FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Comparison of Image Blurring Techniques on Modern Graphics Processing Hardware

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Markus Ernst
Matrikelnummer 0625316

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: David Schedl, MSc.

Wien, 13.03.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Comparison of Image Blurring Techniques on Modern Graphics Processing Hardware

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Markus Ernst
Registration Number 0625316

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: David Schedl, MSc.

Vienna, 13.03.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Markus Ernst
Gerbergasse 7/12/1, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Markus Ernst)

Acknowledgements

First, I want to thank my supervisor David Schedl for supporting during the work on this thesis by answering whatever question asked or giving me feedback.

I also want to thank Michael Wimmer for his valuable feedback as supervisor of this project.

Finally, i want to thank all friends and colleagues who spend their time helping proof-reading this thesis.

Abstract

The increasingly computational power and programmability of modern graphics hardware provides developers of real-time rendering applications with the resources needed to realize more and more complex graphical effects. Some of these effects, like Depth-of-Field, require an efficient image blurring technique to achieve real-time frame rates of 30 frames per second or above. This work presents a comparison of various blurring techniques in terms of their performance on modern graphics hardware. Whereas most of the chosen methods are exclusively used to blur an image, some of them are capable of applying an arbitrary filter. Furthermore, the quality of the different methods has been determined using an automatic process which utilizes a calibrated visual metric. Another aspect when using modern graphics hardware is the increasing scope of operations, especially in the domain of image processing, that can be carried out by using general-purpose computing on graphics processing units (GPGPU). In the recent years, utilizing GPGPU has become increasingly popular inside real-time rendering applications for special tasks like physics simulations. Therefore, all chosen algorithms have been implemented using shaders (GLSL) and GPGPU (CUDA), to answer the question whether or not the usage of a general purpose computing language is applicable for image blurring in real-time rendering and how it compares to using a shading language.

Kurzfassung

Die steigende Rechenleistung und Programmierbarkeit von moderner Grafikhardware erlaubt es den Entwicklern von Echtzeitgrafikanwendungen immer komplexere Grafikeffekte umzusetzen. Manche dieser Effekte, wie Depth-of-Field, benötigen einen effizienten Blur-Filter um eine Bildwiederholrate von 30 Bildern pro Sekunde oder mehr zu erreichen. Diese Arbeite präsentiert einen Vergleich verschiedener Blurring-Techniken bezüglicher ihrer Performance auf moderner Grafikhardware. Während die meisten der gewählten Methoden ausschließlich in der Lage sind ein Bild weichzuzeichnen, bieten manche die Möglichkeit einen beliebigen Filter anzuwenden. Außerdem wurde die Qualität der verschiedenen Methoden mithilfe eines automatischen Prozesses bestimmt, welcher eine kalibrierte visuelle Metrik verwendet. Ein weiterer Aspekt bei der Verwendung von moderner Grafikhardware ist die immer grösser werdende Anzahl von Operationen, besonders im Bereich der Bildverarbeitung, welche mithilfe von general-purpose-computing on graphics processing units (GPGPU) realisiert werden können. In den letzten Jahren wurde das Verwenden von GPGPU für Spezialaufgaben wie Physiksimulation in Echtzeitgrafikanwendungen immer beliebter. Deswegen wurden alle ausgesuchten Algorithmen sowohl mit einer Shader-Sprache (GLSL) als auch GPGPU (CUDA) realisiert, um diese beiden Möglichkeiten hinsichtlich Echtzeitgrafik zu vergleichen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The programmability of modern graphics hardware and its steadily increasing computational power makes the implementation of highly complex image processing techniques in real-time rendering viable. Especially the process of filtering an image with a specific filter kernel (e.g. Gaussian) is of interest for a lot of post-processing effects. Amongst different filter kernels, the Gaussian kernel, which is used to blur or smooth an image, is one of the most used filters in real-time rendering. This work presents a comparison of various blurring techniques in terms of their performance on modern graphics hardware. Whereas most of the chosen methods are exclusively used to blur an image, some of them are capable of applying an arbitrary filter. A prominent example of a post-processing effect that originates from the physical nature of lenses is Depth-of-Field (DoF). In computer graphics we use the model of an idealized pinhole camera with an infinitely small aperture. When utilizing such a camera model, all points independent of their distance to the camera appear in focus. Real world cameras have a finite-size aperture and therefore only points near the focal plane appear in focus whereas other points appear blurred in the image. Figure 1.2 shows the result of a DoF implementation with a pinhole camera image on the left and the scene with simulated DoF on the right. The effect is used to highlight certain regions in an image and it also provides important depth cues. To achieve a high quality DoF simulation, methods like proposed in [15] need to filter a full-screen image several times per frame with filter kernels of different width. Therefore, the image filtering has to be done in an efficient manner to achieve real-time frame rates of more than 30 frames per second. Other post-processing effects that employ image blurring in one way or the other include tone-mapping or glow [7]. The latter can be seen in Figure 1.1.

In image processing, the common way to apply image filtering is convolution, which is a process originating from functional analysis. The data we deal with in image processing in contrast to functional analysis is finite and discrete, where the image as well as the filter kernel are given as two-dimensional matrices containing finite data. When applying convolution, the filter kernel is moved over the entire input image so that the center value of the filter kernel

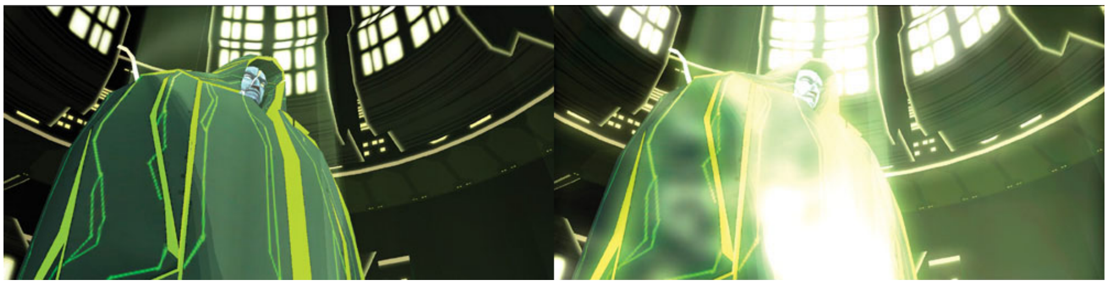


Figure 1.1: Screenshots of a scene with and without glow effect. [7]

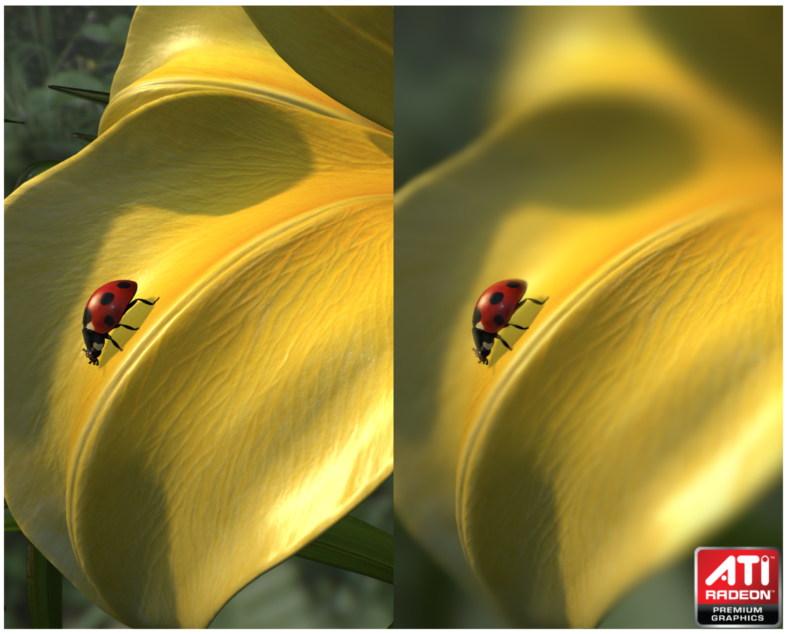


Figure 1.2: Screenshots from the DoF implementation done by Kosloff using Spreading [13].

overlaps every pixel once. At every pixel, the weighted sum of pixels covered by the kernel is computed and written to the output image. For large filter kernels, this requires a large number of lookups at every pixel, which makes the process inefficient for such kernels, therefore different filtering techniques have been proposed for the use in real-time rendering applications. Some of those methods try to achieve a better performance by approximating the convolution with a Gaussian filter kernel, like those which use image pyramids [17], where the input image is first down-sampled N times, where N is the number of desired pyramid levels. Afterwards a chosen level of the pyramid is up-sampled again to obtain a blurred image. Another method of this kind is Spreading [13], which uses a technique that the authors describe as summed-area technique in reverse, where first an image is created by spreading the response function of the used filter kernel and then the output is formed by taking the summed-area table of this image. In contrast, other methods like Reshuffling [24] attempt to enhance performance by

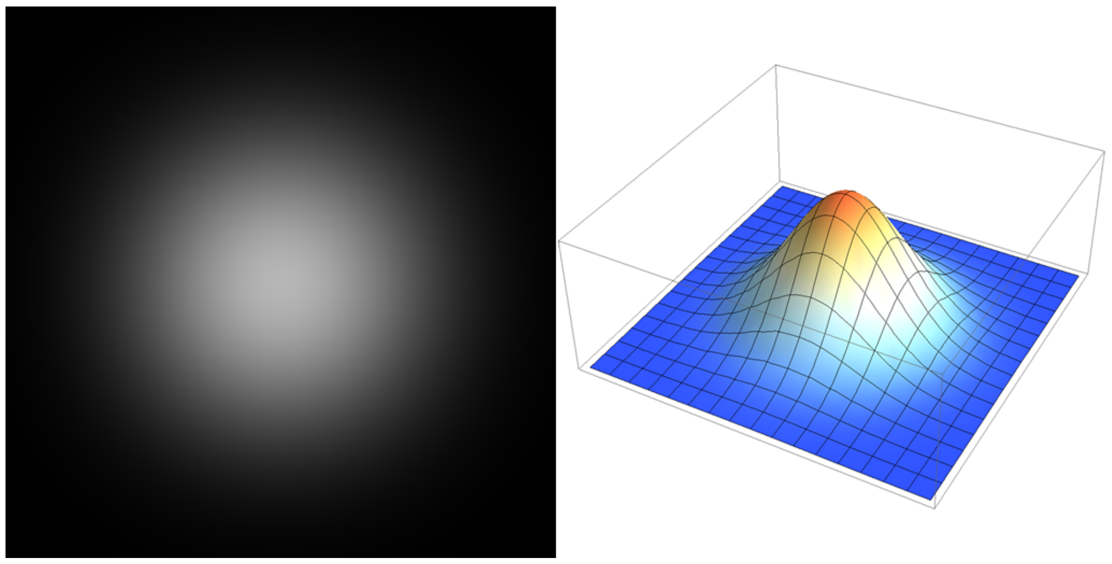


Figure 1.3: Gaussian distribution: on the left a 2D image with the Gaussian distributed illustrated by gray values, on the right a Gaussian distribution represented by a height field in a 3D plot.

preprocessing the filter kernel to reduce the number of operations needed during the convolution process. Some methods transform the image into another domain, for example the frequency domain when using Fast Fourier Transformation (FFT). In case of FFT, both the input image and the filter kernel are transformed into the frequency domain using FFT, where according to the convolution theorem the convolution becomes a pointwise multiplication of the two. This allows to apply a filter kernel of arbitrary size with the same number of operations; in other words the algorithm has a constant time complexity with respect to filter size. Because of the different ways those methods achieve the goal of blurring an image, their relevance in terms of real-time rendering and therefore their performance on modern graphics hardware is of interest. Also due to the fact that some methods are approximations of applying a Gaussian filter kernel via convolution, the quality of these methods compared to the results achieved by convolution is relevant.

Another aspect when using modern graphics hardware is the increasing scope of operations, especially in the domain of image processing, that can be carried out by using a general purpose computing language for graphics processing units (GPUs) like the Compute Unified Device Architecture (CUDA) or the Open Computing Language (OpenCL). These allow the programmer to use the GPU for general purpose computations in a highly parallelized manner by using a non-shader language, e.g. in the case of CUDA a C-like interface. So besides using shaders to implement image filtering using graphics hardware, the algorithms can also be implemented using general-purpose computing on graphics processing units (GPGPU) and its OpenGL interoperability features.

1.2 Problem Statement and Aim of this Work

The main research question of this thesis is how different image filtering techniques on modern graphics hardware perform when used in real-time-rendering applications. Especially techniques that blur an image, like it is achieved by convolution with a Gaussian filter kernel, are of interest. Performance, in this context, not only includes the speed of the methods, but also the image quality for those filtering techniques that produce an approximate result (e.g. pyramid methods [17]). Whether or not the usage of a general purpose computing language is applicable for image filtering in real-time rendering and how it compares to using a shading language are additional questions. They arise from the fact that implementing special tasks like physics simulations using GPGPU inside real-time rendering applications is increasingly popular.

The main task of this work is to implement several image filtering algorithms ranging from sophisticated algorithms like the conventional convolution or Fast Fourier Transformation (FFT) to recently proposed methods like Spreading [13]. The methods listed under "General methods" in the following list are able to apply an arbitrary filter kernel, while the others are used for image blurring only.

- General methods
  - Reshuffling [24]
  - Fast-Fourier-Transform (FTT)
  - Direct convolution
- Pyramid methods
  - Quasi-Convolution [17]
  - Box downsampling (2x2 and 4x4) [16]
- Summed-area table methods
  - Polynomial Spreading [13]
- Iterative methods
  - Repeated box filtering [14]
  - Simulated heat diffusion [11]

For quality comparison, the hdr-vdp2 [19] visual metric is used, which also allows determining the parameters for the Gaussian filter kernel which corresponds to a specific set of parameters when using a method like Spreading [13]. This is done by taking 50 images [1] showing different scenarios (e.g. urban, forest, mountains) and comparing the result for an approximation method for all those images versus the result of the convolution process with a Gaussian filter kernel with different size and sigma to determine the corresponding parameters. This is essential to compare

[1]taken from http://www.flickr.com

the performance of the methods on an equal level by matching the effective filter size when measuring the timings for a performance comparison. To answer the question of the applicability of a general purpose computing language for GPUs in terms of image filtering in real-time rendering, CUDA has been chosen as representative and therefore all methods (except FFT, which is only implemented in CUDA) are implemented in CUDA and in GLSL with OpenGL as graphics API.

1.3 Contributions

The following list contains the contributions of this thesis:

1. Comparison of a broad range of different blurring methods in terms of performance when using different blurring and image sizes and examining different ways of implementing them on modern graphics processing hardware. This is done by implementing the various techniques into a common application to compare the methods in a fair environment with a comparable setup. No such comparison was presented in literature prior to this thesis.

2. Comparison of CUDA versus GLSL regarding image filtering, to determine the relevance of a general purpose computing language for GPUs for real-time rendering. This includes also the examination of the interoperability features provided by CUDA, which make it possible to access OpenGL textures inside a CUDA kernel without downloading the data into CPU main memory, first.

3. Comparing the quality for the methods that achieve image blurring by an approach other than convolution. This is done using an automated process, which utilizes a calibrated visual metric (hdr-vdp2 [19]). The comparison is based on the approximated parameter (sigma) of a Gaussian filter kernel obtained automatically using absolute pixel difference. The results of this approximation also allow for a fair comparison with the determined effective filter size in terms of performance.

4. Creating a guideline for graphics developers who are interested in integrating image filtering (especially image blurring) in their application and showcasing the seemingly most viable method, quasi-convolution [17], in a Depth-of-Field implementation. This includes guidelines for an efficient implementation of image filtering in various ways when using a shading language as well as a GPGPU language.

1.4 Structure

This thesis is structured into 5 chapters, starting with the chapter the current chapter "Introduction".

The following Chapter 2, "Related Work", explains how various real-time rendering post-processing effects work and how image blurring is used in those. Furthermore it presents the used filtering methods in a short overview.

The theory and working principle of the chosen algorithms are explained in detail in Chapter 3, "Filtering Methods". It also contains details about the actual implementation of the methods and for some of those also variations of how they can be implemented. The sections correspond to the chosen methods with its subsections being Theory and Implementation where the Implementation subsection is split into CUDA and GLSL.

In Chapter 4, "Results" first the timings for the named methods considering different filter sizes and input resolutions are presented. Those are presented in plots comparing the CUDA versus the GLSL implementation and the differences are discussed in detail. The chapter continues by showing the results of the quality comparison done by hdr-vdp2 [19] by presenting the determined effective filter sizes of the approximation methods. Based upon those results a discussion of the overall performance comparing the different methods directly versus each other is given.

A short summary of the thesis including a brief review of the results presented in Chapter 4 can be found in Chapter 5, "Conclusion". It also contains the most interesting findings of the comparison as a take-home message.

CHAPTER 2

Related Work

2.1 Summed-Area-Tables

In 1984, Frank Crow presented a method [5] he intended to use for mipmaps in texture mapping. The method builds a data structure called summed-area-table (or integral image), which allows the efficient calculation of the sum of values in a rectangular subarea of an image. In a summed-area-table, the entry at position (x, y) contains the sum of pixels below and to the left of that position including the value at (x, y), which can be expressed as a formula as follows:

$$T(x, y) \equiv \sum_{\substack{x' \leq x \\ y' \leq y}} I(x, y) \tag{2.1}$$

A naive CPU implementation would scan the image iteratively from left to right and bottom to top, while using the following formula to calculate the values in the summed-area-table:

$$T(x, y) \equiv I(x, y) + T(x - 1, y) + T(x, y - 1) - T(x - 1, y - 1) \tag{2.2}$$

Once the summed-area-table is generated, computing the sum of values in an arbitrary rectangular area can be done by using only four lookups. The subsequent formula calculates the sum of values for the area marked with grey in Figure 2.1, with the coordinates for point a being (x_a, x_a) :

$$S = T(x_b, y_b) - T(x_a, y_a) - T(x_d, y_d) + T(x_c, x_c) \tag{2.3}$$

When using a shading language or GPGPU like CUDA, the naive implementation depicted with equation 2.2 is not an option. In case of a shading language like GLSL, it is prior to Version 4.2 not possible to read from and write to the same texture in one pass. With version 4.2 and the introduction of the image type or in CUDA, where it is possible to read from and write to the same texture in a single pass, the non existing synchronization mechanisms circumvent the usage of such an implementation. Therefore, methods were developed to allow the efficient

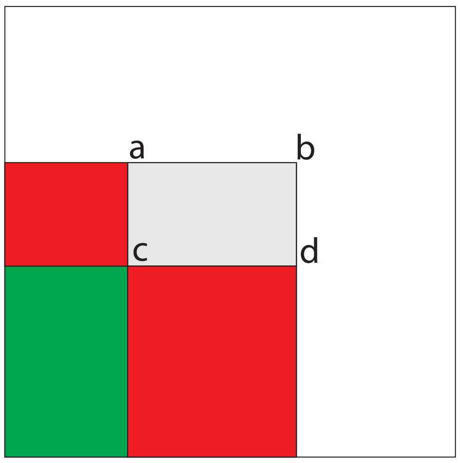


Figure 2.1: Calculating the sum of a rectangular area using a summed-area-table.

implementation of summed-area-tables in parallel, like proposed by Hensley et al. [9]. The algorithm splits the summed-area-table generation into a vertical and a horizontal phase. Furthermore, each phase is split into several passes, where the number of passes (see n and m in Algorithm 2.1) is depending on the parameter r of the algorithm. This parameter controls how many texture lookups are done per pixel per pass. To explain the working principle of the algorithm we examine an example in 1D with the number of texture lookups being two. In the first pass, the value one element to the left is added to the current value. In each successive pass, the stride is doubled when using two texture lookups per pixel, yielding the final result after $log_2(width)$ passes. In general, the stride for a given pass is calculated by r^i and the number of passes is given by $log_r(width)$. An illustration of the explained working principle can be seen in Figure 2.2 and the full algorithm is given in Algorithm 2.1.

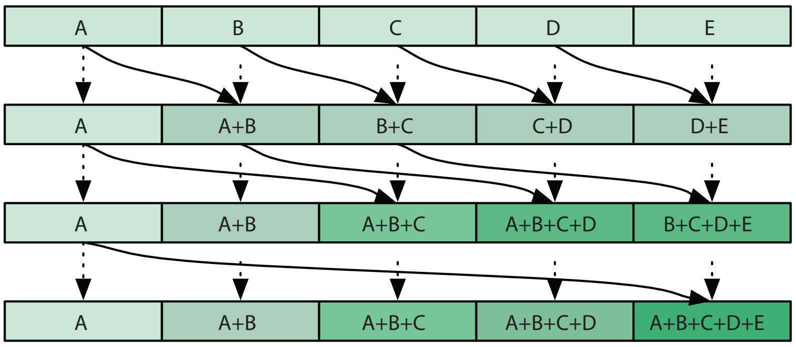


Figure 2.2: Illustration of the working principle of the algorithm proposed by Hensley et al. [9]. (figure from [9])

```
input  : An image and $r$ the number of texture lookups per pixel
output: Summed-area-table
$t_A \leftarrow$ Input Image;
n $\leftarrow log_r(width)$;
m $\leftarrow log_r(height)$;
//horizontal phase
for $i \leftarrow 0$ to $n$ do
    $t_B[x,y] = t_A[x,y] + t_A[x+1*r^i,y] + t_A[x+2*r^i,y] + t_A[x+3*r^i,y] + \cdots + t_A[x+r*r^i,y]$;
    $swap(t_A, t_B)$
end
//vertical phase
for $i \leftarrow 0$ to $m$ do
    $t_B[x,y] = t_A[x,y] + t_A[x,y+1*r^i] + t_A[x,y+2*r^i] + t_A[x,y+3*r^i] + \cdots + t_A[x,y+r*r^i]$;
    $swap(t_A, t_B)$
end
//$t_B$ holds the result
```

Algorithm 2.1: Summed-area-table generation algorithm proposed by Hensley et al. [9].

2.2 HDR-VDP2

The task of validating results in computer graphics and image processing is a challenging one and is often done by running an extensive user study. "Running such a user study for a big number of various images and algorithm parameters is inexpedient so, there is a need for computational metrics that could predict a visually significant difference between a test image and its reference, and thus replace tedious user studies" [19]. The visual metric proposed in [19] is according to the authors calibrated and validated against several data sets and image quality databases. The main focus of their work is: a calibrated visual model for scenes with arbitrary luminance range and the model they derived is the result of testing several alternative model components against a set of psychophysical measurements, choosing the best components, and then fitting the model parameters to that data [19]. In their work they propose two visual metrics, one for predicting visibility (P_{det}) and one for quality called the mean-opinion-score (Q_{MOS}). The generation of these metrics is illustrated in a block-diagram in Figure 2.3. The algorithm also produces a probability map, which gives a probability of detection for each pixel in the test image, allowing to illustrate those areas where an average observer is likely to notice a difference between a pair of images. The full source code of the project written in Matlab is available online[1].

[1]http://hdrvdp.sourceforge.net

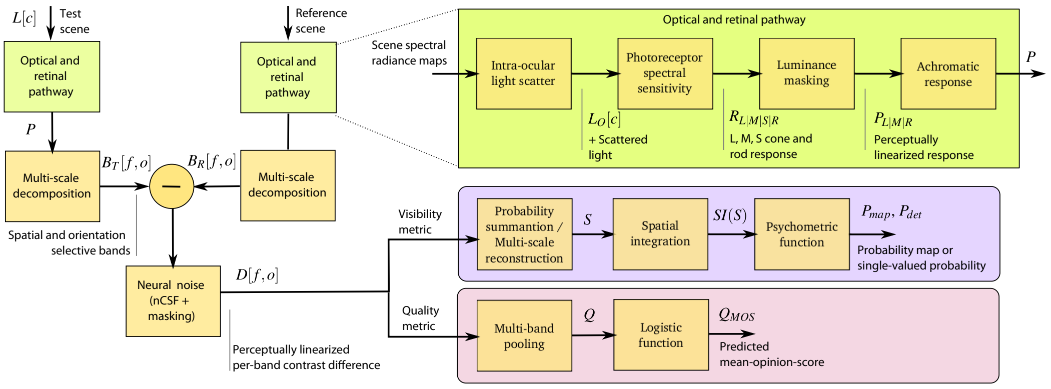

$L[c]$
Test scene
Optical and retinal pathway
P
Multi-scale decomposition
Spatial and orientation selective bands
$B_T[f,o]$
$B_R[f,o]$
Reference scene
Optical and retinal pathway
Multi-scale decomposition
Neural noise (nCSF + masking)
$D[f,o]$
Perceptually linearized per-band contrast difference
Scene spectral radiance maps
Optical and retinal pathway
Intra-ocular light scatter
Photoreceptor spectral sensitivity
Luminance masking
Achromatic response
P
$L_O[c]$
+ Scattered light
$R_{L|M|S|R}$
L, M, S cone and rod response
$P_{L|M|R}$
Perceptually linearized response
Visibility metric
Probability summantion / Multi-scale reconstruction
S
Spatial integration
$SI(S)$
Psychometric function
P_{map}, P_{det}
Probability map or single-valued probability
Quality metric
Multi-band pooling
Q
Logistic function
Q_{MOS}
Predicted mean-opinion-score

Figure 2.3: "The block-diagram of the two visual metrics for visibility (discrimination) and quality (mean-opinion-score) predictions and the underlying visual model." [19] (figure from [19])

2.3 Depth of Field

In photography, the depth-of-field effect originates from the physical nature of lenses. The lens lets light rays pass through onto the cameras sensor and in order for these rays to converge in a single point, their source has to be at a certain distance away from the lens. Everything at this exact distance projects to a point onto the image plain, whereas everything in front or behind that plane is projected to a region called Circle of Confusion (CoC). In photography, there is a range where the CoC is smaller than the resolution of the used sensor, therefore everything inside that range is being refereed to as in focus and everything else out of focus. Figure 2.4 shows the CoC produced on the image plane by an object out of focus. Note that the same principle applies to the human eye, with the retina corresponding to the image plane. The depth-of-field effect is often used to highlight certain objects or portions of a scene or to enhance the sense of depth in such. It can also be used in visualization application, as proposed by Kosara et al. [12] where parts of the scene are blurred according to their relevance. The proposed method is an alternative approach to the focus-and-context techniques commonly used in visualization.

In computer graphics, we usually render perfect pinhole camera images where everything is in focus. Therefore, depth-of-field as postprocessing effect has to approximate the blur occurring when using a real lens. To achieve this, several techniques have been proposed using different approaches to accomplish high quality depth-of-field simulation in real-time. One can simulate the physical light transport in a scene by using distributed ray tracing [4] across the surface of a nonpinhole lens. This method is especially suitable for high quality offline rendering, because of the high computational cost of tracing numerous rays through the scene. Other implementations like proposed by Bertalmio [1], use anisotropic diffusion to achieve the blur necessary on a per pixel basis. Layered depth of field methods, like those proposed by Cary Scofield [27], David Schedl [26] or Kraus et al. [15], solve the problem by breaking down the pinhole camera image into layers and filtering those layers individually. This is done by using a fragment's depth value to group pixels with a similar depth value into the same layer. These separated layers are then blurred uniformly and the blurred textures are afterwards blended together to

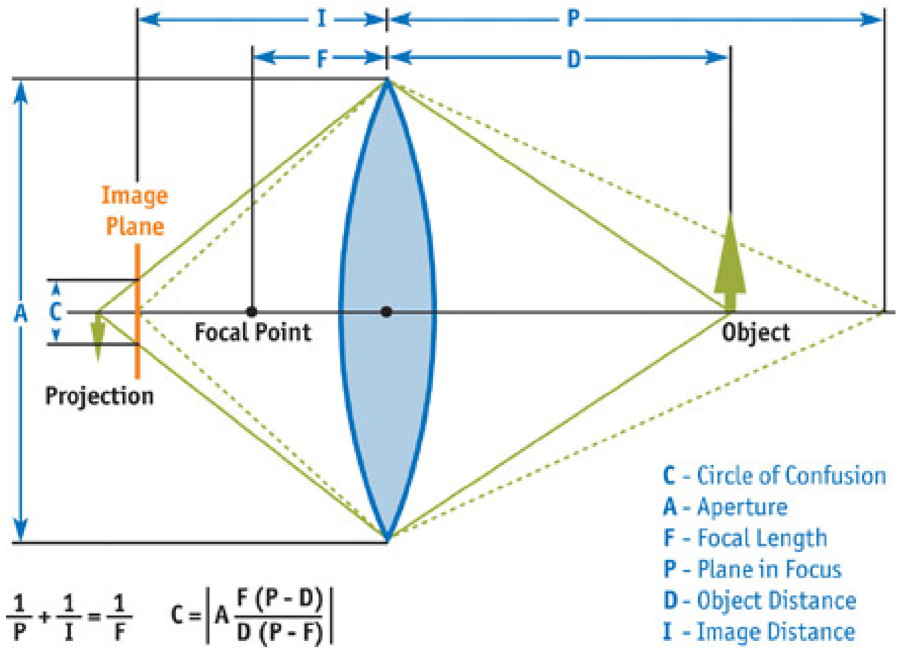


Figure 2.4: Illustration showing the Circle of Confusion produced by an object out of focus. (figure from [6])

form the result with depth-of-field. This approach suffers from the problem that some pixels, which would contribute to the image with depth-of-field, are not present in the pinhole camera image because of occlusion. The way this problem is tackled, besides the used technique of filtering the individual layers, is the main difference between the mentioned layered depth of field approaches. Figure 2.5 shows a comparison of the method proposed by Kraus et. al. [15] and a distributed ray tracer pbrt [22] for different lens parameters.

2.4 Glow

Glow or halos of light have been used in several modern computer games, including *The Elder Scrolls V:Skyrim*, *Project Gotham* or *Tron 2.0* and also animation movies like Pixar's *Finding Nemo* of which an example can be seen in Figure 2.6. “In everyday life, these glows and halos are caused by light scattering in the atmosphere or within our eyes” [29]. “In viewing computer graphics, film, and print, the intensity of light reaching the eye is limited, so the only way to distinguish intense sources of light is by their surrounding glow and halos” [20]. This effect can be realized by using a postprocessing effect, as it was implemented in the computer game Tron 2.0 and described in the GPU Gems Vol. 1 Chapter Real-Time Glow [7].

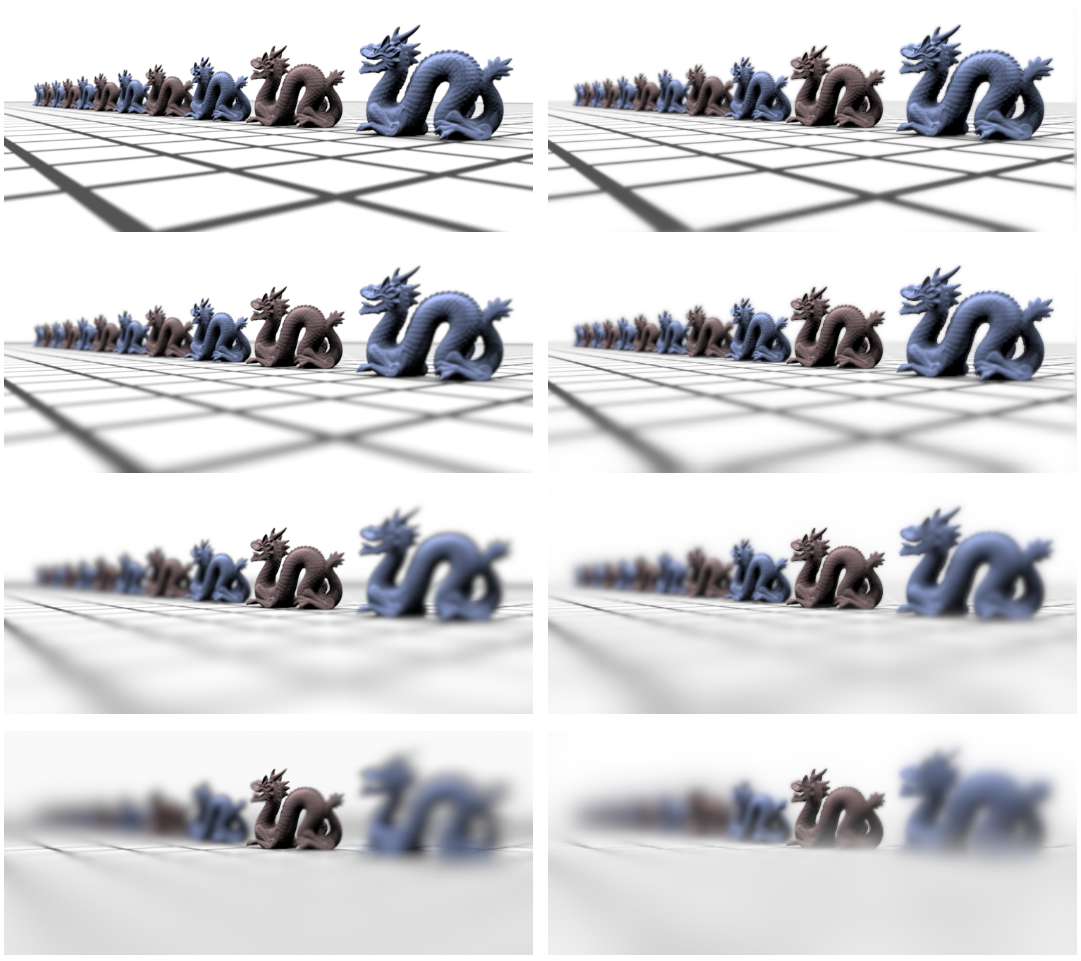


Figure 2.5: “Comparison of renderings with depth of field generated by pbrt [22] (left column) and images computed by Kraus and Strengert [15] (right column) for four settings of the pbrt parameter lensradius: 0.001, 0.003, 0.01, and 0.03 (from top down).” [15] (figure from [15])

Figure 2.7 presents an overview of this implementation, where (a) contains the whole scene and (b) only those parts on which the glow effect will be applied. These parts have to be tagged as glow source, for example by a texture mask or a geometry property flag, so they can be identified by the fragment shader and written into a separate texture. This texture is then blurred using separated convolution with a Gaussian filter kernel which is depicted in the steps from (b) to (c). Finally, the glow texture is blended together with the texture containing the whole scene using additive blending.

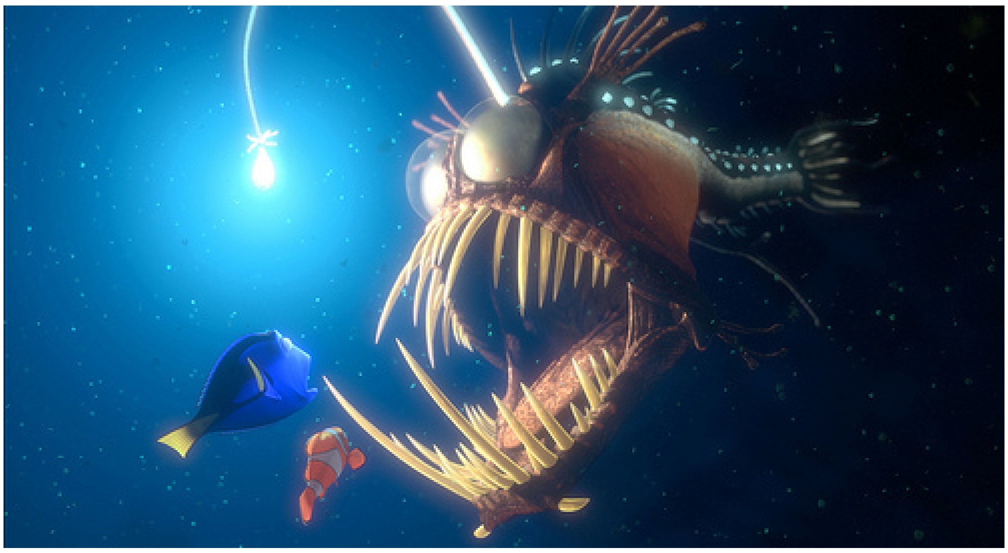


Figure 2.6: Example of the glow effect in Pixar's *Finding Nemo*.

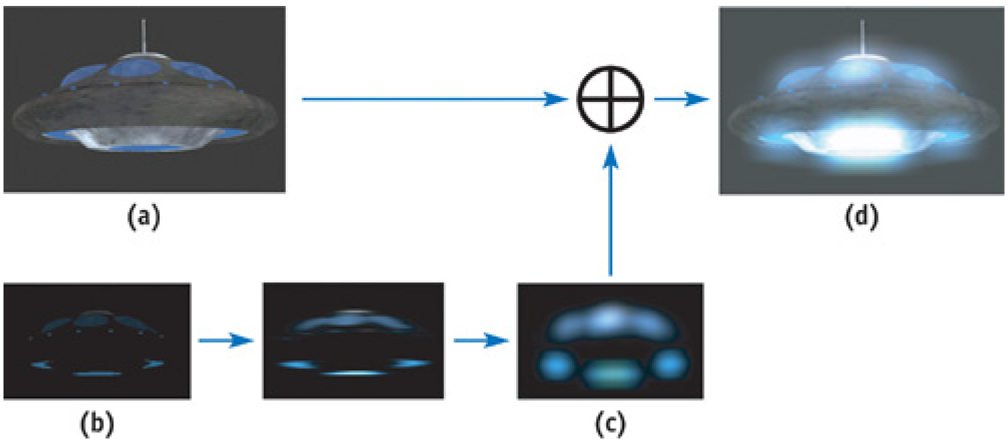


Figure 2.7: Overview of the glow effect proposed in GPU Gems Vol. 1 Chapter Real-Time Glow [7]. (figure from [7])

CHAPTER 3

Filtering Methods

This chapter starts by giving general information about the implementation and CUDA. The subsequent sections correspond to the chosen algorithms and contain their theory and working principle, as well as details on the actual implementation in GLSL and CUDA.

3.1 General

To implement and compare the various filtering methods, a simple framework was written in C++ using Microsoft Visual Studio 2010. As mentioned in the introduction, the decision was made to not only compare the performance and quality of the different methods but also the performance of a modern shading language versus a general purpose GPU computing language. This allows for a comparison of these two ways to implement image filtering on graphics hardware in terms of performance and suitability for the given methods. OpenGL 4.2 was chosen as graphics API to handle rendering to screen and implementing the filtering methods in the associated shading language GLSL. As general purpose GPU computing language, CUDA from NVIDIA was used in version 4.2 with compute capability 2.1. To facilitate the implementation, a few libraries were used for certain tasks: window management library GLFW[1] was integrated, GLM was used as mathematics library, and DeviL was incorporated to load images into OpenGL textures and to write those to hard disk. For the quality analysis done between the filtering methods, the necessary procedures were written in MATLAB R2011b. The full source code of the project is available at google code under the following address: http://code.google.com/p/comparison-of-fast-image-filter-techniques.

[1]http://www.glfw.org

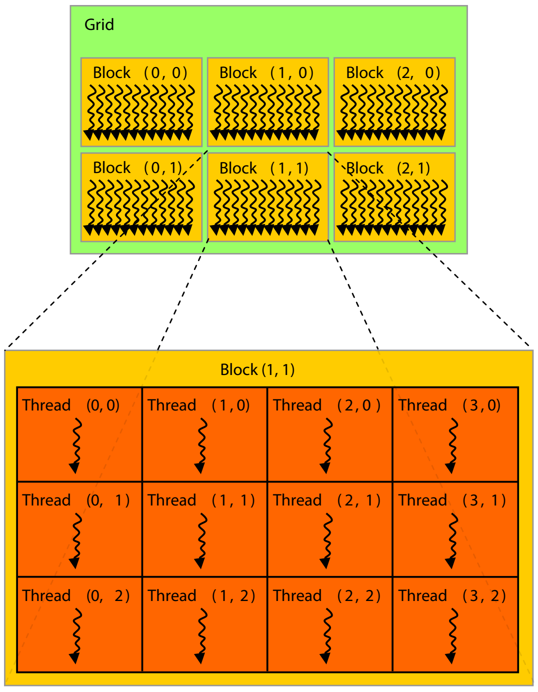


Figure 3.1: Grid of thread blocks. (figure from [21])

3.1.1 CUDA

In CUDA, the procedures which are actually executed on the GPU are called kernels. Those kernels operate on a two-dimensional grid of threads, which is subdivided into two-dimensional thread blocks (threads in one block are executed by a single multiprocessor). Therefore, the equivalent in CUDA to rasterizing a rectangle and have the fragment shader traverse all pixels of a given texture is to define the block dimensions such that they are integer divisors of the respective texture dimensions. Those divisors are then the number of blocks started by the kernel in each dimension respectively (the so-called grid), resulting in one thread per pixel in the input texture. The needed texture coordinates can then be calculated using the provided variables for thread and block index. This structure is illustrated in Figure 3.1 with the grid on the top and the structure of a single block at the bottom. To give a kernel access to data –this could be image data but could also be vertex data for example– it has to be uploaded to the video memory of the GPU. The usual way to do this is to use for example `cudaMemcpy` to copy a chunk of memory from CPU memory to video memory. In real-time rendering applications, our image data usually already resides in video memory (e.g. rendering into a framebuffer). Therefore, the detour to load the texture memory into CPU memory (which is very slow) and afterwards copying the whole content to the GPU memory using `cudaMemcpy` can be avoided. For this scenario, CUDA offers OpenGL interoperability features which allow to flag OpenGL textures and buffers to be used inside a CUDA kernel without copying the whole content. In this context, CUDA offers two functions, called `cudaGraphicsGLRegisterImage` and `cudaGraphicsGLRegisterBuffer`, to register an OpenGL resource with CUDA. The registered resource can then afterwards be mapped for CUDA, which will lock the texture for the usage with CUDA. Any access to the texture object using OpenGL will have undefined behavior according to the CUDA Programming Guide. Considering this, the execution of a CUDA kernel in our project is always preceded by the mapping of one or more textures and is always followed by an unmapping step. The `cudaArray` resulting from the mapping process can then, for instance, be bound to a texture variable, which offers read-only access. Another possibility is to bind it to a surface variable, which offers both read and write access to the underlying texture. Therefore, the input texture is always bound to a texture variable and the output texture to a surface variable.

Shared memory

In CUDA, the shared memory is a fast type of memory which is shared between all threads of a threadblock. This memory functions as a user-managed cache and is limited to 48KByte per multiprocessor. "To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously" [21]. Therefore, memory requests, inside one warp (= 32 threads), of addresses that fall into distinct memory banks can be served in parallel. However, if two addresses of a memory request fall into the same memory bank, which is called a bank conflict, the access has to be serialized and therefore the throughput is decreased. For more information on shared memory and bank conflicts the interested reader should refer to the CUDA Programming Guide [21].

CUDA Libraries

The CUDA SDK already comes with a library called cuFFT, which allow executing the Fast Fourier Transform on 1D, 2D and 3D data of arbitrary size. The FFT algorithm utilized in this library is based on the work from Cooley-Tukey and Bluestein [10].

The CUDA Data Parallel Primitives Library (CUDPP)[2] can be used to efficiently implement summed-area-tables in CUDA.

3.2 Convolution

3.2.1 Theory

In the field of functional analysis, the convolution of two functions f and g is defined as follows:

$$(f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{3.1}$$

When this equation is applied with f and g being box functions, the result is a tent function, which can be seen in Figure 3.2.

Applications of convolution include computer vision, statistics and for this work most importantly image and signal processing. When looking at image processing, convolution offers a

[2]http://code.google.com/p/cudpp

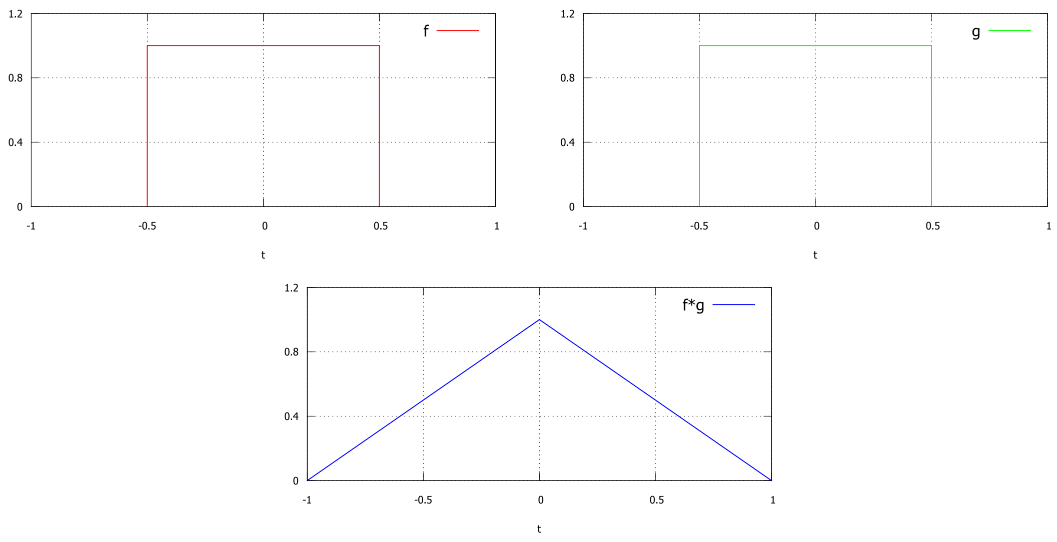


Figure 3.2: Convolution example: box functions f and g and the resulting tent function $f * g$

general method for applying different filters to an image e.g. blurring, sharping, edge-detection. In image processing, the filter function (called kernel in this context) and the image itself consist of finite, discrete data. When f and g are discrete functions, the integral in the convolution formula becomes a sum:

$$(f * g)[n] \equiv \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{3.2}$$

The above mentioned formulas are given for one-dimensional data, whereas in image processing we deal with two-dimensional data. Given an image I and a filter kernel h of size $m \times m$, the result R of two-dimensional convolution for a pixel at position (x, y) is given by:

$$R(x,y) = \sum_{i=-\lfloor \frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{j=-\lfloor \frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} h(i,j)I(x-i, y-j) \tag{3.3}$$

To obtain the filtered output image, this formula has to be applied pixel by pixel to the input image. When doing so, the filter kernel is moved over the entire input image and the output is calculated as weighted sum of the pixels covered by the kernel. The calculated weighted sum is then written into the output image at the current filter location. If the output image should not be overall darker or brighter as the input image, the filter kernel has to be normalized so that the sum of all elements is 1.0.

As mentioned above, there are several effects which can be achieved by using different filter kernels. Figure 3.3 shows different kernels which are commonly used in image processing.

$$h_{box} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad h_{laplace} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad h_{gauss} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 3.3: Sample 3x3 kernels: h_{box} is a simple average filter, every pixel in the output image when convolved with this kernel is the average of its nine neighbours. $h_{laplace}$ is an approximation of the Laplacian operator which can be used for edge detection, and h_{gauss} is a Gaussian filter kernel commonly used for smoothing/blurring an image.

3.2.2 Implementation

GLSL

To provide the fragment shader with the needed texture coordinates to traverse each pixel, a full-screen rectangle is rendered by passing 4 vertices as a triangle-strip to the vertex shader. In the case of convolution, the fragment shader also needs access to the filter kernel in shape of offsets and weights. This could be achieved by calculating the offsets and weights at each fragment using for example the Gaussian distribution formula. Another possibility is to calculate the offsets and weights offline on the CPU and transferring them to the GPU as uniform variables. Considering the high overhead of evaluating the whole filter kernel at each pixel, the latter variant was chosen. When using arrays of uniform variables, the filter size gets limited to quite low values by `GL_MAX_FRAGMENT_UNIFORM_COMPONENTS`. Note that the offsets are two-dimensional float vectors containing normalized texture coordinates, while the weights are scalar float values.

To overcome these limitations, we implemented two ways of passing the offsets and weights to the fragment shader which allow for bigger filter sizes. The first one is to use uniform buffer objects. Uniform buffer objects were introduced with OpenGL version 3.1 and offer a minimum (depending on the actual graphics hardware and driver) of 64KByte per uniform buffer object. When we split weights and offsets into separate uniform buffer objects, one holding scalars and the other one holding two-dimensional vectors, the maximum filtersize can be calculated as follows:

$$\begin{aligned} 65536 \text{ Byte } / \text{ 2 floats per offset } &= 32768 \\ 32768 \text{ Byte } / \text{ 4 bytes per float } &= 8192 \\ \text{filter is square } \sqrt{8192} &\approx 90 \end{aligned} \tag{3.4}$$

As shown, the maximum filter size when using one uniform buffer object which contains both components of the offset values is 89×89, assuming a maximum buffer size of 64KByte.

Another way to give the fragment shader access to the needed data is to use texture buffer objects, which were also introduced in OpenGL version 3.1. They offer 128MByte and more memory, depending on the actual driver and hardware setup. Texture buffer objects can be accessed in GLSL via the `texelFetch` functions. Compared to uniform buffer objects, which reside in local video memory, texture buffer objects reside in global video memory and the access involves the hardwares texture units. Therefore, texture buffer objects allow for a bigger filter size as a trade-off for performance (see Section 4.2.2).

The actual implementation of the convolution process is straight forward, at each fragment in the fragment shader, a loop runs over the filter window, multiplying the corresponding input pixels with the associated weights. The resulting values are accumulated and the final sum is written into the output image.

CUDA

For the CUDA implementation of convolution, a naive shared-memory algorithm was implemented. The algorithm loads the image data needed to execute convolution for a single thread block into shared-memory. By doing so, after the initial loading step, every future lookup to image data can be served by shared-memory. Because shared-memory is only shared between the threads in a single thread block, it is not sufficient to load pixel values inside the given block into shared memory. Instead, it is also necessary to load a so-called halo. The halo is a border around the actual block, as wide as the filter radius, to give all threads in the block access to the data necessary needed to execute convolution. The necessary filter weights are copied to constant memory before the execution of the actual kernel starts.

When loading the necessary texture data into shared memory, we have to decide which threads in a thread block load the needed halo data. This could be done by checking for border threads and only loading halo data inside these threads. However, this approach comes with the disadvantage that the load between threads inside a thread block is not well balanced. While some threads (in the middle of the thread block) only load a single pixel, the threads on the border of the thread block load multiple pixels into shared memory. Therefore, this approach results in a lot of idling threads during the loading stage of the algorithm. These idling threads can be circumvented, as proposed in the CUDA SDK document “Image Convolution with CUDA” [23], by letting each thread load multiple values into shared memory. We have chosen an approach where every thread loads the four pixel values which constitute the corners of the filter window centered at the texel into shared memory. By applying this procedure, we can assure that the whole filter window for each thread in a block (including halo) is loaded into shared memory. Note that this holds true as long as the width of the thread block is higher than the width of the filter. The idea to let every thread load the four corner values of the filter window was taken from the work of Sangyoon Lee named “CUDA Convolution” [18]. Figure 3.4 depicts this behavior by showing the loaded pixel values with arrows for two of the pixels inside a 4x4 block when a filter size of 3x3 is used. By doing so, the filter size is not only limited by the maximum shared memory size (48KByte) but also by the block size. When using this implementation, depending on the used filter and block size, there is the possibility for pixel values getting loaded multiple times. However, this approach allows executing convolution with four texture lookups per pixel, whereas all future accesses to pixel values, after the loading step, are served through shared memory. By doing so, the filter size is not only limited by the maximum shared memory size (48KByte) but also by the block size. When using this implementation, depending on the used filter and block size, there is the possibility for pixel values getting loaded multiple times. However, this approach allows executing convolution with four texture lookups per pixel, whereas all future accesses to pixel values, after the loading step, are served through shared

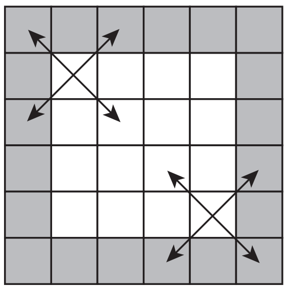


Figure 3.4: Example for loading pixel values for a block size of 4x4 and a filter size of 3x3 into shared memory: actual block pixels are marked in white, whereas halo pixels are marked in grey.

memory. The difference between CUDA and GLSL is that in CUDA we have the chance to achieve a speed up through the usage of shared memory, whereas in GLSL the cached texture lookups could make up for that.

3.3 Separated convolution

3.3.1 Theory

3.3.2 Separable Convolution

When a two-dimensional function can be expressed as the product of two one-dimensional functions (one in each direction), it is called separable. In Equations 3.5-3.7 we show the Gaussian distribution as an example for such a function. Equation 3.8 shows how the two-dimensional h_{gauss} kernel, presented in Figure 3.3, can be separated into two one-dimensional kernels.

$$f(x,y) = f_{hor}(x) * f_{vert}(y) \tag{3.5}$$

$$f(x,y) = \frac{1}{2\pi\sigma^2} * e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.6}$$

$$f_{hor}(x) = \frac{1}{\sqrt{2\pi}\sigma} * e^{-\frac{x^2}{2\sigma^2}} \qquad f_{ver}(x) = \frac{1}{\sqrt{2\pi}\sigma} * e^{-\frac{y^2}{2\sigma^2}} \tag{3.7}$$

$$h_{gauss} = \frac{1}{16}(h_{gaussVer} * h_{gaussHor}) \quad h_{gaussVer} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad h_{gaussHor} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{3.8}$$

Convolution is associative, which means if a filter kernel s is separable into v and h, then filtering with s yields the same result as first filtering with v and then filtering with h (vertical and horizontal respectively).

$$f * (v * h) = (f * v) * h \tag{3.9}$$

If the desired filter kernel is separable, the convolution process can be split up into two one-dimensional filter passes (one in each direction), thus achieving a speed-up due to the fact that less operations per pixel are required. To explain this speed-up, we examine the number of multiplications and additions needed in both cases. Given an image of size $N \times M$ and a filter kernel of size $K \times$L, the non-separated two-dimensional convolution requires $NMKL$ multiplications and additions. In the separable case, the horizontal pass requires NMK operations and the vertical pass NML, which sums up to a total of $NM(K + L)$ multiplies and adds. In the case of a 15×15 filter kernel, the two dimensional convolution requires 225 operations per pixel, whereas separated convolution only requires 30, which is a theoretical speed up of 7.5. Note that every operation per pixel, in case of a GPU implementation, also requires a texture lookup.

3.3.3 Implementation

GLSL

As explained in Section 3.2.1, a separated filter kernel has to be applied in two passes, a horizontal and a vertical one. Uniform buffer objects are used, as for the non-separated convolution implementation, to transfer the necessary offsets and weights to the fragment shaders of the passes. For a single pass, the number of texture lookups is equal to the filter kernel width. Besides the performance boost, splitting the convolution process into two one-dimensional passes additionally impacts the maximum filter size. Since now our filter kernel is one-dimensional, we can omit the last step of Equation 3.4, which results in a maximum filter size of 8000×8000.

CUDA

In CUDA, the implementation follows the same approach as for two-dimensional convolution. The only difference is that only two pixel values per thread have to be loaded into shared memory. For the horizontal pass, a halo at the left and right and for the vertical pass a halo at the top and bottom have to be loaded, which is illustrated in Figure 3.5. The illustration also shows that we process two rows(columns) per thread block to achieve a bigger throughput per multiprocessor. Therefore, it is sufficient to load two pixel values into shared memory per thread to assure every thread has access to the whole filter window needed to execute separated convolution. Note that this again only holds true if the width (height) of the thread block is bigger than the filter size.

Two different ways of laying out the shared memory have been tested, where one uses an array of structs containing 4 float values (float4), whereas the other one splits the RGBA image into four float arrays. This was done with consideration concerning bank conflicts and the impact of those variations are shown in Section 4.2.3.

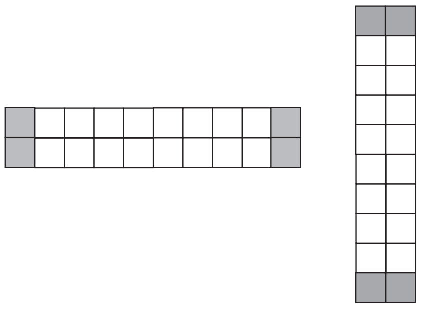


Figure 3.5: Example for the shared memory layout for separable convolution for a filter size of 3x3 and a block size of 8x2 and 2x8 respectively: actual block pixels are marked in white, whereas halo pixels are marked in grey. Note that we process two rows(columns) per thread block to achieve a bigger throughput per multiprocessor.

3.4 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an efficient procedure for the calculation of the Discrete Fourier Transform (DFT). The DFT maps a discrete, finite signal (time domain) to a periodic frequency spectrum (frequency domain). Its counterpart, the inverse Discrete Fourier Transform (iDFT), is used to reconstruct the signal from a given spectrum. In 1965, Cooley and Tukey published the most famous algorithm for the FFT [10], which is a divide and conquer algorithm. The relevance of FFT in terms of image filtering is given by the convolution theorem, which states that convolution in the time domain equals a point-wise multiplication in the frequency domain and vice versa. Let F be the Fourier Transform operator and f and g two functions in the time domain. Furthermore, let $*$ denote the operator for convolution and $\cdot$ the operator for point-wise multiplication, then the convolution theorem can be written as follows:

$$F\{f * g\} = F\{f\} \cdot F\{g\} \tag{3.10}$$

$$F\{f \cdot g\} = F\{f\} * F\{g\} \tag{3.11}$$

This theorem, combined with the FTT, allows implementing image filtering with constant time complexity regarding filter size. This is done by first applying the FFT on the input image and on the filter kernel. The filter kernel has to be padded to the same size as the input image (see Section 3.4.1). After both the input image and the filter kernel are transformed to the frequency domain, they are multiplied pixel by pixel, as stated by the convolution theorem, to achieve convolution in the time domain. The result of the point-wise multiplication is then transformed back to the time domain using the inverse Fast Fourier Transformation (iFFT). The costs of this procedure do not depend on the size of the filter kernel but only on the size of the image. For color images, the procedure explained above has to be applied for each color channel separately.

3.4.1 Implementation

CUDA

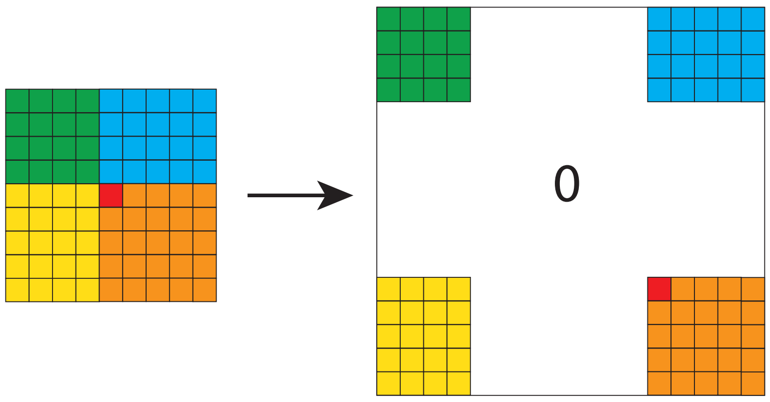


Figure 3.6: Illustration of how the filter kernel is padded to achieve cyclic convolution based on FFT. Original filter kernel on the left and zero-padded filter kernel on the right.

As mentioned in Section 3.1.1, cuFFT was used to implement FFT with CUDA. When using FFT and the convolution theorem, the filter kernel has to be padded in a cyclic fashion. This padding step is illustrated in Figure 3.6, with the original kernel on the left side and the padded filter kernel on the right side. The colored areas show where the respective areas from the original filter kernel are in the zero-padded filter kernel. The FFT can be executed on two-dimensional single channel data, therefore the image is split into one array per channel. After the image has been deinterleaved, FFT is applied to each color channel of the input texture and the padded filter kernel. In the following step, the resulting spectrums of each channel are multiplied point-wise, according to the convolution theorem, with the spectrum of the padded filter kernel. The results are then transformed back into the time domain using iFFT and subsequently interleaved into one texture to obtain the filtered output image.

3.5 Pyramid methods

3.5.1 Theory

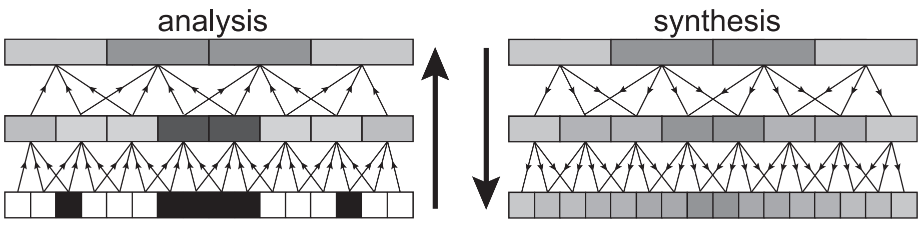


Figure 3.7: Illustration of blurring an image with the pyramid method in 1D. (figure from [17])

The usage of image pyramids to blur an image, which is a frequently used task in real-time rendering for various effects like glow and depth-of-field, was first proposed in 1981 by Peter J. Burt in his work "Fast filter transform for image processing" [2]. Typically, an image pyramid is created by first filtering the input image using a so-called analysis filter, followed by a sub-sampling step (usually by a factor of two) in each dimension. This process is repeated until the desired number of levels are constructed or the lowest level of the pyramid is reached, which is when one of the two image dimensions has reached the size of one pixel. During this process, the analysis filter determines how pixels of a the finer level are combined to obtain a pixel in the coarser level during down-sampling. To obtain a blurred version of the input image, a chosen pyramid level is iteratively up-sampled using a filter known as synthesis filter. This filter determines how pixels of a coarser level are combined to calculate a pixel in the next higher level during up-sampling. The amount of blur applied to the image is controlled by the chosen pyramid level, where a lower level means a larger blur radius. Overall, pyramid methods achieve speed by applying a small filter combined with down-sampling and up-sampling to achieve big blur radii. Equations 3.12-3.13 show how many texture lookups are needed to down-sample an image 3 times using a 2x2 box filter and afterwards up-sampling it to the full resolution of the input image using a biquadratic B-spline [3] synthesis filter. Also the formulas for the number of texture lookups for each step individually are given where d denotes the number of down-sampling steps and u the number of up-sampling steps (both 3 in our example).

$$\begin{gathered}\text{down-sampling:}\\ \sum_{i=1}^{d}\frac{n}{2^{i+1}}\\ 3\rightarrow 2:\frac{n}{8}\\ 2\rightarrow 1:\frac{n}{4}\\ 1\rightarrow 0:n\end{gathered}\tag{3.12}$$

up-sampling:

$$\begin{aligned} n + \sum_{i=1}^{u-1} \frac{n}{2^{i+1}} \\ 3 \rightarrow 2 : \frac{n}{8} \\ 2 \rightarrow 1 : \frac{n}{4} \\ 1 \rightarrow 0 : n \end{aligned} \tag{3.13}$$

The total number of texture lookups in our example is given by $1.8125 * n$. According to our tests (see Section 4.4), the blur applied by down and up-sampling the input image three times achieves approximately the same blur radius as convolution with Gaussian filter of width 39. Note that the number of texture lookups needed to apply this filter is $39^2 * n$ ($m^2 * n$ with m denoting the filter width and n the number of pixels in the input image). The downside of this approach is the low control over the filter size, which can only be chosen in certain steps (chosen level). This process in 1D is illustrated in Figure 3.7, with the down-sampling step on the left and the up-sampling step on the right side. The most prominent example for the utilization of image pyramids, in real-time rendering, is mipmapping, which was introduced by Lance Williams in "Pyramidal Parametrics" [31]. Mipmaps are image pyramids where the input image is down-sampled to a resolution of 1x1 or 2x2 pixels. In texture mapping, minification aliasing effects can be suppressed to a certain level by using mipmaps. This is done by choosing a level of the pyramid so that a texture element (texel) is big enough to fully cover a pixel on screen. Figure 3.8 shows an input image of resolution 128x128 and all its mipmap levels. Note that mipmaps are an example for the analysis part of pyramidal blurring and do not incorporate any synthesis steps. The number of levels that can be created is limited by the original resolution of the input. The maximum number of levels when using a scaling factor of two and the constrain that the lowest dimension in level N_{max} is 1 is calculated by the following equation:

$$N_{max} = \lceil \log_2(\text{max(width,height)}) + 1 \rceil \tag{3.14}$$

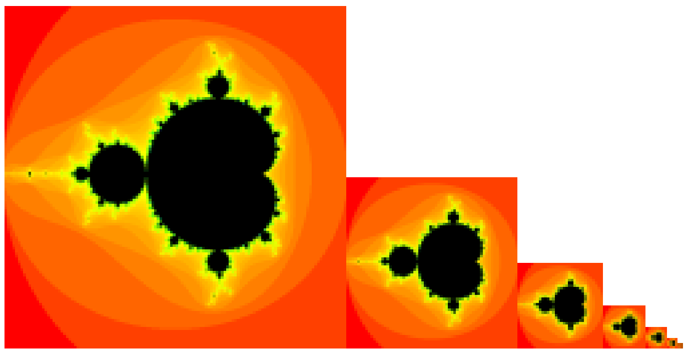


Figure 3.8: Mipmap levels.

3.5.2 Quasi Convolution

Pyramid methods create different blurring effects depending on the choice of analysis and synthesis filter. In graphics, we are mostly interested in a blur that resembles convolution with a Gaussian. To arrive at a pyramidal method that achieves such a blur, Kraus [17] did a quantitative analysis of the response function of pyramidal blurring using different analysis filters (e.g. 2x2 and 4x4 box) to analyze the deviation of these methods compared to convolution with a Gaussian kernel.

In his work, Kraus used a biquadratic B-spline [3] synthesis filter in all scenarios. The process of applying this filter during up-sampling can be seen as first subdividing every pixel in the coarser level into four sub-pixels. A bilinear texture lookup is then placed according to the texture coordinates of the finer level at on of these sub-pixels. The result written into the finer level is therefore a weighted average of four pixels of the coarser level.

By using numeric methods, based upon this analysis, he came up with a new analysis filter, which he calls "quasi-convolution analysis filter". This filter can be used for down-sampling, instead of for example the 2x2 box filter used for mipmaps in OpenGL, to create the image pyramid. Compared to the 2x2 box filter, which takes the average of four pixels of the finer level to calculate a pixel of the coarser level, the quasi-convolution filter calculates a weighted average of 16 pixels. According to the comparison done by Kraus [17], the pyramidal blur, using the quasi-convolution analysis filter, deviates less from the corresponding Gaussian convolution filter than the 2x2 and 4x4 box analysis filters. In Equation 3.15, the quasi-convolution analysis filter is given for both 1D and 2D, where the 2D version is obtained by taking the tensor product of the 1D variant. The tensor product of two vectors u and v is obtained by multiplying each element of u by each element of v. The result is a $m \times n$ matrix with m being the length of vector u and n being the length of vector v. Note that as mentioned in Section 3.3, if a two-dimensional filter kernel can be written as tensor product of two one-dimensional ones, it is separable.

$$\frac{1}{64}\begin{bmatrix} 13 & 19 & 19 & 13 \end{bmatrix} \qquad \frac{1}{4096}\begin{bmatrix} 169 & 247 & 247 & 169 \\ 247 & 361 & 361 & 247 \\ 247 & 361 & 361 & 247 \\ 169 & 247 & 247 & 169 \end{bmatrix} \tag{3.15}$$

Kraus and Strengert [16] presented different analysis filters and an efficient way to implement them on a GPU using bilinear texture lookups. The implementation of these filters is based on the idea proposed by Sigg and Hadwiger [28]. This idea can best be illustrated using the 2x2 box filter as example. Instead of placing four texture lookups with nearest-neighbour sampling and calculating the average of these four values, one can place a single bilinear texture lookup at the crossing point of the four pixels (see black dot in Figure 3.9). By doing so, the averaging is done by the hardware's texture units and the number of texture lookups needed is reduced by factor of 4. The quasi-convolution filter can be applied with four bilinear texture lookups by using the same principle. In Figure 3.9, the position of those lookups (green dots) are shown.

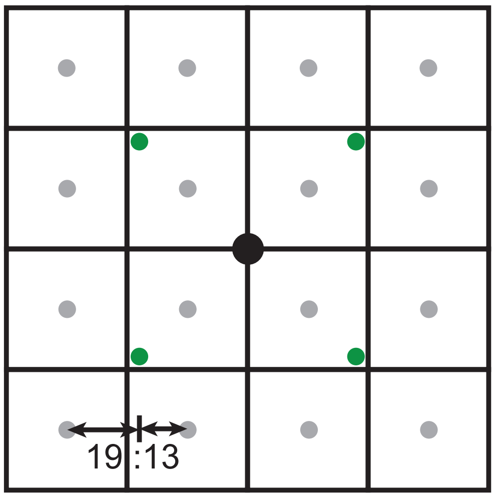


Figure 3.9: Illustration of the positions (green) of the four bilinear texture lookups used for the quasi-convolution analysis filter. "The centers of texels of the finer level are indicated by grey dots, while the black dot indicates the center of the processed texel of the coarser image level."[17] (figure from[17])

3.5.3 Implementation

GLSL

The up-sampling steps are implemented as shown by Strengert et al. [30], by using a biquadratic B-spline [3] synthesis filter, which requires a single bilinear texture lookup. This procedure is equal for all tested pyramid methods, as they only differ in the way down-sampling is achieved. In our case, to implement biquadratic B-spline filtering, we rasterize a full-screen rectangle with the viewport set to the size of level $N+1$, where N is the level we want to apply the up-sampling step to. By doing so, the screen-space coordinates passed by the vertex shader to the fragment shader are already at the positions where we want to place a bilinear texture lookup. This process is repeated until the highest level of the pyramid is reached, which is of the same resolution as the input texture, yielding the final result of the pyramid filtering.

For the down-sampling steps, three different analysis filters have been implemented, namely 2x2 and 4x4 box filtering and the quasi-convolution filter proposed by Kraus [17]. The 2x2 box filter is done by using a single bilinear texture lookup based on the idea of Sigg and Hadwiger [28]. To do this, the same procedure as for the biquadratic B-spline up-sampling is used, in fact the same vertex and fragment shader are used. The difference is that the viewport is now set to the size of level $N-1$ instead of $N+1$, which corresponds to the level with a resolution two times lower than the level currently processed. For the 4x4 box analysis filter, we use two passes per down-sampling step to implement the filter with two bilinear lookups as shown by Strengert et al. [16]. First, the currently processed level is down-sampled using a $h^{\nwarrow}_{box}$ analysis filter, followed by an additional filtering step with a $h^{\searrow}_{box}$. Both are shown in Equation 3.16 and can be realized with a single bilinear texture lookup. The quasi-convolution analysis filter was implemented as Kraus [17] proposed by using four bilinear texture lookups placed as shown in Figure 3.9 and taking the mean of those four values. The single down-sampling steps are repeatedly applied until the desired number of pyramid levels is obtained or the maximum number of levels according to Equation 3.14 is reached. In addition, down-sampling utilizing the mipmap pyramid generated by OpenGL, using the function `glGenerateMipMaps`, was implemented for comparison against the above-mentioned analysis filters in terms of performance. Note that this function applies a 2x2 box analysis filter.

$$h_{box}^{\nwarrow} = \frac{1}{4}\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad h_{box}^{\searrow} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{3.16}$$

CUDA

CUDA provides a bilinear filtermode for texture lookups and therefore the implementation is equal to GLSL. This means that each thread in CUDA executes the same operations as a pixel in the GLSL fragment shader. Note that every frame, the image pyramids have to be mapped and unmapped for the usage in CUDA.

3.6 Spreading

3.6.1 Theory

In his PhD thesis [13], Kosloff presented several methods based on “spreading” the Point Spread Function (PSF) of a filter. He describes his methods with “running summed-area-tables in reverse” [13]. To understand his approach, it is best to look at an example of the method he calls “Rectangular Spreading” in one dimension. Figure 3.10 illustrates the basic idea behind the method. The first plot shows a one-dimensional box function and the second one contains the discrete derivation (finite differencing) of that function.

The proposed method now uses the derivation of the box function, which contains a negative and a positive impulse (note that in two dimensions, the derivation of the box filter contains two positive and two negative values), to apply average filtering. To do so, the image is traversed pixel by pixel and the four values of the derivation of the box filter are spread out. This means that at every pixel, the color value is weighted with the derivations and the corresponding weighted colors are added to the values at the location of the corners of the box window. The method proceeds by integrating the image by calculating a summed-area-table (different ways of calculating a summed-area-table can be found in Sections 2.1 and 3.6.3) of the intermediate image containing the spread derivation values. These steps are illustrated in Figure 3.11.

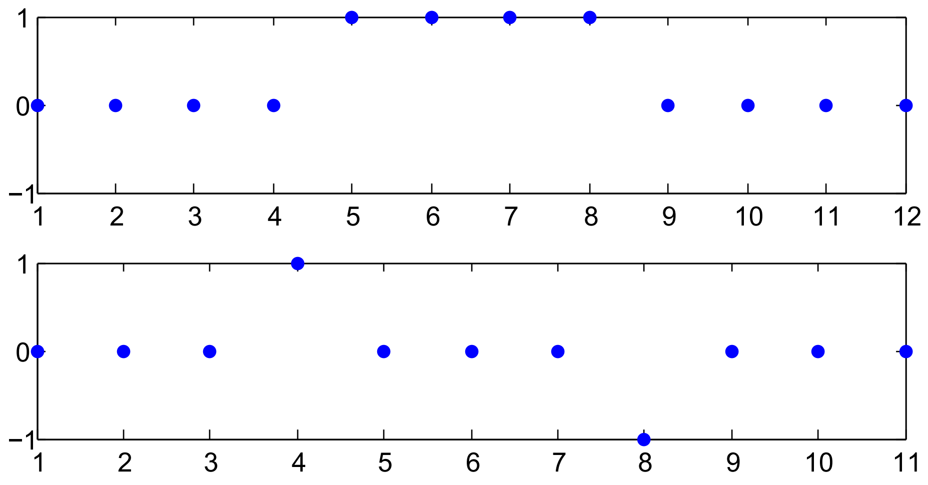


Figure 3.10: Rectangular spreading: 1D discrete box function and its derivation. [13]

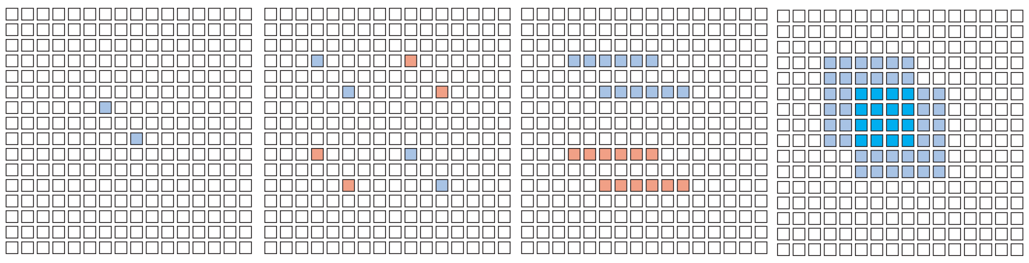


Figure 3.11: Applying the Rectangular Spreading method (from left to right) with a radius of three: Input image, result after spreading the four corner pixels, result after horizontal integration step (left to right), final result after vertical integration step (top to bottom). Note that the redish values indicate negative values and more saturated colors denote double the value of less saturated colors. [13]

3.6.2 Polynomial Spreading

Kosloff also introduced a method he presents as generalization of the “Rectangular Spreading” method, which is called “Polynomial Spreading”, where rectangles are simply first order, constant-valued polynomials [13]. Out of all the variations of spreading presented in his work, he has chosen this method to be implemented in a depth-of-field demonstration for the graphics card and processor manufacturer Advanced Micro Devices (AMD). “The method uses the fact that piecewise-polynomial filter responses become sparse after we take enough derivations” [13].For example, a constant-valued impulse response has a sparse derivative, a linearly-varying impulse response has a sparse second derivative, and a quadratic impulse response has a sparse third derivative [13]. Figure 3.12 shows two functions and their derivations until sparsity emerges. The figure can be seen in both directions, from top to bottom the function is integrated in each

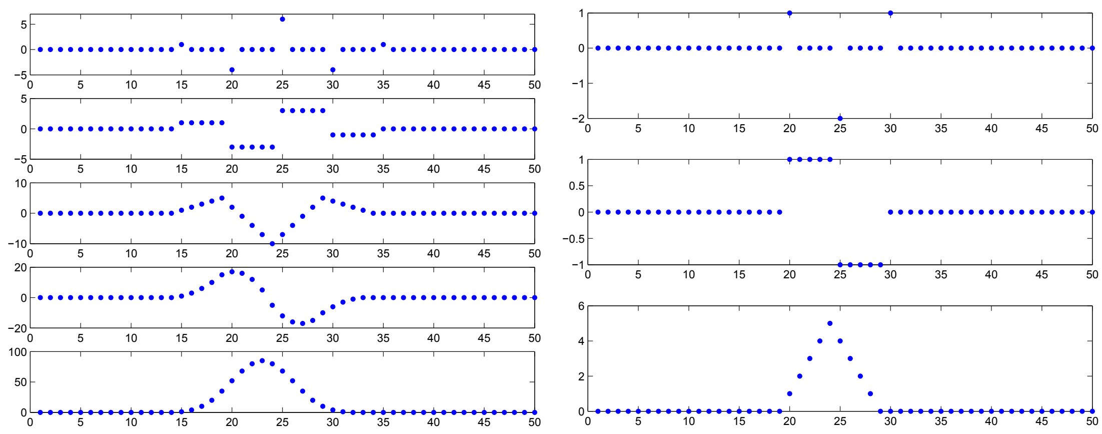


Figure 3.12: Sparse derivation of discrete functions (bottom to top): in the left column, sparsity emerges with the fourth derivation, in the right column only two derivation steps are needed. [13]

step, whereas from bottom to top the function is deriviated in each step. In Figure 3.13, the working principle of polynomial spreading in 1D is shown step by step. The technique starts by spreading the 2nd deriviation of the tent filter (see Figure 3.12) on the right), shown in Figure 3.17, with a spreading radius of four pixels. As can be seen, this is done by multiplying each pixel with the weights of the filter and adding the resulting value to the pixels at a distance of four (except for the center value) to the left and right. The obtained intermediate image is then integrated twice by calculating summed-area-tables. Note that an additional normalization step is required to obtain the final result. This normalization step was omitted in Figure 3.13 and is done by dividing the result of the summed-area-table generation by the spreading radius to the power of two. To apply the method in 2D, in his implementation for AMD, Kosloff used the tensor product of the function shown on the right side in Figure 3.12. In 2D, the method follows the same steps as in 1D, but spreads the values in nine directions (including the center, see Equation 3.17) and the normalization has to be done by dividing by the spreading radius to the power of four. Overall, this method achieves constant time complexity with regards to filter size when used for image blurring. The summed-area-table generation only depends on the input resolution and can be done in $O(n * log(n))$ as shown by Hensley et al. [9]. Due to this constant time complexity with regards to filter size, this method is especially suited for very large blur radii. Note that this method is applicable to every filter which has a sparse Nth derivation. This is done by spreading these derivation values and integrating the image N times.

3.6.3 Implementation

GLSL

As representative for the various spreading methods proposed by Kosloff [13], the polynomial spreading method was implemented. To implement the necessary behavior, where every input pixel is weighted by different values and spread out by a specific radius, we use image vari-

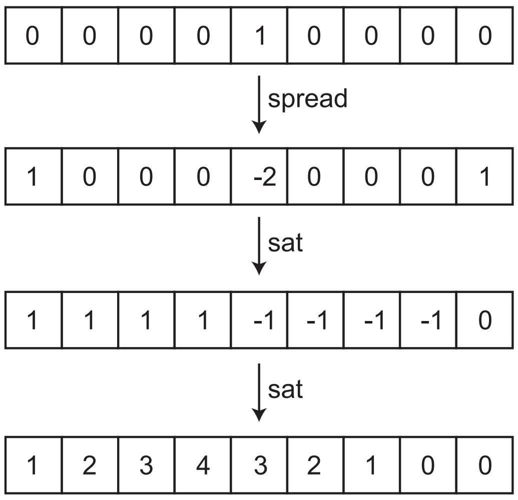


Figure 3.13: Polynomial spreading with a radius of four in 1D (from top to bottom): input image, image after spreading, image after first summed-area-table generation pass,image after second summed-area-table generation pass. Note that to obtain the final output image an additional normalization step is required.

ables, a feature introduced to the core of OpenGL and GLSL version 4.2, which allows storing and reading from a single texture level from any shader stage. For this, the GLSL datatype `image2D` and the corresponding functions `imageLoad` and `imageStore` where introduced in this version. The OpenGL function `glBindImageTexture` was established for binding a texture to an image2D variable. As mentioned above, to extend the process to two dimensions, the outer product of the one-dimensional weights is taken, resulting in the nine values shown in Equation 3.17.

$$\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix} \tag{3.17}$$

The first part of the method, the spreading of those nine values, is split into 9 passes, where each pass spreads exactly one value in one direction. By doing so we bypass the non-existing synchronization mechanisms in GLSL, because it is not possible that we access the same position twice in one pass when spreading with the same distance in the same direction at each pixel. However, we lose the possibility to spread with different radii at different positions, because that would again bring up major synchronization issues if the same position in the texture is accessed by different fragments. By spreading with different radii at different positions one could achieve depth-of-field simulation by spreading with a radius according to a fragments depth value without the need of splitting the image up into layers. The accumulation in one texture is done by reading out the destination position with `imageLoad` and storing the new sum with `imageStore`. The discard function is used to eliminate the requirement of writing an output variable in the fragment shader.

The second part of the method is the integration step, which has to be done twice as can be seen in Figure 3.12. The summed-area-table generation was implemented as proposed by Hensley et al. [9]. The algorithm splits the generation into multiple horizontal and vertical passes as explained in Section 2.1. The number of texture lookups executed per pixel is a parameter of the methods and determines how many passes are needed (see Algorithm 2.1).

The method concludes with a normalization pass, where each pixel is divided by radius to the power of four to obtain the final output. While implementing this method, we came across an issue with the border treatment while spreading the values. When traversing pixels which are closer to the image border than the spreading radius, the texture coordinates for the spreading destination could lie outside of the range $[0, 1]$ (depending on the spreading direction and the location of the pixel) and therefore outside of the image. Obviously, these values can not be written and therefore these missing values lead to an undesired result of the summed-area-table generation process, making the result useless. To overcome this problem, we simply shrink the rasterized rectangle, resulting in a border of exactly the spreading radius where no values are spread.

CUDA

The first part, the spreading of the nine values, is basically done in the same fashion as in GLSL by starting nine kernels in serial which spread one value each. The double integration by generating a summed-area-table twice utilizes the cudPP function `cudppMultiScan`. This function performs the parallel prefix sum algorithm on a given array. When using cudPP for the generation of the summed-area-tables, the image has to be split into one array per channel, because cudPP is not able to handle `float4` arrays. The splitting is done by a simple kernel which deinterleaves the RGBA texture into four arrays. After the summed-area-table generation is finished, again a simple kernel is executed to interleave these four arrays into an RGBA texture to yield the final result. The image-border issue described in Section 3.6.3 is again addressed by simply skipping problematic border pixels. The difference between the CUDA implementation and its GLSL counterpart is mainly the generation of the summed-area-table. The spreading step is done similar in both cases, with GLSL using image variables and CUDA using surface variables. The summed-area-table generation in CUDA is done using cudPP, which basically performs the algorithm presented by Hensley et al. [9] (see Section 2.1) with the parameter (number of texture lookups) being 1. This algorithm can be highly optimized for the usage in CUDA by using shared memory as described in GPU Gems Vol. 3 Chapter Parallel Prefix Sum (Scan) with CUDA [8].

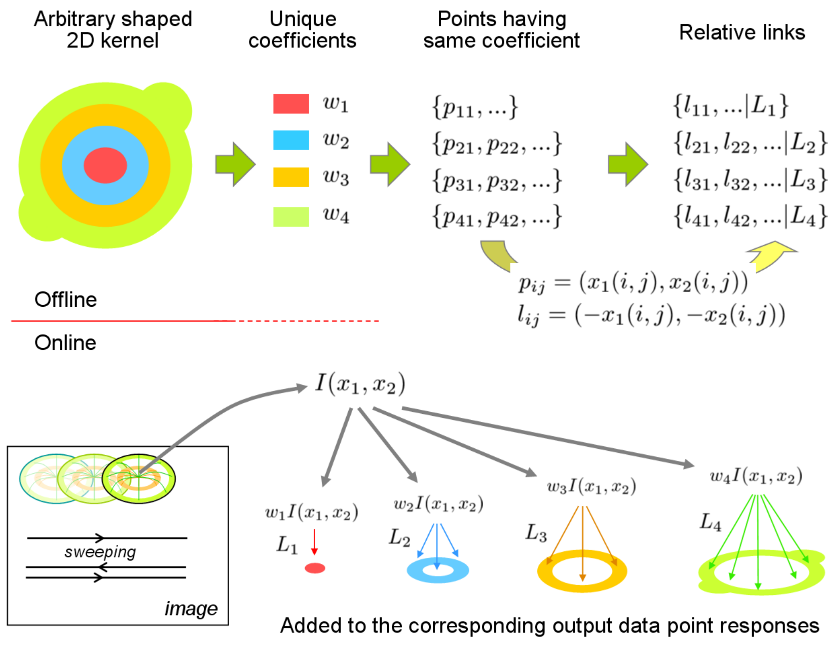
3.7 Reshuffling

3.7.1 Theory

Reshuffling was proposed by Porikli [24]. The basic idea is to exploit the fact that filter kernels often contain identical weights at different positions. The method starts by preprocessing a given filter kernel, yielding a list of unique weights and what Porikli calls a linkage set. The list of unique weights simply contains the weights which are unique across the whole filter kernel. For each entry in the unique weight list, a linkage set is constructed, containing the relative links of the filter positions with equal coefficients (as depicted in Figure 3.14). Both the unique weight list and the linkage set for each weight are calculated once per filter kernel in an offline preprocessing step. Based on those two components, Porikli presented two different algorithms for different platforms. The single input access algorithm is, according to Porikli [24], suitable for streaming processors where the local cache size is limited. The algorithm moves the filter kernel over the entire image, as it is done when applying convolution. Instead of gathering the values covered by the kernel at its center, a different approach is used. The pixel currently covered by the kernels center is multiplied by each of the unique weights and added to the points inside the filter kernel using the linkage sets. This means that instead of gathering values at the center of the filter kernel, the center value is spread out to the other locations covered by the kernel. Therefore, the algorithm only needs a single access to every pixel in the input image, whereas pixels in the output textures are accessed multiple times. The principle of this algorithm is schematically illustrated in Figure 3.14 and pseudo-code is given in Algorithm 3.1.

```
input  : List of unique weights $w_i$
input  : relative links per unique weight $l_{ij}$
input  : Input image $I$
output: filtered image $y$
for all $1 \leq x_1 \leq N_1$ do
    for all $1 \leq x_2 \leq N_2$ do
        $multiplier \leftarrow I(x_1, x_2)$
        for all $1 \leq i \leq U$ do
            $value \leftarrow multiplier * w_i$
            for all $l_{ij}$ do
                $k_1 \leftarrow k_1 + l_{ij}$
                $k_2 \leftarrow k_2 + l_{ij}$
                $y(k_1, k_2) \leftarrow y(k_1, k_2) + value$
            end
        end
    end
end
```

Algorithm 3.1: Reshuffling - single input access



Arbitrary shaped 2D kernel
Unique coefficients
Points having same coefficient
Relative links
w_1
w_2
w_3
w_4
$\{p_{11}, ...\}$
$\{p_{21}, p_{22}, ...\}$
$\{p_{31}, p_{32}, ...\}$
$\{p_{41}, p_{42}, ...\}$
$\{l_{11}, ...|L_1\}$
$\{l_{21}, l_{22}, ...|L_2\}$
$\{l_{31}, l_{32}, ...|L_3\}$
$\{l_{41}, l_{42}, ...|L_4\}$
$p_{ij} = (x_1(i,j), x_2(i,j))$
$l_{ij} = (-x_1(i,j), -x_2(i,j))$
Offline
Online
$I(x_1, x_2)$
sweeping
image
$w_1 I(x_1, x_2)$
$w_2 I(x_1, x_2)$
$w_3 I(x_1, x_2)$
$w_4 I(x_1, x_2)$
L_1
L_2
L_3
L_4
Added to the corresponding output data point responses

Figure 3.14: Schematic illustration of the construction of the unique weight list and the linkage set (top) and the single input algorithm (bottom) proposed by Porikli [24]. (figure from [24])

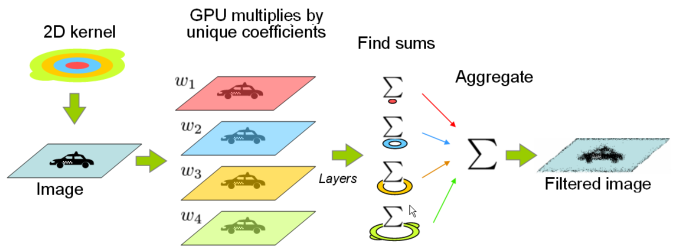


Figure 3.15: Schematic illustration of the single output algorithm proposed by Porikli [24]. (figure from [24])

The second algorithm is based on a single output access per pixel and is hence suitable for a GPU implementation. When applying conventional convolution, all pixels covered by the filter kernel are multiplied by the associated weight and the sum of these weighted values is added to the center location of the kernel. In contrast, the single output access algorithm first calculates the sum of all pixels in each linkage set (note there is one linkage set per unique weight). Afterwards, each sum is multiplied by the corresponding weight and the weighted sums are accumulated at the center of the kernel, to form the final output pixel. The principle of the single output algorithm is schematically illustrated in Figure 3.15 and pseudo-code is given in Algorithm 3.2.

```
input  : List of unique weights $w_i$
input  : relative links per unique weight $l_{ij}$
input  : Input image $I$
output: filtered image $y$
for all $1 \leq x_1 \leq N_1$ do
    for all $1 \leq x_2 \leq N_2$ do
        for all $1 \leq i \leq U$ do
            $sum \leftarrow 0$
            for all $l_{ij}$ do
                $k_1 \leftarrow k_1 + l_{ij}$
                $k_2 \leftarrow k_2 + l_{ij}$
                $sum \leftarrow sum + I(k_1, k_2)$
            end
            $y(k_1, k_2) \leftarrow y(k_1, k_2) + w_i * sum$
        end
    end
end
```

Algorithm 3.2: Reshuffling - single output access

3.7.2 Implementation

GLSL

Out of the two proposed algorithms by Porikli [24], it was only possible to implement the single output access algorithm in GLSL. This is due to the non-existing synchronization possibilities in GLSL, which would be needed to implement the single input access algorithm. The needed index list and weight list is built offline and sent to the fragment shader using uniform buffer objects. The weight list in this case consists of 2D float vectors, unlike the weights used for convolution. A vector contains the weight in the first component and how often this weight is present in the whole filter kernel in the second component. When the fragment shader has access to those two lists, the implementation is straightforward and done as shown in Algorithm 3.2.

CUDA

It was only possible to implement the single output access algorithm proposed by Porikli [24] because we are only able to synchronize the shared memory inside one thread block. However, the data blocks loaded into shared memory by the thread blocks overlap at the halo values. Therefore, those values would be written into the texture by multiple thread blocks when applying the single input access algorithm. There is no possibility to synchronize these writes to the output texture, which makes the algorithm unsuitable for the implementation on a GPU. The implementation of the single output access algorithm basically equal to the GLSL version. The same structure is used for index and weight list, which are copied to constant memory.

3.8 Repeated box filtering

3.8.1 Theory

This technique approximates the convolution with a Gaussian filter kernel by repeated average filtering. As shown in Figure 3.16, the impulse response achieved by recursively applying a box filter is an approximation of the result obtained by convolution with a Gaussian kernel. The notation used for the box filters in Figure 3.16 indicates how often a 3x3 box filter is applied to obtain the shown result. For example, H^{*2}_{box} denotes that the filter was applied two times. In his work [14], Peter Kovesi describes how to determine the averaging filters needed to achieve an approximation of a Gaussian with a specific standard deviation σ. He describes a process which calculates the width of the required average filters based on a specific standard derivation and the number of passes. When using summed-area-tables, box filtering with an arbitrary width has constant time complexity in terms of filter width, as shown in Figure 2.1 and Equation 2.3. The summed-area-table generation, as shown by Hensley et al. [9], can be done in $O(n * log(n)))$ and the actual calculation of the sums using the summed-area-table is in $O(4 * n)$. As can be seen, the process only depends on the number of input pixels and is therefore especially suited if a very large blur radius is needed. Another way to implement average filtering is separated convolution, which is in $O((m + m) * n)$ (with m denoting the filter width). According to our tests, separated convolution should be chosen over using summed-area-tables for average filtering with small blur radii (see Section 4.2.9). Overall the process of repeated box filtering is similar to pyramidal blurring but without the down and up-sampling but instead with a bigger filter width (e.g. 15x15 box filter instead of 4x4).

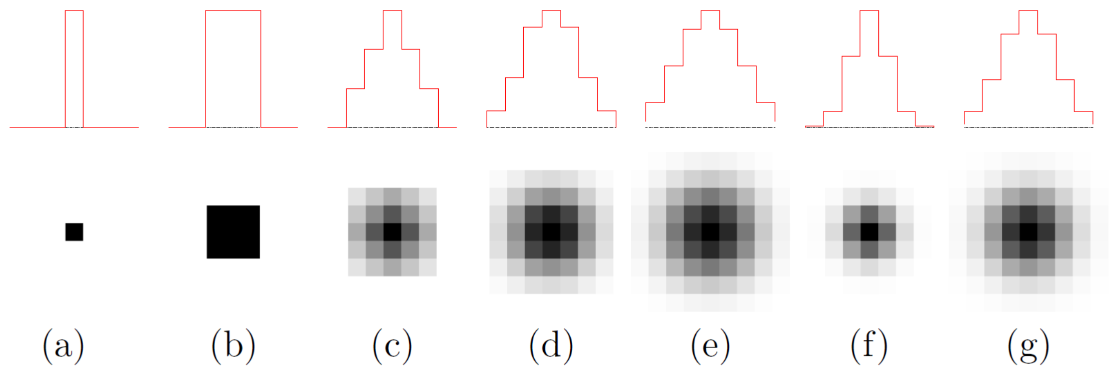


Figure 3.16: "The original signal is a 9x9 pixel image with an impulse at (4,4) (a). The signal convolved with recursive box filtering of the type H_{box}, H^{*2}_{box}, H^{*3}_{box} and H^{*4}_{box} (b-c), respectively. For comparison, the convolutions of (a) with Gauss filters of sizes $\sigma = 1.0$ and $\sigma = 1.5$ are shown in (f) and (g). Note that all plots and images are normalized so that the maximum value is 1 for a better visualization. The plots show the 4th column of the convolved images." [25] (figure from [25])

3.8.2 Implementation

GLSL

Repeated box filtering has been implemented in three different ways, all yielding the same result. The first two variations are to implement box filtering by convolution and separated convolution. The third is to create a summed-area-table of the input texture and carry out box filtering using Equation 2.3. Note that when repeatedly applied, this version requires a summed-area-table generation after each pass.

CUDA

In CUDA, the same three variations of repeated box filtering as in GLSL where implemented.

3.9 Simulated Heat Diffusion

3.9.1 Theory

This method is modeled after the physical heat equation, which describes the diffusion process of a physical property (e.g. temperature) in a homogeneous 3D volume until an equilibrium is reached. If function $f(x,t)$ specifies the physical property at a given position in the volume $x = (x,y,z)$, the heat equation can be written as seen in Equation 3.18. The constant c represents the conductivity of the material the 3D volume is made of.

$$\frac{\partial f}{\partial t} = c \cdot [\nabla^2 f] = c \cdot \left[\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2}\right] \tag{3.18}$$

If we see an image as continuous and time-varying function $I(x,t)$, in two-dimensional space, the same formula can be applied.

$$\frac{\partial I}{\partial t} = c \cdot [\nabla^2 I] = c \cdot \left[\frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}\right] \tag{3.19}$$

Furthermore, because we deal with discrete image data, the partial derivatives can be approximated through finite differencing. For this reason, the case of second derivatives can be done by convolution with a Laplacian filter kernel. This leads to the following approximation of the heat equation for images, where $*$ denotes the convolution operator:

$$\frac{\partial I}{\partial t} = c \cdot [I * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}] \tag{3.20}$$

The method achieves image smoothing by applying this formula in a recursive fashion, as shown in the following equation with $I^{(0)} = I$:

$$I^{(t)} = c \cdot I^{(t-1)} * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{3.21}$$

This can be done in a multipass fashion, with the parameter c, which has to be in the range of $(0, 0.25]$ for numeric stability reasons, controlling the speed of the smoothing process. Note that this isotropic diffusion process approximates the convolution with a Gaussian kernel, where the width of the filter can be controlled by the number of iterations applied. This approximation is of very good quality, as shown in Section 4.1.2 by our quality tests using hdr-vdp2 [19]. Another advantage of this approximation is the high control over the applied filter size by applying a small filter (5 texture lookups) iteratively. Furthermore, the process can be implemented in an anisotropic fashion by for example making the conductivity parameter c a function of the local gradient. By doing so, the blurring could be done in an edge preserving manner by blurring homogenous areas and at the same time keeping sharp edges. As shown in Section 4.1.2, a disadvantage of this method is the high number of iterations needed to achieve a big blur radius. Therefore, the performance of this technique when in the need of a high effective filter size is worse compared to most blurring algorithms.

3.9.2 Implementation

GLSL

The diffusion process was implemented by convolution with a Laplacian filter kernel, as shown in the diffusion Equation 3.21, with the c value and the number of iterations as parameter. The iterations are applied by using Ping-Pong rendering, switching between two textures as input and output, as it is also used for the summed-area-table generation in the method proposed by Hensley et al. [9].

CUDA

In CUDA, diffusion was implemented in two different ways. The first variation was realized similar to the GLSL implementation by using texture lookups and hard-coding the diffusion formula. The second one was implemented similar to convolution by using shared memory. For this, all threads in a thread block load their main data (texture data corresponding to this thread) into shared memory. Additionally, all threads at the border of a thread block load the necessary halo data of one pixel around the thread block. The shared memory is again split into four float arrays instead of using a big float4 array to avoid bank conflicts. Ping-Pong rendering like in the GLSL implementation is used to apply the method iteratively.

CHAPTER 4

Results

This chapter presents the results of the performance and quality tests applied to the chosen methods. The chapter starts by describing the test methodology for these tests, followed by a comparison of CUDA and GLSL when used with the selected methods. In Section 4.4, an overall comparison of all algorithms in terms of performance and quality is given. The chapter concludes by showcasing the most suitable technique, namely quasi-convolution, when used inside a layered depth-of-field framework.

4.1 Test methodology

4.1.1 Performance

All the timings for the various plots in the following sections were taken on a 3.3 GHZ Intel i5-2500k and an NVIDIA GeForce 460 GT. For the time measurements, the timer function implemented in GLFW was used, which allows for double precision timings. To account for any slowdowns during startup, the time measurement starts five seconds after the program was started. For all methods, the timings include the setup needed to carry out the algorithm, for example the switching of framebuffers and textures in GLSL or the mapping of textures to `cudaArrays` in CUDA. These timings are taken for five seconds periodically, with an interval of 200ms, resulting in 25 measurements, from which the mean is taken as representative for the plots. Every method was tested with different filter sizes (possible filter sizes depend on the actual method) with an image with a resolution of 1024×1024. Additionally, the performance of all methods when varying the input resolution, while keeping the filter size constant, has been measured.

4.1.2 Quality

The quality tests for the approximation methods where done in two steps. In the first step, the effective filter size of a given method has to be determined. For this task, an automated process using Matlab has been developed and applied. As reference for this effective filter size, the standard deviation sigma (σ) of a Gaussian filter kernel has been chosen. We tested absolute pixel difference (absolute difference per channel, summed up over all channels and pixels) and P_{mean} and Q_{MOS} for this matter. Absolute pixel difference showed to be the most distinctive metric (clearest minimum). Therefore, the absolute pixel difference was chosen as primary metric for this comparison. Additionally, P_{mean} and Q_{MOS} calculated by hdr-vdp2 [19] are provided for perceptual comparison. The process was implemented using Matlab and a set of 53 test images, 48 of those images where taken from flickr, showing different scenarios (e.g. urban, forest, mountains). The remaining 5 images are synthetic images, containing lines and filled areas (for example a circle like can be seen in Figure 4.1). The Matlab script takes the original images and the images blurred with one of our chosen methods as input. Furthermore, it applies a Gaussian convolution filter, with a range of different sigmas, to the original images. The absolute pixel difference is then calculated between these images and the images blurred with the method in question (e.g. quasi-convolution when using only one level of the pyramid). For each image, the sigma corresponding to the minimum absolute difference is then taken as best-fitting sigma. This process results in 53 sigma values, of which the mean, median and standard deviation is calculated. Furthermore, box plots, which are presented in Section 4.3, where created.

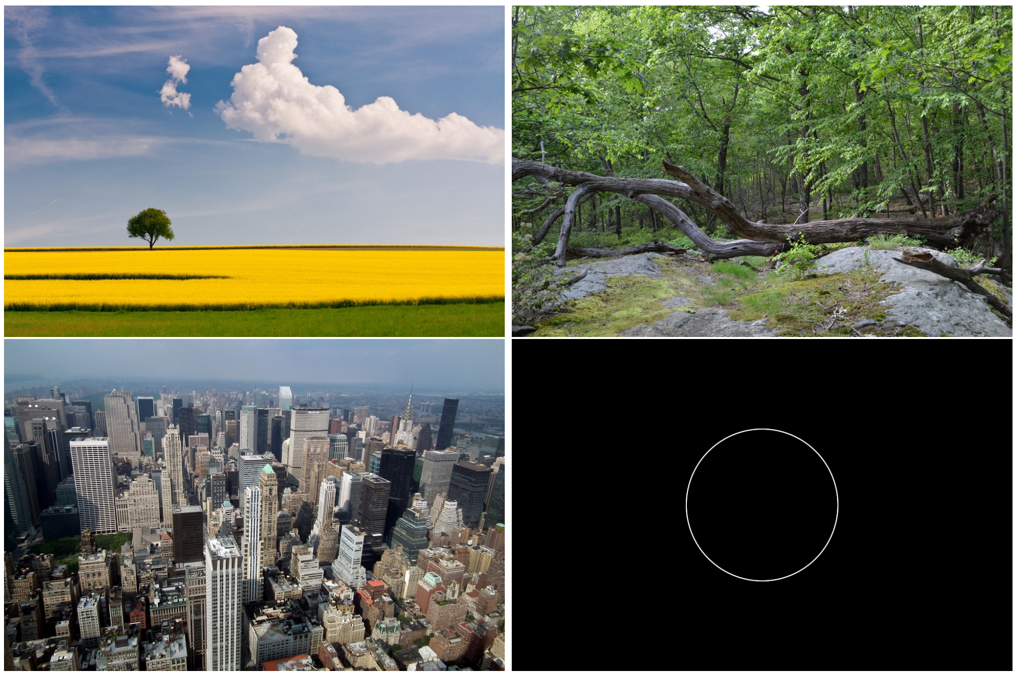


Figure 4.1: Examples of the images used for the quality comparison.

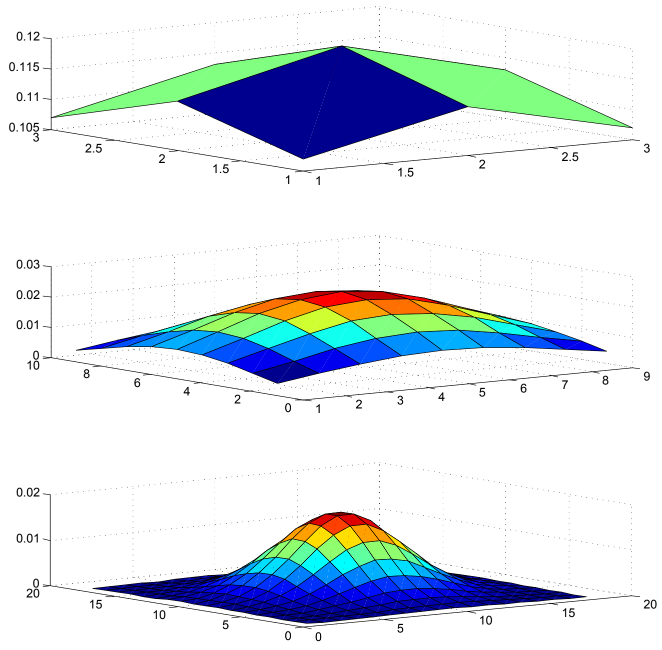


Figure 4.2: Normalized Gaussian filter kernels with $\sigma = 3$ and filter widths of 3, 9 and 18 from top to bottom.

In the second test, the determined sigma per method and parameter is used to compare the quality of the methods against the ground-truth using hdr-vdp2 [19]. The ground truth images for this test where created using convolution. For every image in the test set, the probability of detection (P_{mean}) as well as the quality mean-opinion score (Q_{MOS}) is calculated. This results in a vector of 53 values for each metric, of which again mean, median and standard deviation are calculated.

The filter width for the previously mentioned convolution filter was chosen to be $6 \times \sigma$. This choice was made to preserve the Gaussian shape of the kernel by making the window large enough. In Figure 4.2, filter kernels with equal sigma but different window sizes are presented to illustrate the effect of the filter width on the shape of the kernel.

4.2 Performance

In this section, the methods are examined in terms of their behavior when used with different parameters. Furthermore, the differences between the CUDA and the GLSL versions of the algorithms are discussed. Note that the range of the y-axis, in the presented plots, is different in each scenario. For an overall comparison of the methods see Section 4.4.

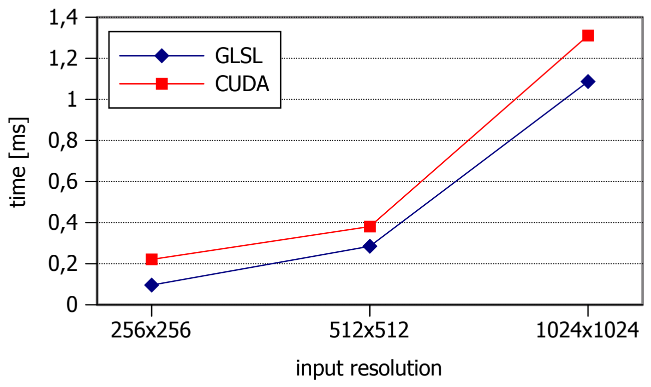


Figure 4.3: Timings when *copying* a texture with varying input resolutions.

4.2.1 Copy

In most of the tested algorithms, the performance of texture lookups has a major impact on the overall performance of the algorithm. To compare GLSL and CUDA in terms of texture lookups, a simple copy operation was implemented. The copy operation includes exactly one texture lookup and one texture write per input pixel. Figure 4.3 shows the timings for this operation when using different input resolutions. It can be seen that the copy operation in CUDA is slower than its GLSL counterpart by a nearly constant offset. This indicates a constant overhead which is slightly lower at a resolution of 512x512. We also measured the setup time needed for the copy operation. For GLSL, this includes the binding of a framebuffer, one vertex array, one texture, and the binding of the shader. In case of CUDA, the setup consists of mapping the input and output texture to `cudaArrays`. For an input texture of size 1024×1024, the process took 0.43ms for GLSL and 0.56ms for CUDA.

4.2.2 Convolution

Figure 4.4 shows the performance of conventional convolution when varying the filter width for both the GLSL and CUDA implementation. As expected, the time needed to convolve the image increases steadily with the used filter width. To illustrate the quadratic trend of the curve, the values shown in Figure 4.4 are calculated by taking the square root of the corresponding timings. As can bee seen, the points form an almost straight line (for both implemenations), confirming that the trend is indeed quadratic. The missing values in the CUDA curve for a filter width bigger than 23 are because of the shared memory limitation of CUDA, which does not allow us to apply filters beyond that width. For devices with compute capability 2.x and higher, the maximum size of shared memory per multiprocessor is limited to 48KByte. When using a block size of 32x32 (note that the block size must be bigger than the filter size in our implementation) and a filter size of 25x25, we need a shared memory array of 57x57.

The total size of this array, containing 4 floats per entry, is 52KByte, which exceeds the maximum permitted size for shared memory per multiprocessor. Overall, it can be seen that

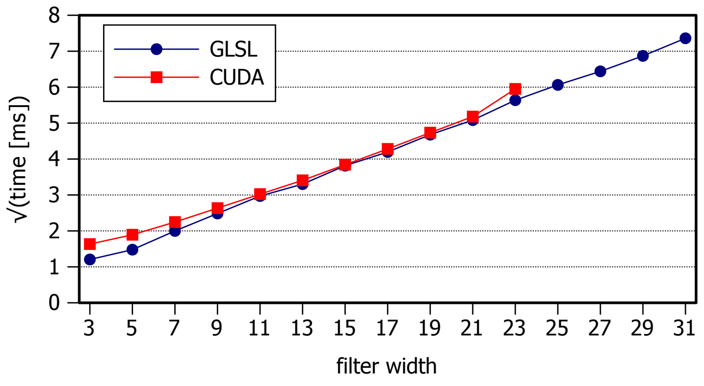


Figure 4.4: Timings of *two dimensional convolution* with varying filter widths (note that the values on the y-axis are the square roots of the measured timings).

when using two-dimensional convolution, GLSL has an edge over CUDA in terms of performance. This is partly due to the fact that when applying two-dimensional convolution, every thread loads a big junk of shared memory. Therefore, in contrast to separated convolution (see Section 4.2.3), bank conflicts can not be avoided by deinterleaving the image in this scenario. Another factor contributing to this outcome is the overhead resulting from the shared memory setup and the automatic texture caching done by GLSL which makes it so that CUDA and its shared memory has no advantages in a scenario where texture lookups are as coherent as when applying convolution.

In Figure 4.5, the timings for convolution when varying the input resolution can be seen. Note that for every step on the x-axis, the number of pixels in the input image is quadrupled. The slow down factor when going from a resolution of 512x512 to 1024x1024 is 4.18 for GLSL and 3.57 for CUDA.

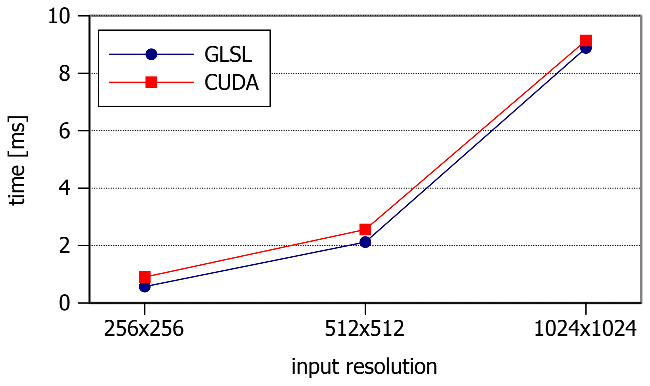


Figure 4.5: Timings of *two-dimensional convolution* with varying input resolutions for a filter width of 11.

As mentioned in Section 3.2, we compared two variations of passing the filter data to the fragment shader when using GLSL. When using texture buffer objects, the lookups of the filter data put additional load on the GPU's texture units. Figure 4.6 shows that this additional load has major impact on the performance of the process. The timings increase with more than double the rate compared to using uniform buffer objects.

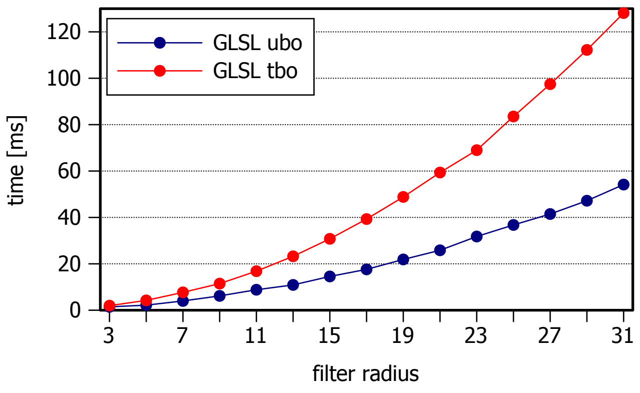


Figure 4.6: Timings of *two-dimensional convolution* when using texture buffer objects (tbo) and uniform buffer object (ubo) for the filter data.

4.2.3 Separated convolution

For separated convolution, in contrast to two-dimensional convolution, the number of texture lookups needed per pixel increases linearly with the filter width. The timings for separated convolution are presented in Figure 4.7. It can be seen that the timings of the GLSL implementation increase at a steeper slope compared to the CUDA implementation. Furthermore, the CUDA implementation is in general faster than its GLSL counterpart. This can be explained by the usage of shared memory in CUDA. As explained in Section 3.3, every thread loads only two pixel values from texture memory, whereas the rest of the process uses lookups from shared memory. This also explains the fact that for a filter width of 3, the CUDA implementation is slightly slower, which is because of the overhead originating from the setup of shared memory (see Section 3.3). In this scenario, this overhead over weighs out the savings of using shared memory instead of texture memory.

As mentioned in Section 3.3, the CUDA implementation was tested using an interleaved and a deinterleaved shared memory data structure for the sake of avoiding bank conflicts. In the case of interleaved data, the shared memory is constructed as two-dimensional `float4` array, whereas for deinterleaved data, there is one two-dimensional float array per channel. We found that interleaved data suffers from bank conflicts, which have a severe impact on the performance of the kernel. This is also reflected by the timings shown in Figure 4.7. The number of bank conflicts per access, in case of interleaved data, depends on the halo needed for the shared memory data structure and therefore the size of the filter. This explains the bumpiness of the curve. The lower amount of shared memory used by each thread block and the bank-conflict free access pattern makes the CUDA implementation, in contrast to two-dimensional convolution, the better choice for separated convolution.

In Figure 4.8, the timings for separated convolution when varying the input resolution can be seen. The slowdown factor when going from a resolution of 512x512 to 1024x1024 is 2.22 for GLSL and 2.46 for CUDA.

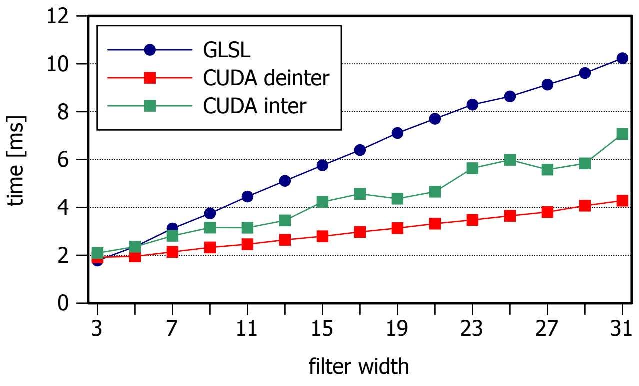


Figure 4.7: Timings of *separated convolution* with varying filter widths.

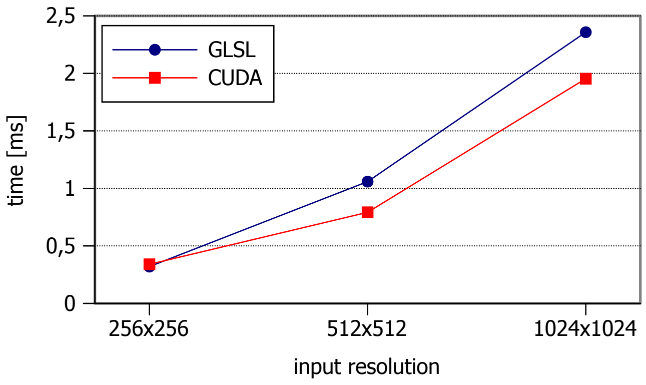


Figure 4.8: Timings of *separated convolution* with varying input resolutions for a filter width of 11.

4.2.4 Pyramid methods

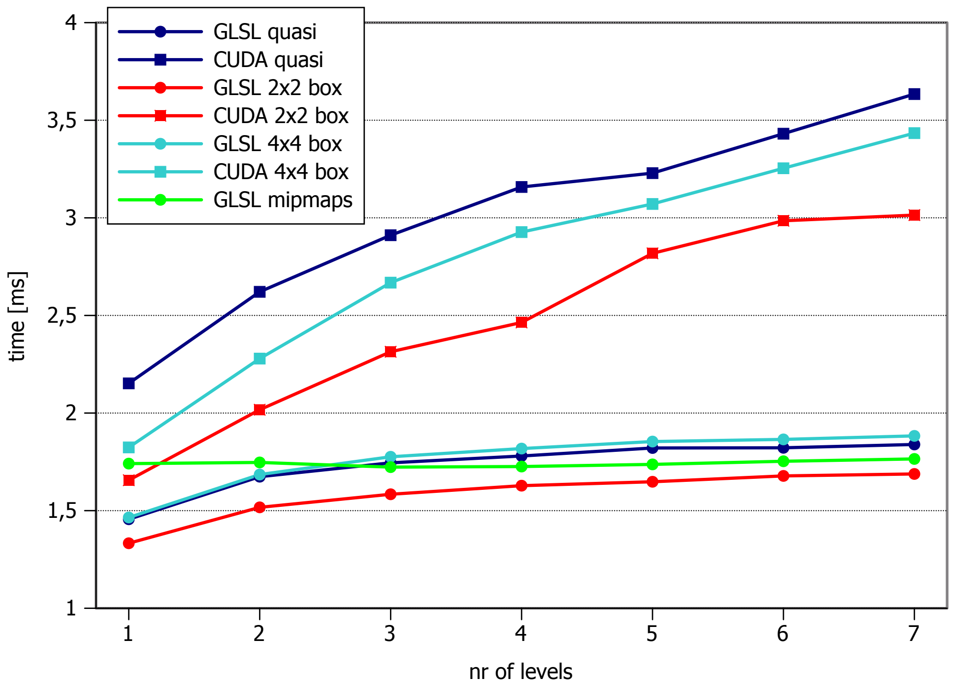


Figure 4.9: Timings of the various down-sampling methods, namely *quasi-convolution*, *2x2 box*, *4x4 box* and *mipmaps*, with varying number of levels.

As explained in Section 3.5, we implemented three different types of analysis filter in both GLSL and CUDA, and additionally in GLSL down-sampling with the built-in mipmap function from OpenGL. The timings for all those down-sampling techniques can be seen in Figure 4.9, for an input texture with a resolution of 1024×1024, when up-sampling from different levels. These are the timings needed to blur the input image using the given number of pyramid levels. Therefore, the values on the x-axis represent the number of down-sampling steps as well as up-sampling steps. For the down-sampling using mipmaps, the number of down-sampling steps is always dependent on the input resolution, because in this case the full pyramid down to a resolution of 1×1 is created. Overall, we can see that for all variations, the CUDA implementations are slower than their GLSL complements. This is expected when looking at the timings presented in Section 4.2.1 for the copy operations and its setup. When comparing the timings of the different analysis filters versus each other, we can see that the for the CUDA implementations, the ranking is in the same order as the number of bilinear texture lookups per pixel. The 2×2 box filter needs one bilinear lookup, whereas the 4×4 box filter needs two and the quasi-convolution filter needs four bilinear lookups per pixel and per level. In case of GLSL, we observe that the 4×4 box filter is slightly slower than the quasi-convolution filter, which is because the 4×4 box filter is applied in a two pass process. The cost of the additional pass is higher than the cost of the two additional bilinear texture lookups needed for the quasi convolution filter. For the mipmap pyramid, created with OpenGL, the timings do not depend on the actual number of levels used, this is due the fact that OpenGL always creates the full mipmap pyramid.

4.2.5 Spreading

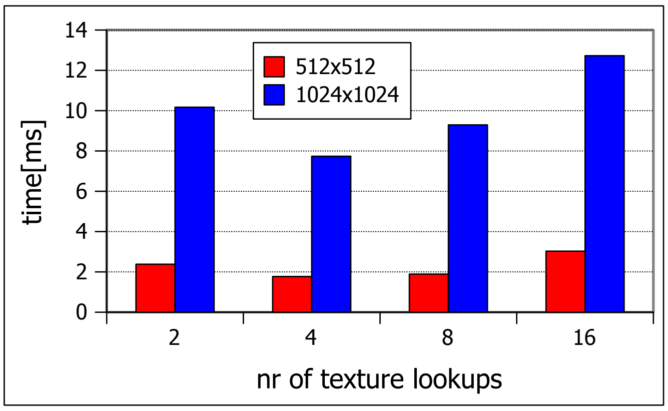


Figure 4.10: Timings for GLSL *summed-area-table* generation with varying number of texture lookups per pixel.

The first performance analysis for spreading was to determine the number of texture lookups per pixel and pass for the summed-area-table generation in GLSL. As mentioned in Section 3.6.3, the number of texture lookups is a parameter of the algorithm used for summed-area-table generation. It also determines how many passes are needed for the horizontal and the vertical steps of the algorithm. Therefore, the timings when this parameter is varied were measured and can be seen in Figure 4.10.

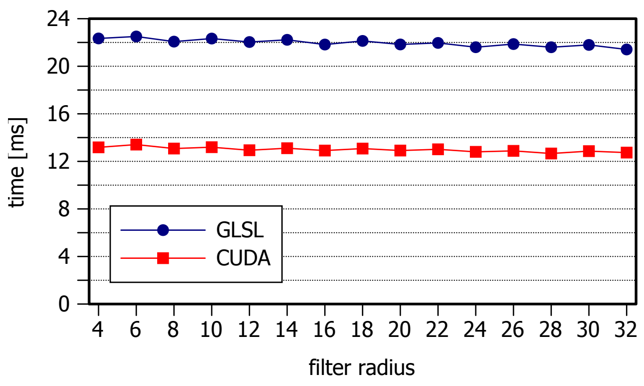


Figure 4.11: Timings of the *spreading* method with varying spreading radius.

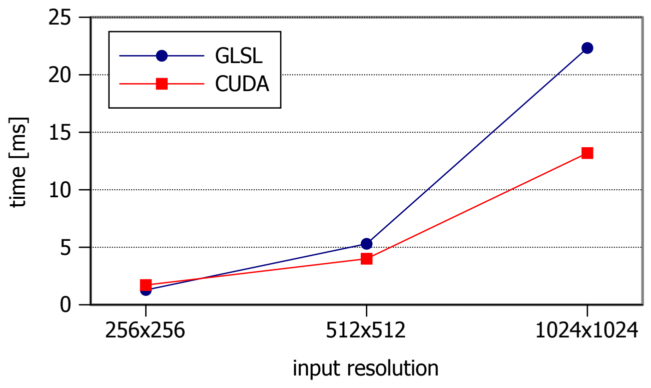


Figure 4.12: Timings of the *spreading* method with varying input resolution.

According to our measurements, four texture lookups per pixel and pass offer the best trade-off between the number of passes and texture lookups, yielding the best performance. Taking this into consideration, this parameter was kept at four for all future performance tests involving the generation of summed-area-tables in GLSL. Figure 4.11 shows the timings of the spreading process, including the double summed-area-table generation, for the CUDA and GLSL implementation. As expected, the method does not depend on the actual filter size, because the number of operations carried out per pixel does not depend on it. It can also be seen that the CUDA implementation is on average faster by a factor of 1.7. This is due to the fact that the highly optimized pre-fix sum algorithm, implemented in cudPP, is faster than the GLSL implementation of the algorithm proposed by Hensley et al. [9], despite the need for the additional deinterleave, interleave and transpose steps. Also the spreading process itself, using the image variables introduced in GLSL version 4.2, is slower than its CUDA counter part. This part of the method for GLSL takes on average 37% (8.1 ms) and for CUDA 53% (6.9 ms) of the whole filtering process. Figure 4.12 shows the timings for different input resolutions, these were recorded with a spreading radius of 10. The timings show that for very small images (e.g. 256x256), the GLSL implementation is actually slightly faster than the CUDA implementation. This can be explained by the need of the already mentioned additional steps for cudPP, these outweigh the savings of a faster summed-area-table generation for low image resolutions. The slowdown factor when going from a resolution of 512x512 to 1024x1024 is 4.21 for GLSL and 3.29 for CUDA.

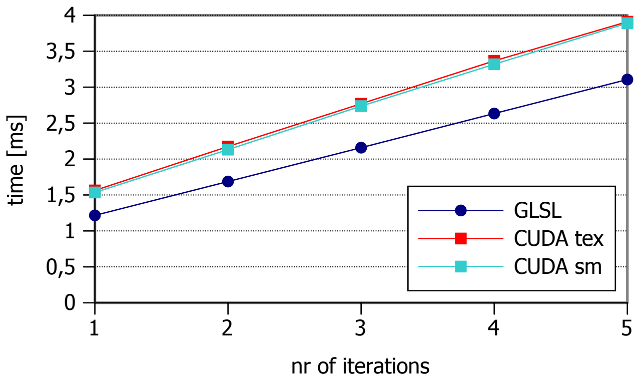


Figure 4.13: Timings for *simulated heat diffusion* with varying number of iterations (1-5).

4.2.6 Diffusion

In Figure 4.13, the timings for the first five iterations of the diffusion process are shown for the GLSL implementation and the two CUDA implementations mentioned in Section 3.9.2. The timings reveal that the initial pass for both CUDA implementations is slower than the initial pass in GLSL, but also that each additional pass is of higher cost in CUDA than it is in GLSL. The process is equal to two-dimensional convolution, without the need of weights and offsets, and therefore the same conclusions as in Section 4.2.2 can be made. In Figure 4.14, the timings in terms of number of iterations are presented on bigger scale. The figure illustrates the different slopes of the curves, where the GLSL implementation has the lowest slope followed by the CUDA shared-memory implementation. The timings for different input resolutions can be seen in Figure 4.15, which were recorded when running diffusion with ten iterations. The slowdown factor when going from a resolution of 512x512 to 1024x1024 is 4.14 for GLSL and 3.49 for CUDA.

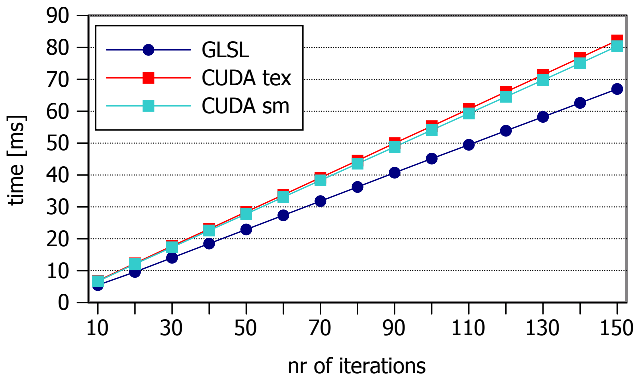


Figure 4.14: Timings for *simulated heat diffusion* with varying number of iterations (10-150).

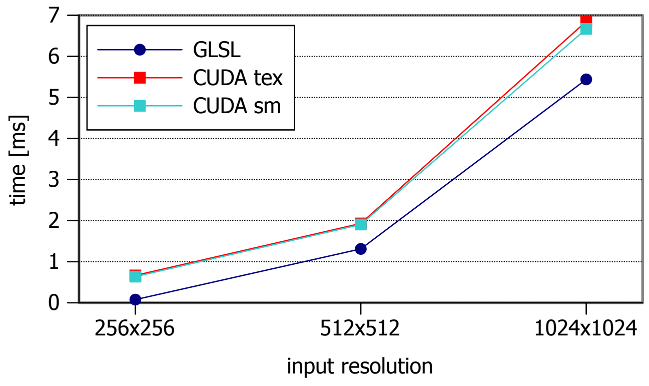


Figure 4.15: Timings for *simulated heat diffusion* with varying input resolution.

4.2.7 FFT

As mentioned in Section 3.4, image filtering using FFT has a constant time complexity with respect to filter size. Therefore, similar to repeated box filtering, the question arises at which point (filter size) the performance of this algorithm actually surpasses the performance of convolution. Figure 4.16 shows the timings of the FFT algorithm compared to the timings of direct convolution for CUDA and GLSL. It can be seen that filtering using FFT for a 1024x1024 image takes $10.2ms$, independent of the actual size of the filter. In the case of two-dimensional convolution, the cross-over points are at a filter width of 13 for both implementations. When using separated convolution, the respective curves cross at a width of 31 for GLSL and 81 for CUDA. Figure 4.17 presents the timings of the algorithm for different input resolutions. The slowdown factor when going from a resolution of 512x512 to 1024x1024 is 2.85.

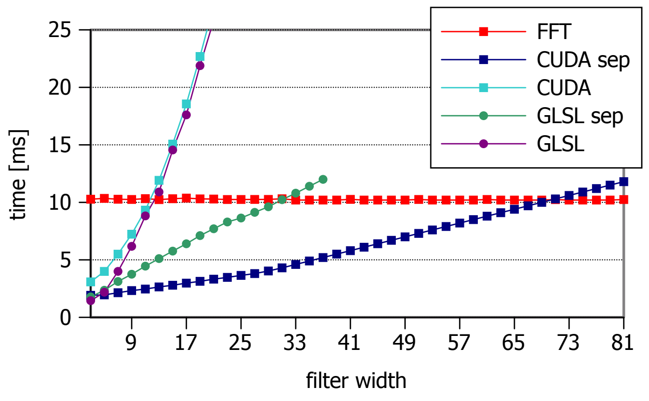


Figure 4.16: Timings for convolution using *FFT* in comparison to direct convolution (separated and two-dimensional) for various filter widths.

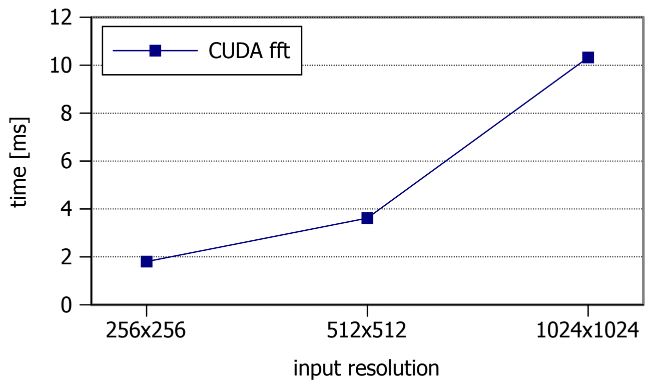


Figure 4.17: Timings for convolution using *FFT* with varying input resolution.

4.2.8 Reshuffling

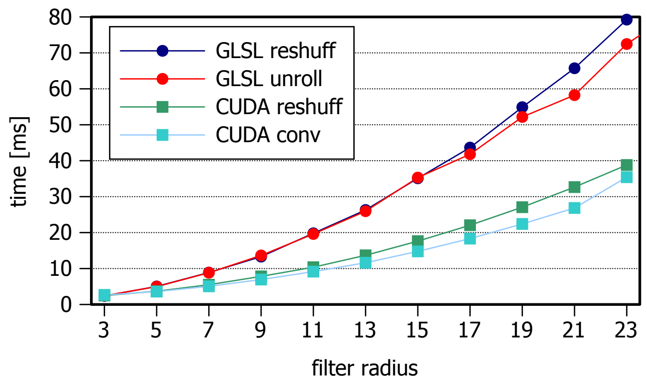
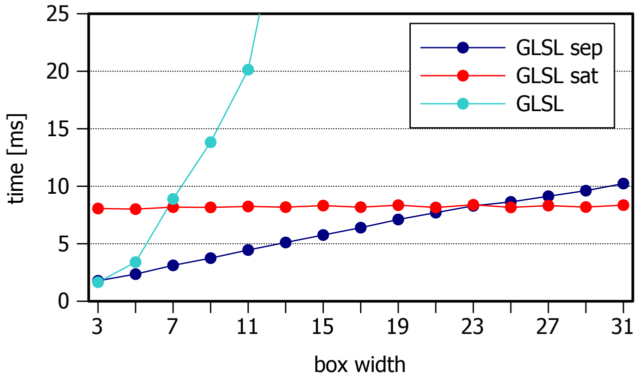


Figure 4.18: Timings for *reshuffling* with varying filter widths.

In theory, the reshuffling approach reduces the number of operations needed for the two-dimensional convolution by preprocessing the filter kernel. In our tests, we found that when implemented in GLSL, the approach has a drawback which has a major impact on performance. When realizing convolution the traditional way, as presented in 3.2, the filter size can be passed to the shader at compile-time via `#DEFINE`. By doing so, we enable the compiler to unroll the for-loops used to iterate over the pixel neighborhood covered by the filter kernel. Obviously, this is not possible when using the reshuffling single-output algorithm explained in 3.7. To examine the impact of loop unrolling on the speed of the convolution process, a version of convolution where loop unrolling is suppressed by passing the filtersize as uniform variable was implemented. The timings of this altered version, compared to the timings of reshuffling, are shown in Figure 4.18. The plot also shows the timings of reshuffling and convolution when implemented in CUDA (note that the curve for GLSL convolution is nearly the same as for CUDA and was therefore omitted in this plot). Due to these results, we can conclude that loop unrolling has a major impact on the performance when using a shading language to implement convolution. However, this is not true in the case of CUDA, where the plot shows a very narrow difference between reshuffling and the convolution kernel. Also, the costs of additional constant memory lookups (weightlist and offsetlist, see Section 3.7) outweigh the multiplication savings of the process.

4.2.9 Repeated box filtering

As mentioned in Section 3.8, with the help of summed-area-tables, box or average filtering can be realized using only four texture lookups per pixel. To determine the point where the savings of the reduced number of lookups outweigh the additional costs of the summed-area-table generation, the method was tested with different filter sizes when running one iteration of the process. In Figure 4.19, the results of this test are presented using separate plots for CUDA and GLSL to show the point where the generation of the summed-area-table pays off for these two separately.



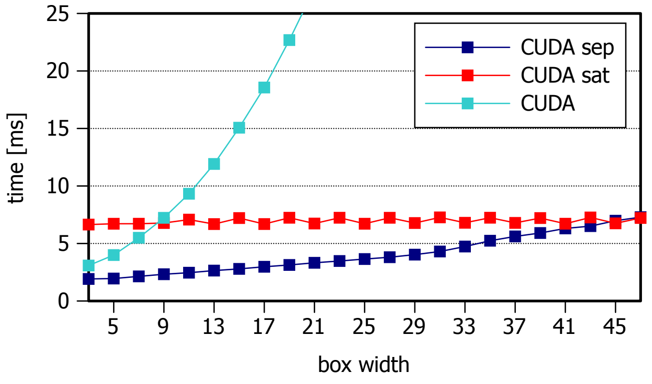


Figure 4.19: Timings for *box filtering using summed-area tables*, two-dimensional convolution and separated convolution with varying number of iterations for GLSL (top) and CUDA (bottom).

In the case of two-dimensional convolution, using summed-area tables is more efficient when using a box width of 7 or above for GLSL, whereas for CUDA, the same applies for a box width of 9 or above. For separated convolution, this point is reached at a box width of 45 for CUDA and at a box width of 25 for GLSL. Based on these results, separated convolution was chosen as preferred method when using repeated box filtering, because the repeated application of a small box filter is sufficient to achieve a high effective filter size (see Section 4.1.2). The timings of the method when using more than one iteration are shown in Figure 4.20. As presented in Section 4.2.3, the actual size of the filter has a bigger impact on the performance when using GLSL compared to CUDA. When analyzing the performance of the method with regard to the number of iterations, we can observe several facts. First, the curve for the CUDA implementation has a higher slope when going from one iteration to two than it has from two iterations to three. This is because of the additional intermediate texture needed for the repeated application of the filter, which increases the setup time. Furthermore, when looking at the whole curve, the slope is similar for CUDA and GLSL. For example when using a filter size of 9, the slowdown factor between one and three iterations is 2.6 for GLSL and 2.7 for CUDA. Overall, it can be said that like for separated convolution, CUDA is better suited for the implementation of repeated box filtering.

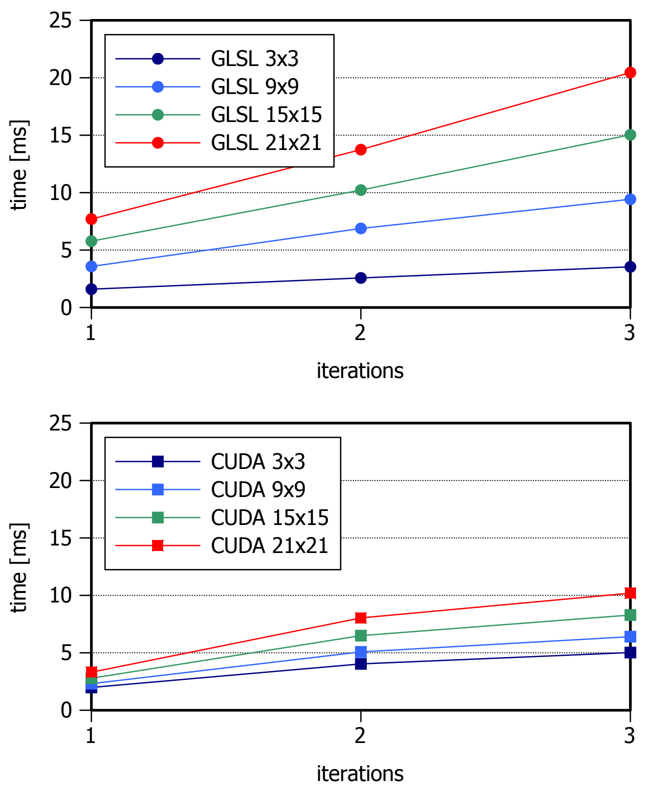


Figure 4.20: Timings for *box filtering* using separated convolution for different filter sizes with varying number of iterations for GLSL (top) and CUDA (bottom).

4.3 Quality

The following subsections present the results of the quality tests described in Section 4.1.2. The values of the best-fitting sigma estimation tests are presented in box plots, and the results of the quality tests are discussed. These box plots show the distribution of the determined sigma values over all images. The box starts at the 0.25-quantile and ends at the 0.75-quantile of the underlying data. The so-called whiskers outside of the box show the minimum or maximum value of the data if they do not coincide with one of the box edges. Inside the box, the median is marked by a line, which in most of our cases coincides with one edge of the box. Note that if there is very little variance, for example if all 53 values are equal, the box degenerates to a line with the whiskers marking outliers. We found that a low variance of the data obtained through absolute pixel difference correlates with the quality metrics provided by hdr-vdp2 [19].

Figure 4.21 shows the result of convolution with Gaussian filter kernels with different sigmas, to give a sense of how much blur is actually achieved by a kernel with a specific sigma.

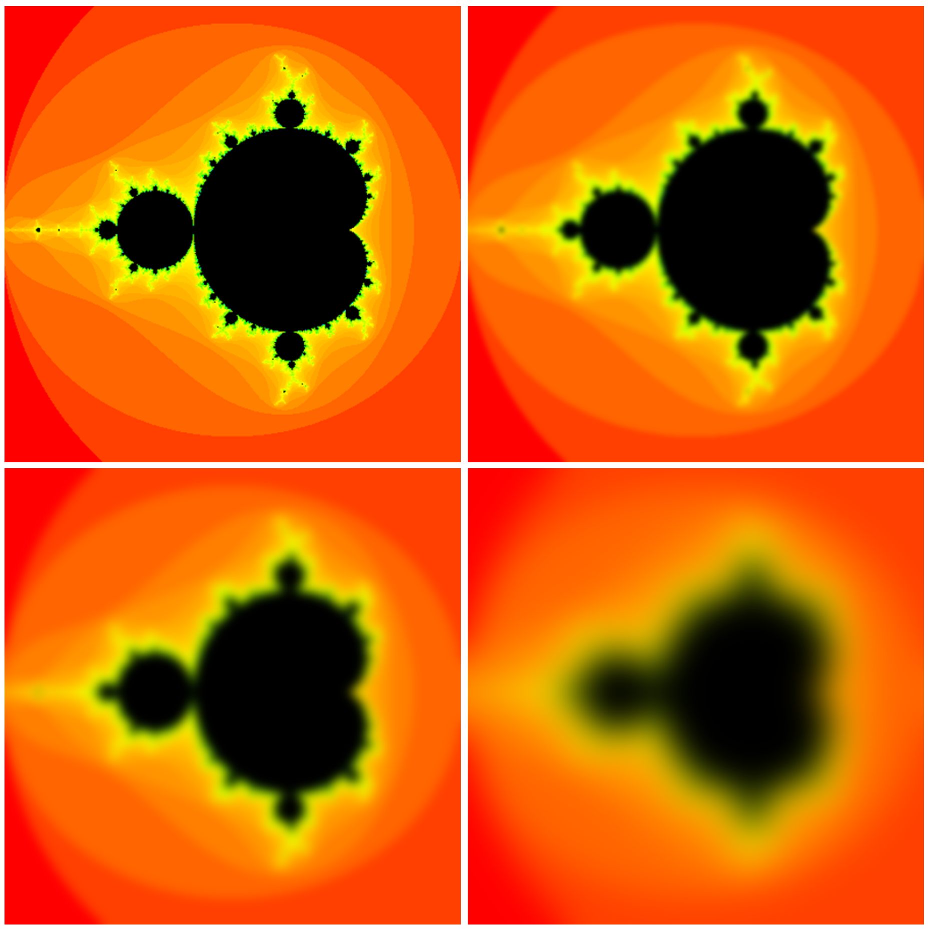


Figure 4.21: Example image and the results of the convolution with Gaussian filter kernels with different sigmas: sigma=3 (top-right), sigma=6 (bottom-left), sigma=20 (bottom-right).

4.3.1 Pyramid methods

When looking at the determined sigma values for pyramid methods, as shown in Figures 4.22-4.24, it can be seen that they achieve a high effective filter size when using multiple levels of the pyramid (e.g., sigma>20 when using 4 levels of the pyramid). The box plots also show that pyramid methods offer the least control over the effective filter size. Sigma roughly doubles per additional level used, which results in a low resolution of possible sigma values. Regarding the different down-sampling methods, quasi convolution and 4x4 box approximate roughly the same sigma values while the values for 2x2 box are slightly lower. The results of the quality measurements using hdr-vdp2 [19] are presented in Tables A.1- A.3. They show that quasi-convolution scores are the best out of the tested down-sampling methods, with the lowest average probability of detections amongst them (average of 5% over all cases). For 4x4 box down-sampling, the average probability of detection is slightly higher and for 2x2 box down-sampling the values are over 20% in four out of five test cases.

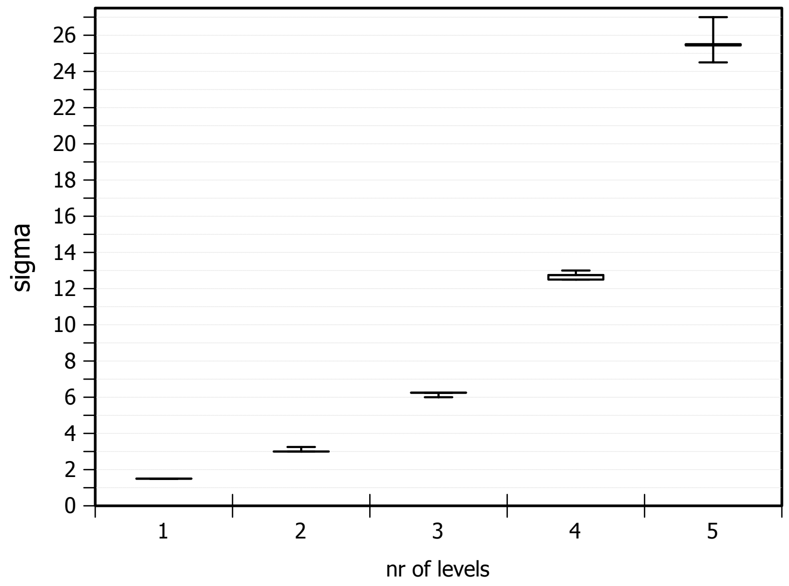


Figure 4.22: Box plots for the determined sigmas when using *quasi convolution down-sampling* with varying number of levels .

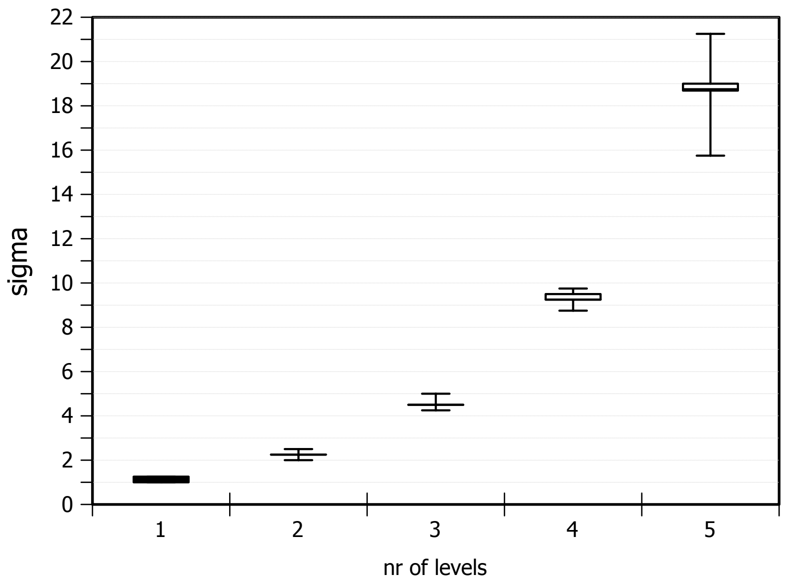


Figure 4.23: Box plots for the determined sigmas when using *2x2 box down-sampling* with varying number of levels .

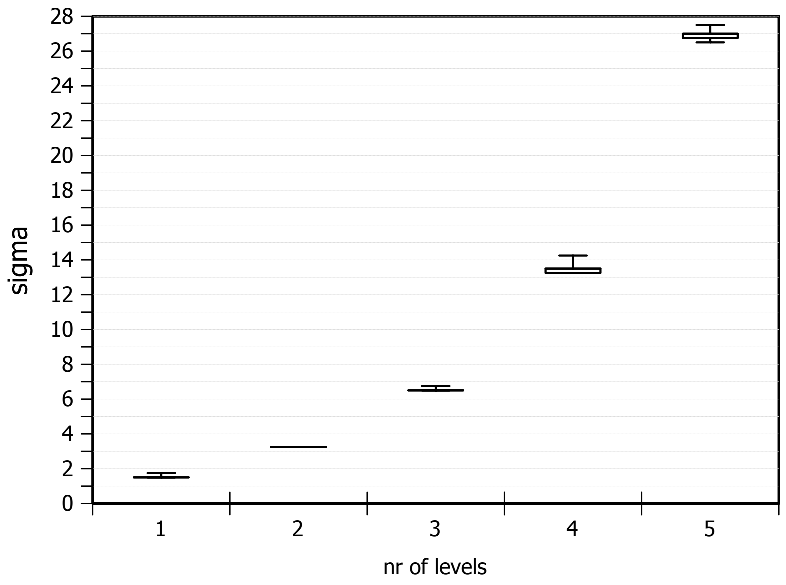


Figure 4.24: Box plots for the determined sigmas when using *4x4 box down-sampling* with varying number of levels .

4.3.2 Spreading

Spreading, in contrast to pyramid methods, gives bigger control over the effective filter size as shown in 4.25. In terms of quality, the approximation achieves the best results when using a spreading radius from 10 to 30, which corresponds to sigma values of 4.25 and 12.75 respectively. In this range, the tent-shape of the filter is a good approximation of the Gaussian kernel, achieving an average probability of detection of lower than 6%.

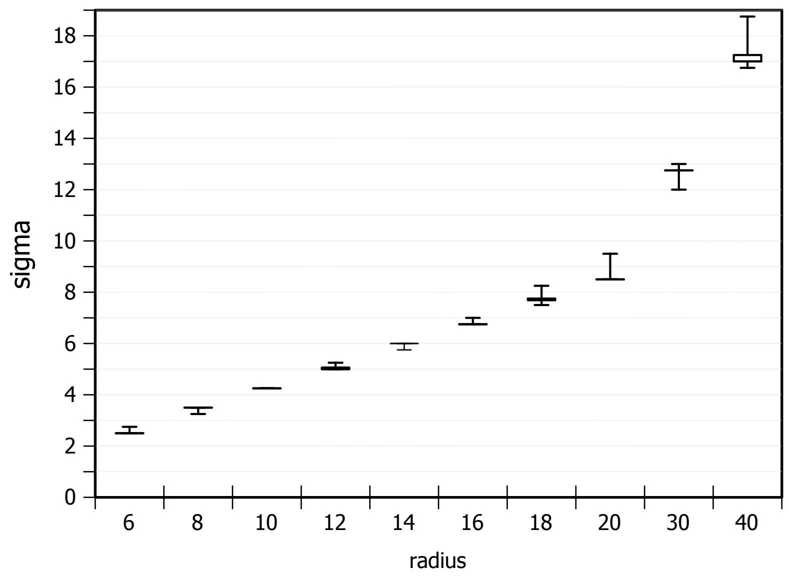


Figure 4.25: Box plots for the determined sigmas when using *spreading* with varying radius.

4.3.3 Repeated box filtering

With the possibility of altering the filter width, as well as the number of iterations, this method grants the biggest control over the approximated filter size. Figures 4.26-4.27 show the determined sigma values for different widths and number of iterations. When analyzing the results of the hdr-vdp2 [19], which can be seen in Tables A.6-A.7, we observe that increasing the number of iterations increases the quality of the approximation. For a filter width of 9, the average probability of detection when using one iteration is $\sim$ 20%, while for 4 iterations it is below 1%.

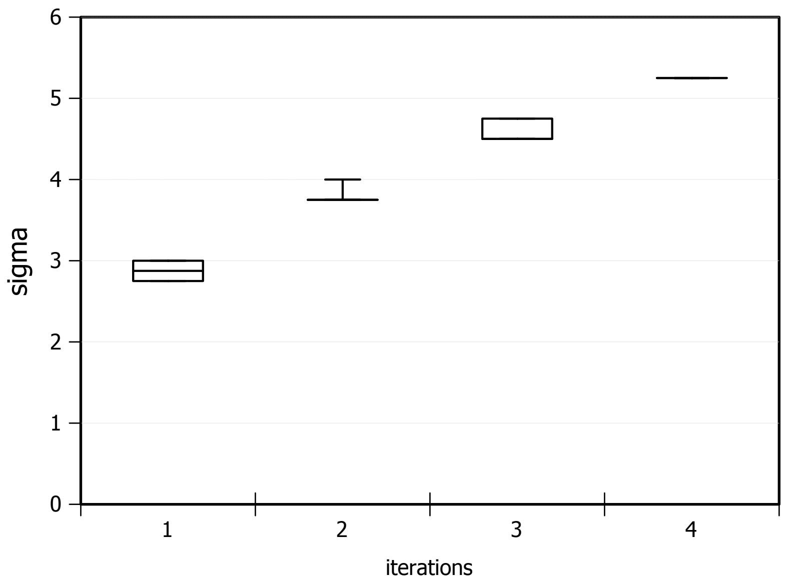


Figure 4.26: Box plots for the determined sigmas when using *repeated box filtering* with a width of 9 and varying number of iterations.

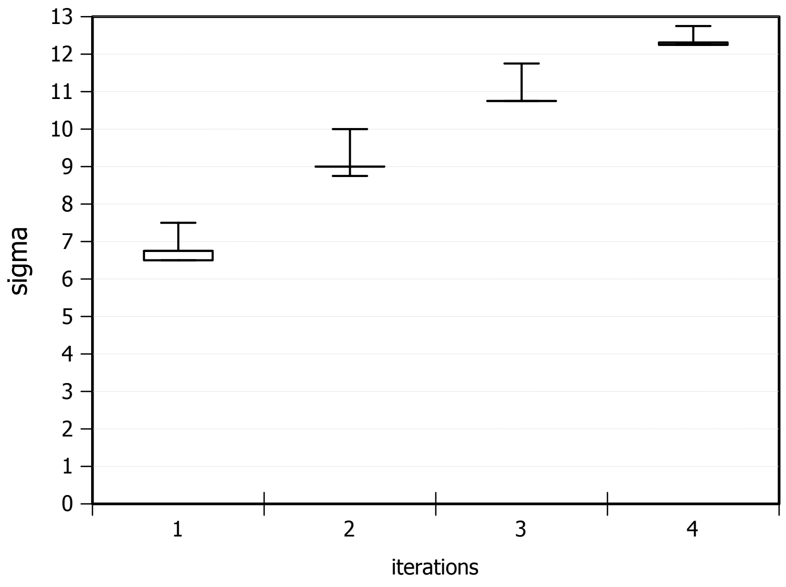


Figure 4.27: Box plots for the determined sigmas when using *repeated box filtering* with a width of 21 and varying number of iterations.

4.3.4 Diffusion

Simulated heat diffusion shows a similar behavior as repeated box filtering. The method allows a fine resolution of possible sigmas, but struggles with the quality of the approximation at lower iteration counts. As can be seen in Table A.4, the average probability of detection decreases strongly with the number of iterations used. For 3 iterations, there is an average chance of detection of $\sim 26\%$, whereas for 5 iterations, the probability of detection is already below 3%. Also note that the increase of sigma per iteration starts to slow down a lot past 20 iterations. To obtain double the sigma value achieved by 20 iterations (sigma=5), the number of iterations has to be increased to 200 (sigma=10).

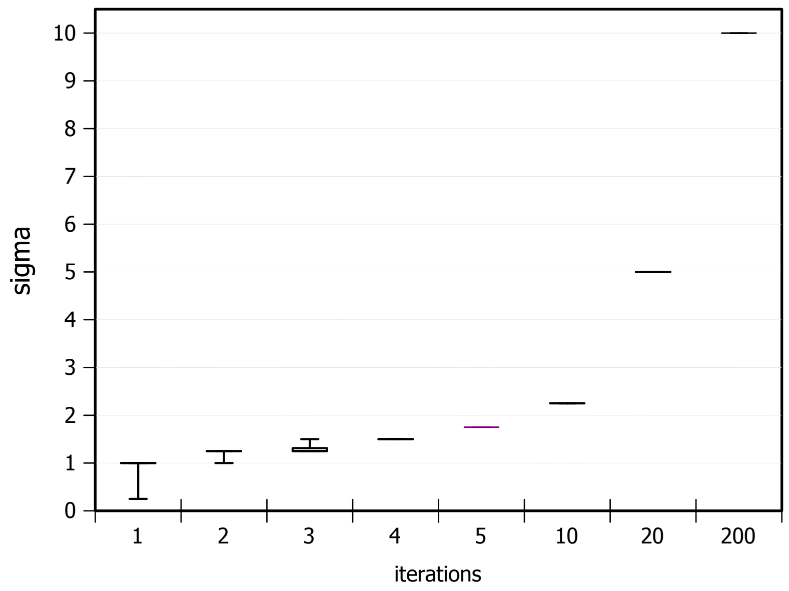


Figure 4.28: Box plots for the determined sigmas when using *simulated heat diffusion* with varying number of iterations.

4.4 Overall Comparison

The following sections present an overall comparison using roughly the same effective filter size for all methods. Three barplots have been created, where each barplot corresponds to a specific sigma value. The parameter set of each method, which corresponds to the sigma according to the results of the test described in 4.1.2, was then chosen for the respective plot. The parameters for each method are given in square brackets on the x-axis, and the closest sigma is given in round brackets. For example, the parameter for pyramid methods is the number of levels, whereas for separated convolution or FFT, the parameter is the filter width. The first overall conclusion that can be made from the Figures 4.29-4.31 is that pyramid methods outperform all other methods at any filter size. This comes at the cost of the least control over the amount of blur applied

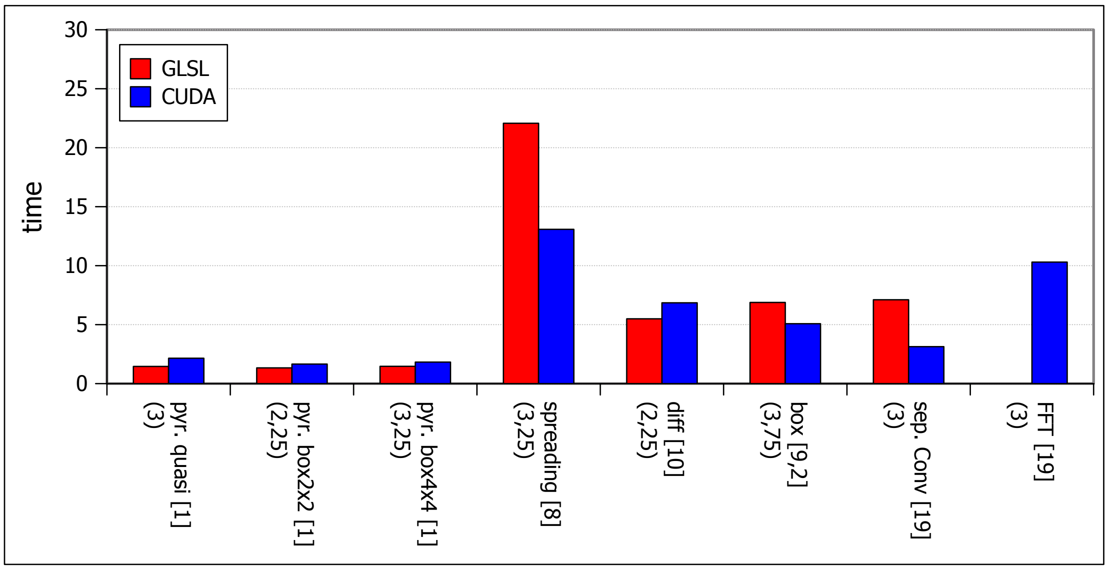


Figure 4.29: Overall performance comparison for $\sigma = 3$

to the image. Also noticeable is that separated convolution implemented in CUDA performs second best in all 3 scenarios. Also we can observe, as has been shown in Section 4.2, that the CUDA implementations outperform the GLSL implementations, except for pyramid methods and simulated heat diffusion.

When looking at the figures separately, starting at Figure 4.29, we see that in case of GLSL, the performance of separated convolution is worse than for repeated box filtering and simulated heat diffusion. The same does not hold true for the CUDA implementations, where separated convolution outperforms both these methods. The two methods with constant time complexity regarding filter size, namely FFT and Spreading, are at the end of the ranking for a sigma value of 3.

For a sigma of 6, the results can be seen in Figure 4.30. The ranking of the methods is equal to the ranking when using a sigma of 3, with the exception that FFT outperforms separated convolution in GLSL in this scenario.

In the last scenario, a sigma value of 12, we see a reversal of the rankings between repeated box filtering and separated convolution in GLSL. This can be explained by the increasing number of iterations needed to achieve higher sigma values using repeated box filtering (the same holds true for simulated heat diffusion). Also the performance of FFT and Spreading surpasses the performance of these two GLSL implementations. Note that for readability reasons, the timings for reshuffling and two-dimensional convolution have been omitted in the shown figures, because the timings of these are far bigger than the timings shown. For better comparison between the different filter sizes, the y-axis has been kept equal in all plots. Therefore, the bars corresponding to simulated heat diffusion are clipped at the edge of the plot in Figure 4.31. The timings are 88ms for GLSL and 106ms for CUDA when applying diffusion with 200 iterations.

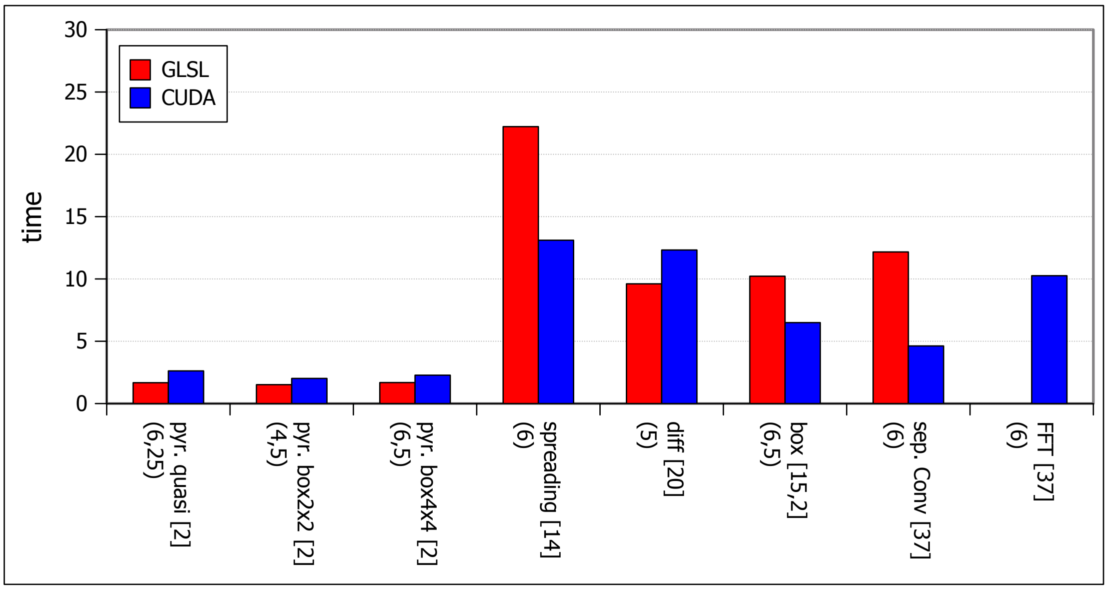


Figure 4.30: Overall performance comparison for $\sigma = 6$.

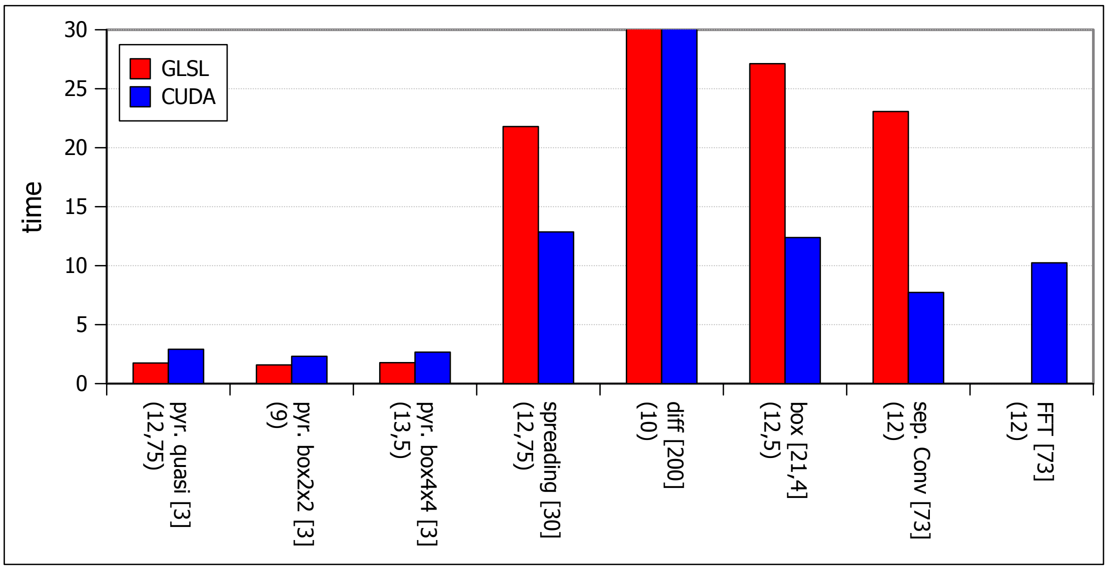


Figure 4.31: Overall performance comparison for $\sigma = 12$.

4.5 Application

Considering the results shown in Section 4.4, we chose to integrate the quasi-convolution technique into a depth-of-field application. The filtering was added to the Layered DoF framework written by David Schedl during his work on “A layered depth-of-field technique for handling partial occlusion in computer renderings” [26]. The previous implementation utilizes separated convolution to blur the depth layers with different Gaussian filter kernels. The speed of the simulation mainly depends on the number of depth layers used and therefore on the number of images that have to be blurred. Figure 4.32 shows the timings for both implementations in dependency of the number of layers used. The values presented in this figure are the times needed to generate a full frame and render it to screen when using a scene containing ∼74k triangles. The function used to split the scene into layers, which is determined by the effective filter size of the method, was chosen to be the same in both implementations to allow a fair comparison. It can be seen that quasi-convolution is superior to separated convolution in terms of performance in every scenario. Furthermore, quasi-convolution shows a lower dependency on the number of layers used, which makes the process three times faster compared to separated convolution when splitting the scene into 8 depth layers.

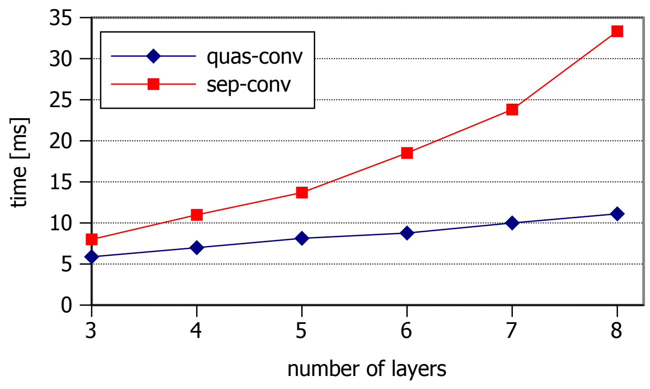


Figure 4.32: Performance comparison for the depth-of-field application when using quasi-convolution [17] and separated convolution.

When looking at the quality of the new implementation, there was no difference noticeable in our test scene. In Figure 4.33, screenshots of the application when using different focal settings are shown for both implementations. When testing the difference of the resulting images with hdr-vdp2 [19], the probability of detection is below 0.5% in the scenarios shown in Figure 4.33. To show where these images differ from each other, we calculated difference images and scaled the grey values in these images up by a factor of 20 (white=255) to make the differences visible.

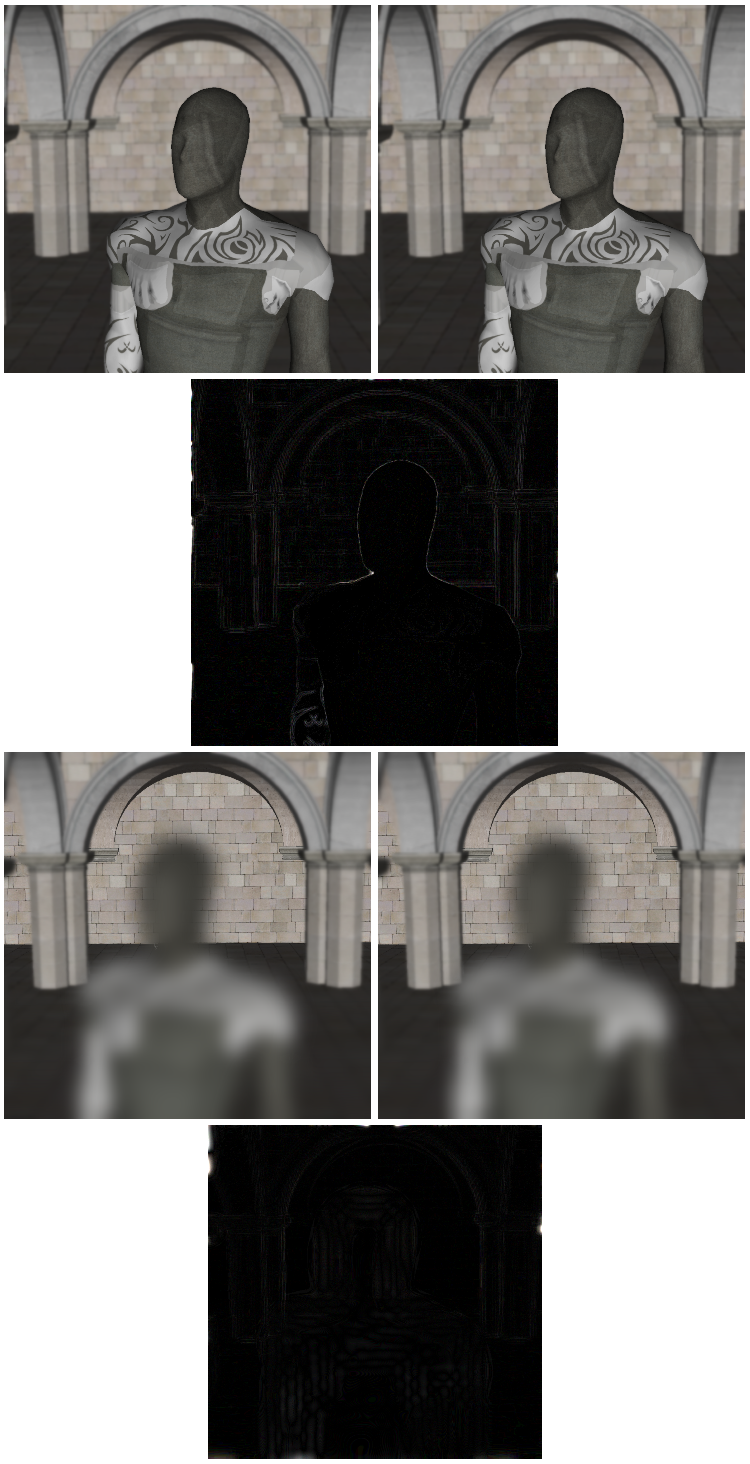


Figure 4.33: Screenshots of the Depth of Field application when using different focal settings quasi-convolution [17] (right-column) and separated convolution (left column). Additionally, difference images for the given scenarios are shown, which are scaled by a factor of 20 (white=255).

CHAPTER 5

Conclusion

We presented a comparison of various filtering techniques in terms of their performance on modern graphics hardware. The increasing computational power of modern GPUs provides developers of real-time rendering applications with the resources needed to realize more and more complex graphical effects. Some of these effects, like Depth-of-Field, require an efficient image filtering technique to achieve real-time frame rates. We also examined the performance of the chosen methods when using a shading language as well as a GPGPU language, additionally to comparing the algorithms versus each other. This was done because implementing special tasks like physics simulations using GPGPU inside real-time rendering applications is increasingly popular.

Considering the vastly different working principles of some of the chosen methods, we first had to determine the effective filter size of each method to allow for a fair comparison. This was done as described in Section 4.4, by comparing the results of a set of images convolved with a Gaussian filter kernel versus the results of a given method. The results of this automatic process where then used to compare the methods on equal footing. Furthermore, they were utilized to compare the quality of those methods which achieve speed by approximation (e.g., pyramid methods) against the ground truth created by convolution. This quality comparison was done utilizing a calibrated visual metric (hdr-vdp2 [19]).

The overall performance comparison showed a clear winner, namely pyramid methods, which have the best performance in all tested scenarios. However, this comes at the cost of having the least control over the filtering process amongst all the tested methods. In case of the two methods having a constant time complexity with regards to filter size, namely FFT and Spreading, we found that FFT is faster in every case. When the application requires the application of convolution with a very large filter kernel, FFT becomes viable, especially if the filter kernel is not separable.

In terms of quality, the results show that amongst the tested variations of down-sampling filters for pyramid methods, quasi-convolution achieves the best result. However, the high performance of the method is a tradeoff for lower image quality when using an increasing number of pyramid levels. Our results show that according to hdr-vdp2, the mean probability that an average user detects the difference between the results of quasi-convolution and the results of convolution increases with the number of pyramid levels. In contrast, for simulated heat diffusion and repeated box filtering, the probability of detection decreases when increasing the number of iterations. When using a radius between 10 and 30, the Spreading algorithm achieves a low probability of detection, showing that the tent-like filter shape is a good approximation of a Gaussian in that range.

The results of the comparison between GLSL and CUDA, in terms of image filtering, do not determine an obvious winner. While we could achieve a noticeable speed-up of separated convolution by the usage of CUDA and its shared memory, in case of pyramid methods, GLSL performed better on our test setup. This is partly because of the higher number of textures needed for the algorithm and the therefore higher setup time of the textures for CUDA. When using summed area tables, we found that the CUDA library cudPP is very useful for an efficient implementation. Through the usage of this library, the Spreading algorithm could be implemented noticeably faster with CUDA. In case of the reshuffling algorithm, we found that the method is not viable for the implementation on graphics hardware. The GLSL version suffers from the disadvantage that the compiler is not able to unroll the used loops, which has a major impact on performance as shown in Section 4.2.8. In CUDA, the operation savings of the algorithm, compared to two-dimensional convolution, are not enough to compensate the additional memory accesses needed.

By implementing quasi-convolution into a layered depth-of-field framework, which originally used separated convolution, we were able to achieve a considerable speed-up of the application while still maintaining comparable quality.

This work can serve as a guideline for graphics developers who are interested in integrating image filtering (especially image blurring) in their application. This includes guidelines for an efficient implementation of image filtering in various ways when using a shading language as well as a GPGPU language. To help developers decide which technique to chose in certain scenarios, a decision tree with explanations is given in Figure 5.1.

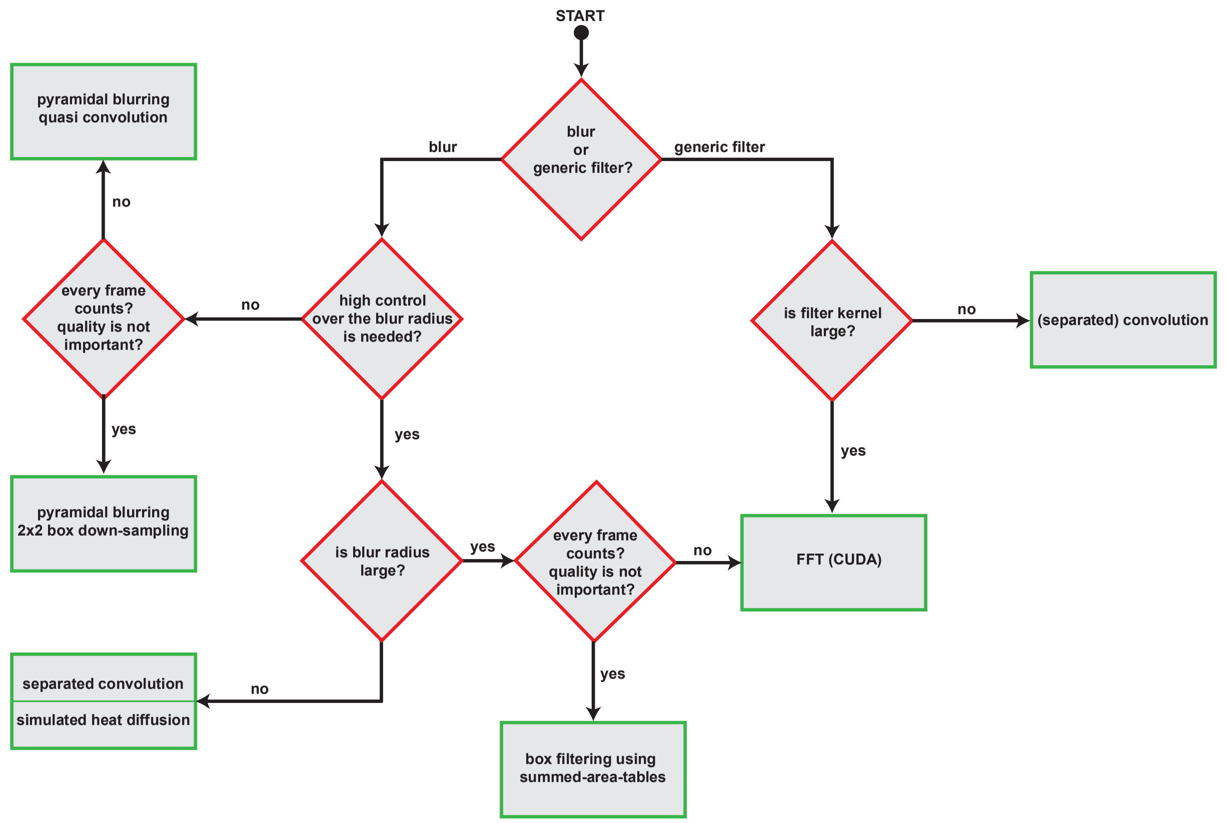


Figure 5.1: Decision tree showing which method to use in different scenarios in terms of performance, quality and filter size. Performance was the main criteria used when constructing this decision tree. When reaching a node where a decision has to be made between performance and quality, the performance path should only be chosen if the time consumed by the filtering process should be as low as possible. For example, the performance difference between quasi-convolution and 2x2 box down-sampling when using pyramidal blurring is very small but could be important if every frame counts. A high control over the blur radius means that the blur radius can be adjusted in fine steps (e.g. pyramidal blur versus convolution). When asking for a large filter radius the definition of large is ambiguous because the actual numbers depend on the filter used (separable or not) and the test setup. For our test setup the exact numbers can be found in Chapter 4. For example, in case of separated convolution versus FTT, large means a filter width of 31 in case of our GLSL implementation and 81 for our CUDA implementation.

APPENDIX A

Estimated sigmas and hdr-vdp2 results

This appendix shows all results of the tests described in Section 4.1.2. There is one table for each method tested with the columns corresponding to different parameter values of the method. The rows correspond to the results of the tests using absolute pixel difference and the quality comparison using hdr-vdp2. The results for absolute difference are labeled with D and the results of hdr-vdp2 are labeled as Q and P (quality mean-opinion score and probability of detection). Note that the rows labeled with D contain the statistics calculated based on the sigmas corresponding to the minimum absolute difference per image. The row labeled lmsE contains the the sigma determined through least-mean-square-Error across all images.

	quasi 1 level	quasi 2 levels	quasi 3 levels	quasi 4 levels	quasi levels 5
D mean	1,5	3,004808	6,235577	12,668269	25,466346
D median	1,5	3	6,25	12,75	25,5
D stddev	0	0,034669	0,058859	0,128349	0,284523
D var	0	0,001202	0,003464	0,016473	0,080953
D lmsE	1,5	3	6	12,75	24,5
Q mean	93,753558	93,903088	93,638355	92,352195	89,244076
Q median	93,774378	93,921711	93,799927	92,549199	89,9437
Q stddev	0,138891	0,105632	0,593361	1,618919	4,422355
P mean	0,082161	0,013241	0,020079	0,02164	0,114088
P median	0,069918	0,008612	0,007716	0,006428	0,027048
P stddev	0,060737	0,027628	0,053701	0,041834	0,194096

Table A.1: Estimated sigma and quality comparison results for quasi convolution downsampling.

	2x2 box 1 lvl	2x2 box 2 lvls	2x2 box 3 lvls	2x2 box 4 lvls	2x2 box 5 lvls
D mean	1,139423	2,24519	4,51442	9,34134	18,71153
D median	1,25	2,25	4,5	9,25	18,75
D stddev	0,125377	0,060439	0,104006	0,178699	0,817074
D var	0,015719	0,003653	0,010817	0,031933	0,667609
D lmsE	1,25	2	5	9	21.5
Q mean	92,67337	92,209028	89,370829	90,01504	86,826128
Q median	92,761699	92,78649	90,737864	91,169228	88,176848
Q stddev	0,819159	2,42225	5,442167	5,410483	5,269664
P mean	0,636263	0,261743	0,281891	0,120606	0,239814
P median	0,710548	0,203468	0,164335	0,051106	0,113844
P stddev	0,258816	0,204844	0,297388	0,23412	0,281608

Table A.2: Estimated sigma and quality comparison results for 2x2 box down-sampling.

	4x4 box 1 lvl	4x4 box 2 lvls	4x4 box 3 lvls	4x4 box 4 lvls	4x4 box 5 lvls
D mean	1,504808	3,25	6,533654	13,408654	26,947115
D median	1,5	3,25	6,5	13,5	27
D stddev	0,034669	0	0,086161	0,191926	0,200101
D var	0,001202	0	0,007424	0,036835	0,040041
D lmsE	1,5	3,25	6,5	14,25	27,25
Q mean	93,688347	93,667544	92,860538	89,989574	85,695062
Q median	93,74298	93,838263	93,472493	90,801502	86,729319
Q stddev	0,239106	0,444646	1,597278	4,388172	7,771568
P mean	0,113021	0,065827	0,032105	0,151036	0,221703
P median	0,071289	0,026269	0,011821	0,035637	0,081208
P stddev	0,140166	0,129775	0,047074	0,245745	0,298087

Table A.3: Estimated sigma and quality comparison results for 4x4 box down-sampling.

	spreading r=6	spreading r=8	spreading r=10	spreading r=12	spreading=14
D mean	2,509615	3,471154	4,25	5,0625	5,995192
D median	2,5	3,5	4,25	5	6
D stddev	0,048546	0,080651	0	0,109309	0,034669
D var	0,002357	0,006505	0	0,011949	0,001202
D lmsE	2,5	3,25	4,25	5	6
Q mean	84,810535	91,161034	92,822716	93,251335	93,244922
Q median	90,321713	93,102546	93,328545	93,418849	93,34653
Q stddev	13,674226	5,977536	2,366573	0,581592	0,487999
P mean	0,256944	0,111448	0,0574	0,029996	0,032712
P median	0,059219	0,028646	0,03029	0,01448	0,015222
P stddev	0,350449	0,218931	0,103678	0,039342	0,039629

	spreading r=16	spreading r=18	spreading r=20	spreading r=30	spreading r=40
D mean	6,778846	7,697115	8,538462	12,745192	17,086538
D median	6,75	7,75	8,5	12,75	17
D stddev	0,080651	0,134096	0,159538	0,144256	0,279551
D var	0,006505	0,017982	0,025452	0,02081	0,078149
D lmsE	7	7,75	9	12,75	17,5
Q mean	93,090785	92,981157	92,871551	91,099813	81,977084
Q median	93,28906	93,271165	93,041107	91,245619	82,609702
Q stddev	0,752084	0,879819	0,644438	1,726675	2,530296
P mean	0,027401	0,030395	0,029015	0,09191	0,157975
P median	0,011804	0,008313	0,011687	0,029645	0,081528
P stddev	0,042504	0,070807	0,049811	0,1584	0,178605

Table A.4: Estimated sigma and quality comparison results for spreading.

	diff i=1	diff i=2	diff i=3	diff i=4	diff i=5	diff i=10	diff i=20
D mean	0,961538	1,235577	1,3125	1,5	1,75	2,25	5
D median	1	1,25	1,25	1,5	1,75	2,25	5
D stddev	0,151662	0,058859	0,109309	0	0	0	0
D var	0,023002	0,003464	0,011949	0	0	0	0
D lmsE	1	1,25	1,25	1,5	1,75	2,25	5
Q mean	92,71380	93,42329	93,68883	93,82386	93,86672	93,9405	93,90884
Q median	92,77302	93,47703	93,71532	93,83581	93,87231	93,94226	93,9256
Q stddev	0,6415	0,31589	0,17338	0,10244	0,08215	0,05749	0,12092
P mean	0,67276	0,25795	0,1140	0,04430	0,0235	0,00632	0,00676
P median	0,6906	0,25100	0,10708	0,03730	0,01950	0,00536	0,00339
P stddev	0,23516	0,14343	0,06482	0,03314	0,01834	0,00423	0,01004

Table A.5: Estimated sigma and quality comparison results for simulated heat diffusion.

	box w=3 i=1	box w=9 i=1	box w=15 i=1	box w=21 i=1
D mean	0,966346	2,875	4,778846	6,725962
D median	1	2,75	4,75	6,75
D stddev	0,14876	0,12622	0,136965	0,205677
D var	0,02213	0,015931	0,018759	0,042303
D lmsE	1	2,875	5	7,25
Q mean	93,206833	93,476168	93,016646	92,862531
Q median	93,255142	93,583281	93,072608	92,99956
Q stddev	0,418251	0,469056	0,673881	0,834124
P mean	0,409445	0,196604	0,369517	0,258634
P median	0,388292	0,129024	0,308556	0,188578
P stddev	0,203566	0,199829	0,302081	0,249228

	box w=3 i=2	box w=9 i=2	box w=15 i=2	box w=21 i=2
D mean	1,25	3,754808	6,379808	8,975962
D median	1,25	3,75	6,5	9
D stddev	0	0,034669	0,126126	0,173344
D var	0	0,001202	0,015908	0,030048
D lmsE	1,25	3,75	6,5	9
Q mean	93,681848	93,958035	93,951482	93,920585
Q median	93,699569	93,960669	93,949255	93,936266
Q stddev	0,174558	0,057391	0,052102	0,103346
P mean	0,121704	0,004406	0,00455	0,003056
P median	0,113205	0,001622	0,002021	0,001458
P stddev	0,076265	0,009323	0,007554	0,00441

Table A.6: Estimated sigma and quality comparison results for repeated box filtering.

	box w=3 i=3	box w=9 i=3	box w=15 i=3	box w=21 i=3
D mean	1,5	4,586538	7,75	10,778846
D median	1,5	4,5	7,75	10,75
D stddev	0	0,120096	0,049507	0,145638
D var	0	0,014423	0,002451	0,02121
D lmsE	1,5	4,75	7,75	11
Q mean	93,831703	93,960231	93,943313	93,882955
Q median	93,84269	93,963119	93,951714	93,910007
Q stddev	0,098782	0,059708	0,073888	0,14021
P mean	0,0411	0,005839	0,002403	0,003074
P median	0,036655	0,001711	0,000775	0,001095
P stddev	0,02997	0,018138	0,005564	0,008294

	box w=3 i=4	box w=9 i=4	box w=15 i=4	box w=21 i=4
D mean	1,75	5,25	8,769231	12,317308
D median	1,75	5,25	8,75	12,25
D stddev	0	0	0,083522	0,122428
D var	0	0	0	0,006976
D lmsE	1,75	5,25	9	12,5
Q mean	93,889416	93,960761	93,905017	93,75758
Q median	93,893393	93,958948	93,937592	93,87755
Q stddev	0,074711	0,051221	0,129914	0,326048
P mean	0,01704	0,004452	0,002054	0,001925
P median	0,012715	0,00138	0,001035	0,001176
P stddev	0,014567	0,010528	0,003287	0,002231

Table A.7: Estimated sigma and quality comparison results for repeated box filtering.

Bibliography

[1] M Bertalmio. Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. In *3DPVT '04 Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 767–773, 2004.

[2] Peter J Burt. Fast filter transform for image processing. In *Computer Graphics and Image Processing*, volume 16, pages 20–51, May 1981.

[3] E Catmull and J Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-aided design*, 10:350–355, 1978.

[4] RL Cook, T Porter, and L Carpenter. Distributed ray tracing. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 137–145, 1984.

[5] Franklin C. Crow. Summed-area tables for texture mapping. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 207–212, July 1984.

[6] Joe Demers. Depth of Field: A Survey of Techniques. In *Gpu Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 23. Addison-Wesley Longman, 2004.

[7] Randima Fernando. Real-Time Glow. In *Gpu Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 21. Addison-Wesley Longman, 2004.

[8] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, chapter 39. 2007.

[9] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast Summed-Area Table Generation and its Applications. In *Computer Graphics Forum*, volume 24, pages 547–555, September 2005.

[10] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965.

[11] Michael Kass, Aaron Lefohn, and John Owens. Interactive depth of field using simulated diffusion on a GPU. *Pixar Animation Studios Tech Report*, 2006.

[12] R. Kosara, S. Miksch, and H. Hauser. Semantic depth of field. In *IEEE Symposium on Information Visualization*, pages 97–104. Ieee, 2001.

[13] TJ Kosloff. *Fast image filters for depth-of-field post-processing*. Phd.thesis, University of California at Berkeley, 2011.

[14] Peter Kovesi. Arbitrary Gaussian filtering with 25 additions and 5 multiplications per pixel. Technical report, 2009.

[15] M. Kraus and M. Strengert. Depth-of-Field Rendering by Pyramidal Image Processing. *Computer Graphics Forum*, 26(3):645–654, September 2007.

[16] M Kraus and M Strengert. Pyramid filters based on bilinear interpolation. In *Proceedings GRAPP 2007 (Volume* . . . , pages 21–28, 2007.

[17] Martin Kraus. Quasi-Convolution Pyramidal Blurring. *Journal of Virtual Reality and Broadcasting*, 6(6), 2009.

[18] Sangyoon Lee. CUDA Convolution, 2008. URL `http://www.evl.uic.edu/sjames/cs525/final.html`.

[19] Rafat Mantiuk, Kil Joong Kim, Allan G. Rempel, and Wolfgang Heidrich. HDR-VDP-2. In *ACM Siggraph Computer Graphics*, page 1, New York, New York, USA, 2011. ACM Press.

[20] Eihachiro Nakamae, Kazufumi Kaneda, Takashi Okamoto, and Tomoyuki Nishita. A lighting model aiming at drive simulators. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques - SIGGRAPH '90*, pages 395–404, New York, New York, USA, 1990. ACM Press.

[21] NVIDIA Corporation. NVIDIA CUDA C Programming Guide Version 4.2, 2012.

[22] M PHARR and G HUMPHREYS. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc.,, 2004.

[23] Victor Podlozhnyuk. Image Convolution with CUDA (SDK document), 2007.

[24] Fatih Porikli. Reshuffling: a fast algorithm for filtering with arbitrary kernels. In *SPIE 6811, Real-Time Image Processing 2008*, volume 6811, page 68110M. SPIE, 2008.

[25] David Schedl. *A layered depth-of-field technique for handling partial occlusion in computer renderings*. Master thesis, University of Applied Sciences, Upper Austria, Campus Hagenberg, 2011.

[26] David Schedl and Michael Wimmer. A layered depth-of-field technique for handling partial occlusion. *Journal of WSCG*, 20:239–246, 2012.

[27] Cary Scofield. 2 1/2-D Depth of Field Simulation for Computer Animation. In *Graphics Gems III*, pages 36–38. 1994.

[28] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. *GPU gems*, pages 313–318, 2005.

[29] Greg Spencer, Peter Shirley, Kurt Zimmerman, and Donald P. Greenberg. Physically-based glare effects for digital images. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH '95*, pages 325–334, New York, New York, USA, 1995. ACM Press.

[30] Magnus Strengert, Martin Kraus, and Thomas Ertl. Pyramid methods in gpu-based image processing. *Proceedings Vision, Modeling, and Visualization*, 2006.

[31] Lance Williams. Pyramidal parametrics. In *ACM Siggraph Computer Graphics*, pages 1–11, New York, New York, USA, 1983. ACM Press.