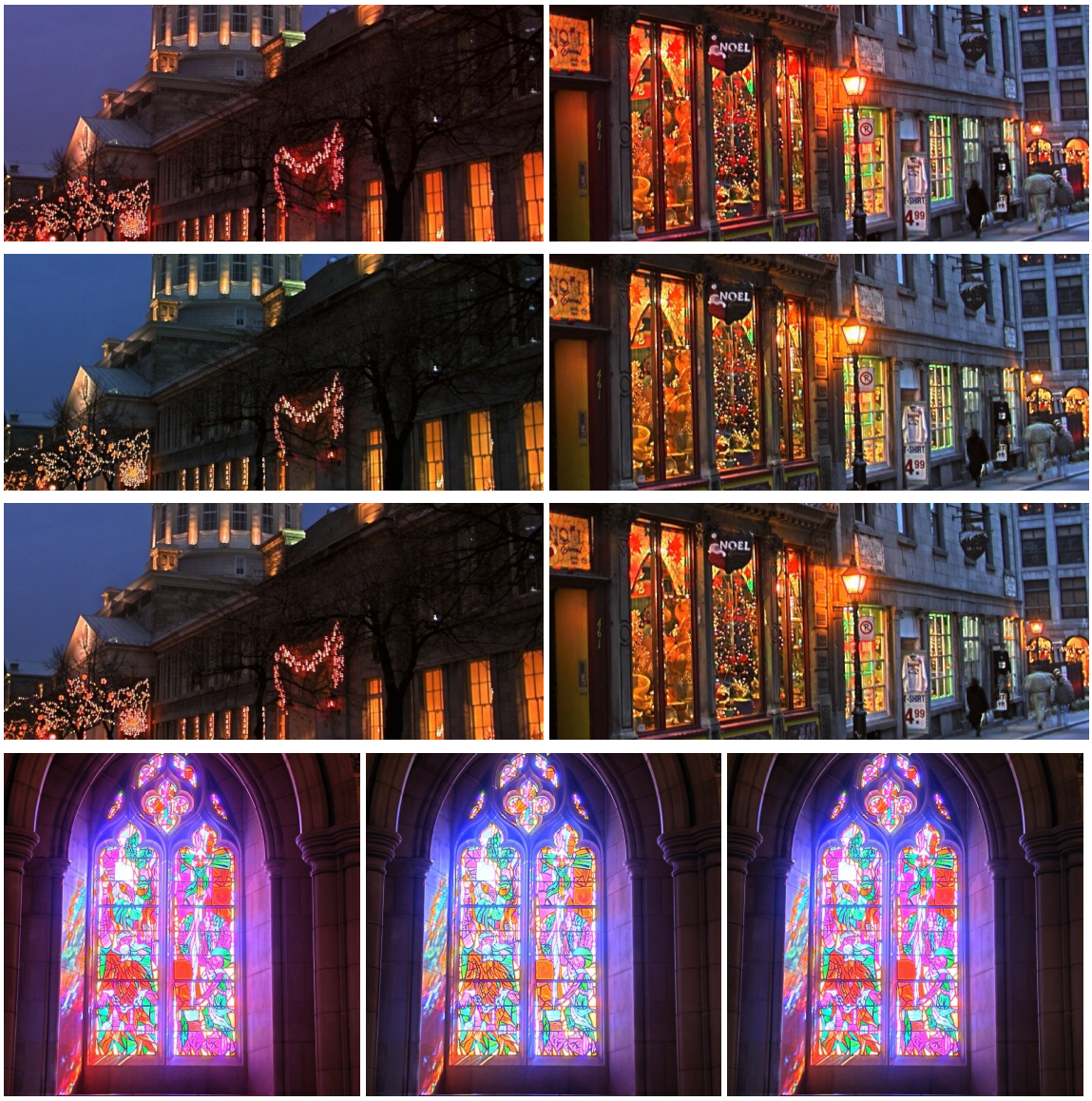


Figure A.16: Examples of the spatially-uniform and spatially-varying mesopic filters. The first row of the top images is without the filter, the second row is the spatially-uniform filter and the third row is the spatially-varying filter; similarly for the images at the bottom, but arranged in columns instead of in rows.

			GTX 280					9800 GT					ATI FirePro 3D V3700				
	width	height	RL	TM.N	TM.MU	TM.MV	FT	RL	TM.N	TM.MU	TM.MV	FT	RL	TM.N	TM.MU	TM.MV	FT
1	256	256	0.345	0.141	0.152	0.153	1.565	0.427	0.227	0.241	0.245	1.921	0.201	1.274	1.496	1.372	5.690
	65536																
2	512	512	0.414	0.337	0.380	0.382	1.715	0.536	0.558	0.634	0.642	2.082	0.739	5.092	6.013	5.487	11.302
	262144																
3	720	480	0.452	0.420	0.477	0.481	1.695	0.594	0.706	0.187	0.825	2.355	1.003	7.230	7.900	7.234	13.978
	345600																
4	800	600	0.482	0.499	0.482	0.493	1.833	0.715	0.926	1.079	1.088	2.956	1.433	9.345	11.015	10.082	18.279
	480000																
5	1024	768	0.555	0.873	0.972	0.985	2.436	0.875	1.444	1.695	1.709	4.036	2.715	15.299	18.013	16.475	27.589
	786432																
6	1280	720	0.633	0.963	1.110	1.125	2.725	0.956	1.644	1.910	1.933	6.093	2.744	19.291	21.077	19.287	31.906
	921600																
7	1024	1024	0.684	1.120	1.279	1.318	2.912	0.977	1.865	4.046	4.077	8.166	2.864	20.360	24.033	21.966	35.222
	1048576																
8	1280	1024	0.715	1.335	1.543	1.565	3.444	1.366	2.321	2.722	2.744	9.844	3.638	25.405	29.951	27.423	43.365
	1310720																
9	1680	1050	0.911	1.738	2.061	2.090	4.395	1.724	3.704	3.622	3.640	12.087	5.036	36.905	40.283	34.255	57.723
	1764000																
10	1920	1080	0.957	2.075	2.409	2.439	4.957	1.722	3.625	4.265	4.295	14.495	6.232	40.633	47.769	43.705	67.299
	2073600																
11	1920	1200	0.987	2.258	2.625	2.655	5.377	2.140	7.626	8.752	8.808	16.197	6.798	45.107	53.095	48.426	74.053
	2304000																
12	2048	1280	1.041	2.560	2.966	3.028	5.921	2.229	4.253	5.325	5.368	17.583	7.513	51.336	60.390	55.175	83.611
	2621440																
13	2048	1536	1.108	3.061	3.543	3.585	6.886	2.223	10.403	11.935	12.015	20.310	8.580	61.705	72.488	66.305	99.227
	3145728																
14	2560	1440	1.329	3.595	4.131	4.169	8.035	3.198	12.246	14.022	14.127	23.756	10.856	71.668	84.561	77.104	115.669
	3686400																
15	2048	2048	1.366	4.088	4.704	4.759	8.830	3.125	13.783	15.983	16.032	26.336	11.471	82.131	96.755	88.255	130.719
	4194304																

Table A.1: Complete list of performance times profiled. The abbreviations in the table are as follows: (RL) relative luminance; (TM.N) tone mapping without mesopic simulation; (TM.MU) tone mapping with the uniform mesopic filter; (TM.MV) tone mapping with the varying mesopic filter; (FT) full frame time.

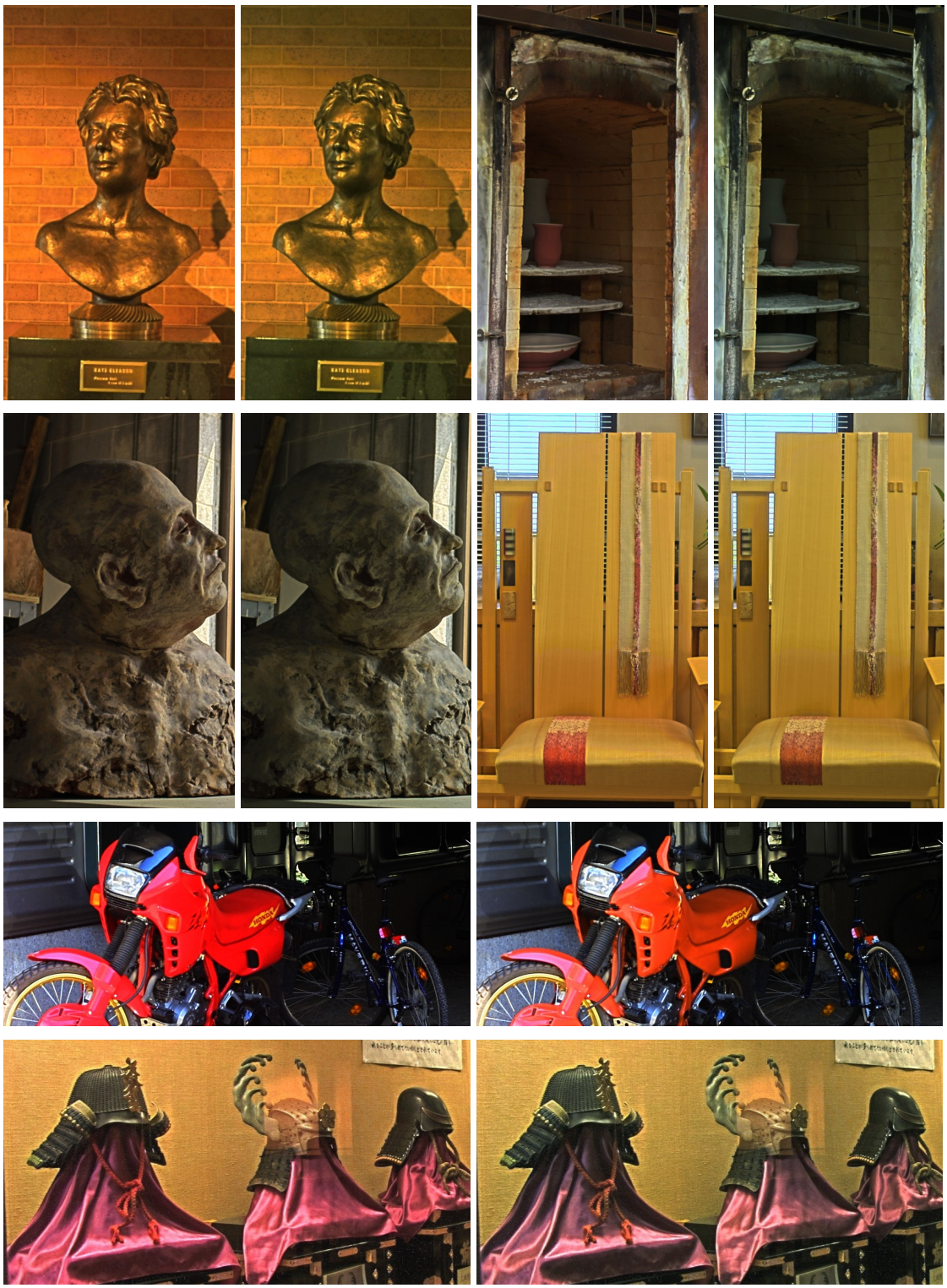


Figure A.17: More examples of the spatially-uniform and spatially-varying mesopic filters. The leftmost image of each image set is without any filter; the rightmost is the spatially-varying filter. For these images, either the global or the local mesopic filters produce very similar results because there are no strong light intensities in the images, which makes the local averages to be somewhat close to the global average.

A.7 Limitations and Future Work

It must be noticed that either the spatially-uniform or spatially-varying mesopic filters decrease responses from red intensities only, as suggested by the curves of equivalent lightness that were studied. A future opportunity is to investigate how the response coming from of other colors (green and yellow, namely, for the assumed opponent-system) behave according to different levels of luminosity, leading to a much more robust and believable mesopic vision experience.

Another limitation of both filters is that color-shifts happen instantly. It is a known fact that chrominance changes in the HVS do not occur abruptly, but instead stabilize gradually on due time, just like with luminosity adaptation. Temporal luminance adaptation is a topic already studied extensively, but little is known about temporal chromatic adaptation, which still remains as a fascinating open field for further research.

The local averages used by the spatially-varying mesopic filter come directly from the brightness perception model used implicitly by the local photographic operator. Such local averages are convenient since they are already part of the luminance compression framework that was utilized and are efficient to be computed. The estimation of such local averages is based on psychophysical brightness perception experiments and the obtained results look plausible; however, they may not be the most suitable candidate for local averages under mesopic conditions. Additional physiological evidence must be researched in order to validate the accuracy of the employed brightness perception model for mesopic vision.

An implicit assumption made for both filters was that the source HDR imagery was properly calibrated. In other words, the HDR images were expected to be holding physically accurate quantities. This may not be always the case, but HDR images should supposedly contain proportionally equivalent quantities at least, differing to real quantities only by some constant, uniform scaling factor. In the context of the proposed mesopic reproduction filters, any HDR image that infringes this constraint is considered an ill-formed image.

Finally, adapting other local operators to function along with the proposed spatially-varying mesopic filter is a possible direction for future work. However, not all local operators are guided by perceptual characteristics of the Human Visual System and thus may be less suitable for a proper mesopic vision reproduction experience.

A.8 Conclusion

A novel, general and efficient spatially-varying approach to reproduce the Purkinje effect under mesopic vision conditions was introduced. The method was designed to work with popular, widely-spread HDR imagery formats. The blue-shift is simulated by suppressing the responses coming from red intensities according to local luminosity conditions. These red responses are smoothed based on an equivalent-lightness curve recovered through psychophysical experiments performed on real subjects. The smoothing is applied in an HVS-compatible, perceptually-linear fashion through the $CIE\ L^*a^*b^*$ opponent-color space, and this linearity conforms to recent physiological evidence. The proposed spatially-varying filter exploits the foundations and perceptually-based characteristics of the local photographic operator to perform mesopic adjustments in a per-pixel basis. The filter is simple to be implemented entirely on the GPU and the overhead introduced is negligible and should not hurt the performance of existing real-time applications.

A.9 Related Publications

SLOMP, M.; MIKAMO, M.; KANEDA, K. Fast Local Tone Mapping, Summed-Area Tables and Mesopic Vision Simulation. In: MUKAI, N. (Ed.). **Computer Graphics**. [S.l.]: InTech, 2012. p.AAA–BBB

MIKAMO, M. et al. A tone reproduction operator accounting for mesopic vision. In: SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 POSTERS, New York, NY, USA. **Proceedings...** [S.l.: s.n.], 2009. p.41:1–41:1

鴨道弘, Marcos Slomp, 玉木徹, 金田和文: 「薄明視における視覚特性を考慮したトーンリプロダクション手法」, 画像ラボ, vol.22, no.6, pp.52-58 (2011 06).

三鴨道弘, Marcos Slomp, 玉木徹, 金田和文: 「薄明視における視覚特性を考慮したトーンリプロダクション」映像情報メディア学会誌, Vol.64, No.9, pp.1372-1378 (2010).

三鴨道弘, 島田洋輔, Marcos Slomp, 玉木徹, 金田和文: 「薄明視における人間の視覚特性を考慮したトーンリプロダクション手法」, 画像電子学会第243回研究会講演予稿, 08-05-27, pp.155-161, 鹿児島大学, 鹿児島(2009 03).

REFERENCES

ADAMS, A. **The Print**. [S.l.]: Little, Brown and Company, 1983.

BATTIATO, S. et al. Digital Mosaic Frameworks - An Overview. **Eurographics - Computer Graphic Forum**, [S.l.], v.26, n.4, p.794–812, 2007.

BAVOIL, L.; SAINZ, M.; DIMITROV, R. Image-space horizon-based ambient occlusion. In: SIGGRAPH '08: ACM SIGGRAPH 2008 TALKS, New York, NY, USA. **Proceedings...** ACM, 2008. p.1–1.

BEARD, K.; CHUANG, C. A New Model for the Equilibrium Shape of Raindrops. **J. Atmos. Sci.**, [S.l.], v.44, p.1509–1524, 1987.

BLASI, G. D.; PETRALIA, M. Fast Photomosaic. In: IN POSTER PROCEEDINGS OF ACM/WSCG2005, 2005. **Proceedings...** [S.l.: s.n.], 2005.

BLELLOCH, G. E. **Prefix Sums and Their Applications**. [S.l.]: Synthesis of Parallel Algorithms, 1990.

BLOMMAERT, F. J.; MARTENS, J.-B. An object-oriented model for brightness perception. **Spatial Vision**, [S.l.], v.5, n.1, p.15–41, 1990.

BRAUWERS, M.; OLIVEIRA, M. M. Real-time refraction through deformable objects. In: I3D '07: PROCEEDINGS OF THE 2007 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, New York, NY, USA. **Proceedings...** ACM, 2007. p.89–96.

BUNNELL, M. Dynamic Ambient Occlusion and Indirect Lighting. In: **GPU Gems 2**: programming techniques for high-performance graphics and general-purpose computation. [S.l.]: Addison-Wesley Professional, 2005. p.223–233.

CAO, D. et al. Rod contributions to color perception : linear with rod contrast. **Vision Research**, [S.l.], v.48, n.26, p.2586–2592, May 2008.

CARR, N. A. et al. Fast GPU ray tracing of dynamic meshes using geometry images. In: GI '06: PROCEEDINGS OF GRAPHICS INTERFACE 2006, Toronto, Ont., Canada, Canada. **Proceedings...** Canadian Information Processing Society, 2006. p.203–209.

CHANGBO, W. et al. Real-time modeling and rendering of raining scenes. **Vis. Comput.**, Secaucus, NJ, USA, v.24, n.7, p.605–616, 2008.

CHRISTENSEN, P. **Ambient occlusion, image-based illumination and global illumination**. 2002.

CROW, F. C. Summed-area tables for texture mapping. **SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques**, New York, NY, USA, v.18, p.207–212, January 1984.

DEBEVEC, P. Image-Based Lighting. **IEEE Computer Graphics and Applications**, Los Alamitos, CA, USA, v.22, p.26–34, 2002.

DEERING, M. et al. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In: SIGGRAPH '88: PROCEEDINGS OF THE 15TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, New York, NY, USA. **Proceedings...** ACM, 1988. p.21–30.

PHARR, M.; FERNANDO, R. (Ed.). **Depth of Field**: a survey of techniques. [S.l.]: Addison-Wesley Professional, 2005. (GPU Gems 2).

DEVLIN, K. **A Review of Tone Reproduction Techniques**. [S.l.]: University of Bristol, 2002.

DÍAZ, J. et al. Real-time ambient occlusion and halos with Summed Area Tables. **Computers & Graphics**, [S.l.], v.34, n.4, p.337 – 350, 2010.

DUBOIS, P.; RODRIGUE, G. An analysis of the recursive doubling algorithm. **High Speed Computer and Algorithm Organization**, New York, NY, USA, p.299–307, 1977.

DURAND, F.; DORSEY, J. Interactive Tone Mapping. In: EUROGRAPHICS WORKSHOP ON RENDERING TECHNIQUES 2000, London, UK. **Proceedings...** Springer-Verlag, 2000. p.219–230.

FERWERDA, J. A. et al. A model of visual adaptation for realistic image synthesis. In: SIGGRAPH 96, New York, NY, USA. **Proceedings...** ACM, 1996. p.249–258.

FILION, D.; MCNAUGHTON, R. StarCraft 2: effects & techniques. In: ACM SIGGRAPH 2008: PROCEEDINGS OF THE CONFERENCE COURSE NOTES, ADVANCES IN REAL-TIME RENDERING IN 3D GRAPHICS AND GAMES. **Proceedings...** [S.l.: s.n.], 2008. p.133–164.

FILION, D.; MCNAUGHTON, R. StarCraft 2: effects & techniques. In: SIGGRAPH '08: ACM SIGGRAPH 2008 CLASSES, New York, NY, USA. **Proceedings...** ACM, 2008. p.133–164.

FOX, M.; COMPTON, S. Ambient Occlusive Crease Shading. **Game Developer Magazine**, [S.l.], 2008.

GARG, K.; NAYAR, S. **Photometric Model of a Rain Drop**. [S.l.: s.n.], 2004.

GARG, K.; NAYAR, S. Detection and Removal of Rain from Videos. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR). **Proceedings...** [S.l.: s.n.], 2004. v.I, p.528–535.

GARG, K.; NAYAR, S. K. Photorealistic rendering of rain streaks. In: SIGGRAPH '06: ACM SIGGRAPH 2006 PAPERS, New York, NY, USA. **Proceedings...** ACM, 2006. p.996–1002.

GOLD, S. et al. New Algorithms for 2D and 3D Point Matching: pose estimation and correspondence. **Pattern Recognition**, [S.l.], v.31, p.957–964, 1997.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

GOODNIGHT, N. et al. Interactive time-dependent tone mapping using programmable graphics hardware. In: EUROGRAPHICS WORKSHOP ON RENDERING, 14., Aire-la-Ville, Switzerland, Switzerland. **Proceedings...** Eurographics Association, 2003. p.26–37. (EGRW '03).

GOODNIGHT, N. et al. Interactive time-dependent tone mapping using programmable graphics hardware. In: SIGGRAPH '05: ACM SIGGRAPH 2005 COURSES, New York, NY, USA. **Proceedings...** ACM, 2005. p.180.

GORAL, C. M. et al. Modelling the Interaction of Light between Diffuse Surfaces. **Computer Graphics**, [S.l.], p.212–222, 1984.

GPGPU Forum. **FBO and point sprite**. `http://www.gpgpu.org/phpBB2/viewtopic.php?t=2332`, Website.

HARRIS, M. **Parallel Prefix Sum (Scan) with CUDA**. [S.l.]: NVIDIA Corporation, 2007.

HARRIS, M.; SENGUPTA, S.; OWENS, J. D. Parallel Prefix Sum (Scan) with CUDA. In: NGUYEN, H. (Ed.). **GPU Gems 3**. [S.l.]: Addison-Wesley Professional, 2007. p.851–876.

HECKBERT, P. S. Filtering by repeated integration. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.20, p.315–321, August 1986.

HECKBERT, P. S. Filtering by repeated integration. In: SIGGRAPH 86, New York, NY, USA. **Proceedings...** ACM, 1986. p.315–321.

HEGEMAN, K. et al. Approximate ambient occlusion for trees. In: I3D '06: PROCEEDINGS OF THE 2006 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, New York, NY, USA. **Proceedings...** ACM, 2006. p.87–92.

HENSLEY, J. et al. Fast summed-area table generation and its applications. **Computer Graphics Forum**, [S.l.], v.24, p.547–555, 2005.

HOFFMANN, G. **CIE Color Space**. [S.l.]: University of Applied Sciences, Emden, 2000.

HULST, H. C. van de. **Light Scattering by Small Particles**. [S.l.]: Wiley & Sons, 1957.

HURVICH, L.; JAMESON, D. Some quantitative aspects of an opponent-colors theory. II. Brightness, saturation, and hue in normal and dichromatic vision. **Journal of the Optical Society of America**, [S.l.], v.45, p.602–616, 1955.

IKEDA, M.; ASHIZAWA, S. Equivalent lightness of colored objects of equal munsell chroma and of equal munsell value at various illuminances. **Color Reserch and Application**, [S.l.], v.16, p.72–80, 1991.

JENSEN, H. W. Global Illumination using Photon Maps. In: RENDERING TECHNIQUES '96. **Proceedings...** Springer-Verlag, 1996. p.21–30.

KAJIYA, J. T. The rendering equation. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.20, n.4, p.143–150, 1986.

KANAMORI, S. et al. Physically Based Rendering of Rainbows Under Various Atmospheric Conditions. In: THE 18TH PACIFIC CONFERENCE ON COMPUTER GRAPHICS AND APPLICATIONS (PACIFIC GRAPHICS): POSTER SECTION, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010.

KANEDA, K.; IKEDA, S.; YAMASHITA, H. Animation of water droplets moving down a surface. **Journal of Visualization and Computer Animation**, [S.l.], v.10, n.1, p.15–26, 1999.

KANEDA, K.; KAGAWA, T.; YAMASHITA, H. Animation of water droplet flow on curved surfaces. **In Proc. Computer Animation**, [S.l.], p.177–189, 1993.

KHAN, S. M.; PATTANAIK, S. N. Modeling Blue shift in Moonlit Scenes by Rod Cone Interaction. **Journal of VISION**, [S.l.], v.4, n.8, p.316a, 2004.

KIPFER, P.; SEGAL, M.; WESTERMANN, R. UberFlow: a gpu-based particle engine. In: HWWS '04: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, New York, NY, USA. **Proceedings...** ACM, 2004. p.115–122.

KIRK, A. G.; ARIKAN, O. Real-time ambient occlusion for dynamic character skins. In: I3D '07: PROCEEDINGS OF THE 2007 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, New York, NY, USA. **Proceedings...** ACM, 2007. p.47–52.

KIRK, A. G.; O'BRIEN, J. F. Perceptually based tone mapping for low-light conditions. In: ACM SIGGRAPH 2011 PAPERS, New York, NY, USA. **Proceedings...** ACM, 2011. p.42:1–42:10. (SIGGRAPH '11).

KIRKPATRICK, S.; GELATT., C. D.; VECCHI, M. P. Optimization by simulated annealing. **Science**, [S.l.], v.220, n.4598, p.671–680, 1983.

KLEIN, A. W. et al. Video Mosaics. In: NPAR 2002: SECOND INTERNATIONAL SYMPOSIUM ON NON PHOTOREALISTIC RENDERING. **Proceedings...** [S.l.: s.n.], 2002. p.21–28.

KOLB, A.; LATTA, L.; REZK-SALAMA, C. Hardware-based simulation and collision detection for large particle systems. In: HWWS '04: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, New York, NY, USA. **Proceedings...** ACM, 2004. p.123–131.

KONTKANEN, J.; AILA, T. Ambient Occlusion for Animated Characters. In: RENDERING TECHNIQUES 2006 (EUROGRAPHICS SYMPOSIUM ON RENDERING). **Proceedings...** [S.l.: s.n.], 2006.

KONTKANEN, J.; LAINE, S. Ambient Occlusion Fields. In: ACM SIGGRAPH 2005 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES. **Proceedings...** ACM Press, 2005. p.41–48.

KRAWCZYK, G.; MYSZKOWSKI, K.; SEIDEL, H.-P. Perceptual effects in real-time tone mapping. In: COMPUTER GRAPHICS (SCCG) 2005, 21., New York, NY, USA. **Proceedings...** [S.l.: s.n.], 2005. p.195–202.

LANDIS, H. Production-Ready Global Illumination. In: SIGGRAPH 2002, COURSE NOTES. **Proceedings...** [S.l.: s.n.], 2002.

LAURITZEN, A. Summed-Area Variance Shadow Maps. In: **GPU Gems 3**. [S.l.]: Addison-Wesley Professional, 2007. p.157–182.

Leadwerks Team. **Leadwerks Engine**.

LEDDA, P. et al. Evaluation of tone mapping operators using a High Dynamic Range display. **ACM Trans. Graph.**, New York, NY, USA, v.24, p.640–648, July 2005.

LEDDA, P.; SANTOS, L. P.; CHALMERS, A. A local model of eye adaptation for high dynamic range images. In: COMPUTER GRAPHICS, VIRTUAL REALITY, VISUALISATION AND INTERACTION IN AFRICA, 3., New York, NY, USA. **Proceedings...** ACM, 2004. p.151–160. (AFRIGRAPH '04).

LEE, S.; EISEMANN, E.; SEIDEL, H.-P. Depth-of-field rendering with multiview synthesis. **ACM Trans. Graph.**, New York, NY, USA, v.28, p.134:1–134:6, December 2009.

LEE, S.; KIM, G. J.; CHOI, S. Real-Time Depth-of-Field Rendering Using Point Splatting on Per-Pixel Layers. **Comput. Graph. Forum**, [S.l.], v.27, n.7, p.1955–1962, 2008.

LUFT, T.; COLDITZ, C.; DEUSSEN, O. Image enhancement by unsharp masking the depth buffer. **ACM Trans. Graph.**, New York, NY, USA, v.25, n.3, p.1206–1213, 2006.

MALMER, M. et al. Fast Precomputed Ambient Occlusion for Proximity Shadows. **Journal of Graphics Tools**, [S.l.], v.12, n.2, p.59–71, 2007.

MARSHALL, J.; PALMER, W. The distribution of raindrops with size. **Journal of Meteorology**, [S.l.], v.5, p.165–166, 1948.

MAX, N. L. Horizon mapping: shadows for bump-mapped surfaces. **The Visual Computer**, [S.l.], v.4, n.2, p.109–117, July 1988.

MEIER, B. J. Painterly rendering for animation. In: SIGGRAPH '96: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, New York, NY, USA. **Proceedings...** ACM, 1996. p.477–484.

MIKAMO, M. et al. A tone reproduction operator accounting for mesopic vision. In: SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 POSTERS, New York, NY, USA. **Proceedings...** [S.l.: s.n.], 2009. p.41:1–41:1.

MIKAMO, M. et al. Maximizing Image Utilization in Photomosaics. In: FIRST INTERNATIONAL CONFERENCE ON NATURAL COMPUTATION (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.275–278.

MILLER, G. Efficient algorithms for local and global accessibility shading. In: SIGGRAPH '94: PROCEEDINGS OF THE 21ST ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, New York, NY, USA. **Proceedings...** ACM, 1994. p.319–326.

MINNAERT, M. **The Nature of Light and Colour in the Open Air**. [S.l.]: Dover Publications, 1954.

MITTRING, M. Finding next gen: cryengine 2. In: SIGGRAPH '07: ACM SIGGRAPH 2007 COURSES, New York, NY, USA. **Proceedings...** ACM, 2007. p.97–121.

NVIDIA Corporation. **NVIDIA Direct3D SDK 10 Code Samples**. 2009.

Ogre Team. **Screen Space Ambient Occlusion in Ogre**.

PATTANAIK, S.; YEE, H. Adaptive gain control for high dynamic range image display. In: COMPUTER GRAPHICS, 18., New York, NY, USA. **Proceedings...** ACM, 2002. p.83–87. (SCCG '02).

PHARR, M.; GREEN, S. Ambient Occlusion. In: **GPU Gems**: programming techniques, tips and tricks for real-time graphics. [S.l.]: Addison-Wesley Professional, 2004. p.223–233.

PHONG, B. T. Illumination for computer generated pictures. **Commun. ACM**, New York, NY, USA, v.18, n.6, p.311–317, 1975.

POLICARPO, F.; FONSECA, F. **Deferred shading tutorial**. 2004.

PURCELL, T. J. et al. Ray tracing on programmable graphics hardware. In: SIGGRAPH '05: ACM SIGGRAPH 2005 COURSES, New York, NY, USA. **Proceedings...** ACM, 2005. p.268.

REINHARD, E. Parameter Estimation for Photographic Tone Reproduction. **Journal of Graphics Tools**, [S.l.], v.7, n.1, p.45–52, 2003.

REINHARD, E. et al. Photographic tone reproduction for digital images. **ACM Transactions on Graphics**, New York, NY, USA, v.21, p.267–276, July 2002.

REINHARD, E. et al. **High Dynamic Range Imaging**: acquisition, display, and image-based lighting (2nd edition). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

RITSCHEL, T.; GROSCH, T.; SEIDEL, H.-P. Approximating Dynamic Global Illumination in Image Space. In: ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES (I3D) 2009. **Proceedings...** [S.l.: s.n.], 2009.

ROUSSEAU, P.; JOLIVET, V.; GHAZANFARPOUR, D. Realistic real-time rain rendering. **Computers & Graphics**, [S.l.], v.30, n.4, p.507–518, aug 2006. special issue on Natural Phenomena Simulation.

SENGUPTA, S. et al. Scan Primitives for GPU Computing. In: GRAPHICS HARDWARE 2007. **Proceedings...** Association for Computing Machinery, 2007. p.97–106.

SENGUPTA, S.; LEFOHN, A.; OWENS, J. D. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In: WORKSHOP ON EDGE COMPUTING USING NEW COMMODITY ARCHITECTURES, 1., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.D–26–27.

SHANMUGAM, P.; ARIKAN, O. Hardware accelerated ambient occlusion techniques on GPUs. In: I3D '07: PROCEEDINGS OF THE 2007 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, New York, NY, USA. **Proceedings...** ACM, 2007. p.73–80.

SILVERS, R. **Photomosaics**. New York, NY, USA: Henry Holt and Co., Inc., 1997.

SINKHORN, R. A relationship between arbitrary positive matrices and doubly stochastic matrices. **The Annals of Mathematical Statistics**, [S.l.], v.35, p.876?879, 1964.

SLOAN, P.-P. et al. Image-Based Proxy Accumulation for Real-Time Soft Global Illumination. In: PG '07: PROCEEDINGS OF THE 15TH PACIFIC CONFERENCE ON COMPUTER GRAPHICS AND APPLICATIONS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.97–105.

SLOMP, M. et al. Photorealistic Real-time Rendering of Spherical Raindrops with Hierarchical Reflective and Refractive Maps. In: ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES (I3D), New York, NY, USA. **Proceedings...** ACM, 2010.

SLOMP, M. et al. Photorealistic real-time rendering of spherical raindrops with hierarchical reflective and refractive maps. **Journal of Computer Animation and Virtual Worlds (CAVW)**, [S.l.], v.22, p.393–404, August 2011.

SLOMP, M. et al. GPU-based SoftAssign for Maximizing Image Utilization in Photomosaics. **International Journal of Networking and Computing (IJNC)**, Los Alamitos, CA, USA, v.1, p.211–229, July 2011.

SLOMP, M.; MIKAMO, M.; KANEDA, K. Fast Local Tone Mapping, Summed-Area Tables and Mesopic Vision Simulation. In: MUKAI, N. (Ed.). **Computer Graphics**. [S.l.]: InTech, 2012. p.AAA–BBB.

SLOMP, M.; OLIVEIRA, M. M. Real-Time Photographic Local Tone Reproduction Using Summed-Area Tables. In: COMPUTER GRAPHICS INTERNATIONAL 2008, Istambul, Turkey. **Proceedings...** [S.l.: s.n.], 2008. p.82–91.

SLOMP, M.; TAMAKI, T.; KANEDA, K. Screen-Space Ambient Occlusion through Summed-Area Tables. In: FIRST INTERNATIONAL CONFERENCE ON NETWORKING AND COMPUTING (ICNC), 2010., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010. p.1–8. (ICNC '10).

STARIK, S.; WERMAN, M. Simulation of rain in video. In: IN PROC. 3RD INT'L WORKSHOP ON TEXTURE ANALYSIS AND SYNTHESIS. **Proceedings...** [S.l.: s.n.], 2002. p.95–100.

TABELLION, E.; LAMORLETTE, A. An approximate global illumination system for computer generated films. **ACM Trans. Graph.**, New York, NY, USA, v.23, n.3, p.469–476, 2004.

TAMAKI, T. et al. SoftAssign and EM-ICP on GPU. **International Conference on Natural Computation**, Los Alamitos, CA, USA, v.0, p.179–183, 2010.

TAMAKI, T. et al. CUDA-based implementations of Softassign and EM-ICP. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR) DEMOS, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2010.

TATARCHUK, N. Artist-directable real-time rain rendering in city environments. In: SIGGRAPH '06: ACM SIGGRAPH 2006 COURSES, New York, NY, USA. **Proceedings...** ACM, 2006. p.23–64.

Tom's Hardware. **GPU Performance chart**. `http://www.tomshardware.com/charts/gaming-graphics-charts-q3-2008/compare,793.html?prod[2111]=on&prod[2108]=on`, Website.

VIOLA, P.; JONES, M. Robust Real-time Object Detection. **International Journal of Computer Vision**, [S.l.], v.57, n.2, p.137–154, 2004.

WANG, H.; MUCHA, P. J.; TURK, G. Water drops on surfaces. **ACM Trans. Graph.**, New York, NY, USA, v.24, n.3, p.921–929, 2005.

WANG, L. et al. Real-time rendering of realistic rain. In: SIGGRAPH '06: ACM SIGGRAPH 2006 SKETCHES, New York, NY, USA. **Proceedings...** ACM, 2006. p.156.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, New York, NY, USA, v.23, n.6, p.343–349, 1980.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, New York, NY, USA, v.23, n.6, p.343–349, 1980.

WYMAN, C. An approximate image-space approach for interactive refraction. **ACM Trans. Graph.**, New York, NY, USA, v.24, n.3, p.1050–1053, 2005.

ZHANG, X.; WANDELL, B. A spatial extension of CIELAB for digital color image reproduction. **Journal of the Society for Information Display (SID)**, [S.l.], v.5, p.61–63, 1997.

ZHAO, X.; YANG, X. Real-time horizon-based reflection occlusion. In: SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 SKETCHES, New York, NY, USA. **Proceedings...** ACM, 2009. p.1–1.

ZHUKOV, S.; IONES, A.; KRONING, G. An Ambient Light Illumination Model. In: EUROGRAPHICS RENDERING WORKSHOP 1998. **Proceedings...** [S.l.: s.n.], 1998. p.45–56.