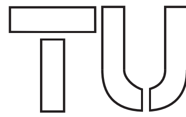


TECHNISCHE UNIVERSITÄT WIEN

Diplomarbeit

Applications of Hardware-Accelerated Filtering in Computer Graphics

unter der Leitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller,
Institut 186 für Computergraphik und Algorithmen,
unter Mitbetreuung von
Dipl.-Ing. Dr.techn. Helwig Hauser,
Forschungszentrum VRVis in Wien,

eingereicht
an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik

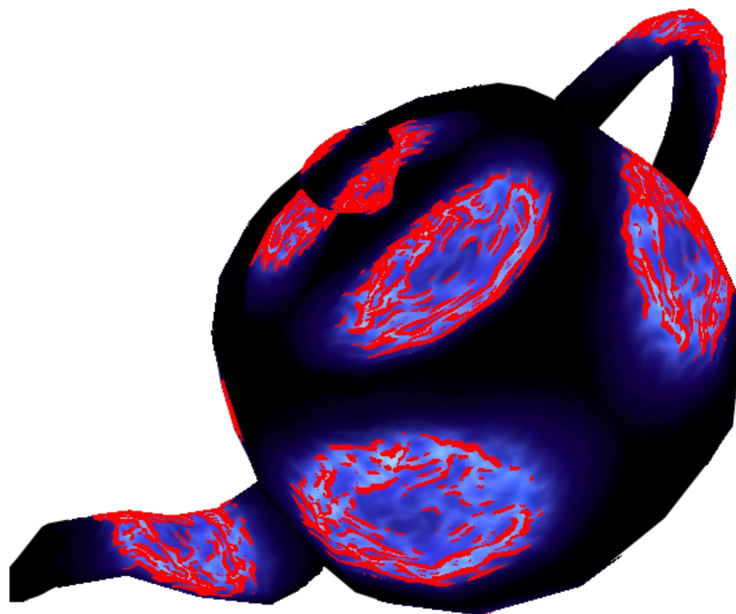
von
Ivan Viola,
Matrikelnummer 9726070,
Kráľovské údolie 25,
81102 Bratislava, Slowakei,
geboren am 25. Juni 1977 in Bratislava.

Wien, im März 2002.

Ivan Viola

Applications of Hardware-Accelerated Filtering in Computer Graphics

(Master Thesis)



supervised by
Dipl.-Ing. Markus Hadwiger,
Dipl.-Ing. Dr.techn. Helwig Hauser,
VRVis Research Center in Vienna, Austria,
and
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller,
Institute of Computer Graphics und Algorithms,
Vienna University of Technology, Vienna, Austria.

http://www.VRVis.at/vis/resources/DA-IViola/
mailto:viola@cg.tuwien.ac.at

Trademarks used in this thesis generally belong to respective owners.

Abstract, Kurzfassung

(engl.) Two of the most important issues of computer graphics – especially in raster graphics and volume visualisation – are sampling and reconstruction. These operations must fulfill particular conditions of sampling theory in order to be able to represent arbitrary continuous functions by discrete samples and reconstruct them from these samples without significant information loss. Several approaches for high-quality reconstruction have been introduced to the computer graphics community. These approaches are mostly implemented in software, possible only as pre-process step. The only two ways of reconstruction that are usually fast enough for real-time rendering are nearest neighbor and linear interpolation filtering, but the quality of these filtering processes is often not sufficient.

This work summarizes the hardware-based methods that exploit the features of today's graphics chips for filtering tasks. These methods are divided in two parts, i.e., high-resolution filtering and image processing. Both methods are based on the distribution principle of convolution known from splatting based volume rendering algorithms and they share the same general principle. The difference is in the implementation of the algorithms themselves.

High-resolution filtering employs high-order filters in order to improve the quality of resampling tremendously. The implemented algorithms exploit symmetry or separability properties to make the filtering more efficient. We compare it to the existing, natively supported solution, i.e., linear interpolation to our filtering implementation using higher order filters. This is shown in various application areas like surface-texturing, and solid-texturing, animated textures, or derivative filtering; at interactive framerates.

The image processing algorithms are simplified general filtering algorithms to increase effciency and performance. We show the usability on smoothing and edge detection, the important operations of image processing and pattern recognition. We combine this techniques together with other hardware features to provide hardware-accelerated artistic rendering techniques. These are also presented in a rendering system that provides non-photorealistic rendering effects.

(deut.) Zwei der wichtigsten Problemstellungen der Computergraphik – besonders der Rastergraphik und der Volumenvisualisierung – sind Sampling und Rekonstruktion. Diese Operationen müssen bestimmte Bedingungen der Samplingtheorie erfüllen, um beliebige kontinuierliche Funktionen durch diskrete Samples darstellen, und später die Originalfunktion aus diesen Samples ohne wesentlichen Informationsverlust wieder rekonstruieren zu können.

In der Computergraphik gibt es verschiedene Ansätze für Rekonstruktion mit hoher Qualität. Meistens können diese nur in Software implementiert werden, was aus Geschwindigkeitsgründen meist vorab durchgeführt werden muß. Die einzigen beiden Rekonstruktionsfilter, die üblicherweise schnell genug sind um für Echtzeit-Rendering eingesetzt zu werden, sind lineare Interpolation und Nearest-Neighbor-Interpolation.

Die vorliegende Arbeit beschäftigt sich mit Hardware-basierten Methoden für Filtering und Rekonstruktion, welche die Fähigkeiten heutiger Graphikhardware ausnützen. Diese Methoden können in die zwei Bereiche High-Resolution Filtering und Bildverarbeitung eingeteilt werden. Beide basieren auf der Berechnung der Faltungssumme in verteilter Reihenfolge – eine Idee, die aus allen Splatting-basierten Volumenrenderingalgorithmen bekannt ist. Jedoch haben die hier dargestellten Methoden nur diese Grundidee mit Splattingverfahren gemeinsam und unterscheiden sich insgesamt sehr stark, besonders in bezug auf die Implementierung.

High-Resolution Filtering erlaubt den Einsatz von Filtern höherer Ordnung, wodurch die erreichbare Qualität des Resamplingvorgangs deutlich erhöht werden kann. Die implementierten Algorithmen nützen Symmetrie und/oder Separabilität dieser Filter, um die Berechnungsgeschwindigkeit zu erhöhen. Wir vergleichen die Resultate mit den Filtern, die direkt von der Graphikhardware unterstützt werden: lineare Interpolation und Nearest-Neighbor-Interpolation. Hierzu zeigen wir Anwendungen in vielfältigen Bereichen, wie Oberflächentexturen, Volumenstexturen, animierten Texturen, oder Filtering von Gradienten; all dies mit interaktiven Bildaufbauraten.

Die Algorithmen für Image Processing sind vereinfachte Versionen der allgemeinen Filteringalgorithmen, um die Berechnungsgeschwindigkeit weiter zu erhöhen. Wir zeigen Anwendungsgebiete am Beispiel von Glättungs- und Kantenerkennungsfiltern, welche wichtige Grundoperationen in der Bildverarbeitung und der Mustererkennung darstellen. Diese Techniken kombinieren wir mit anderen Fähigkeiten der Graphikhardware für hardwarebeschleunigtes "artistic" Rendering. Wir zeigen diese Möglichkeiten anhand eines Renderingsystems, das nicht-photorealistische Effekte darstellen kann.

Contents

Abstract, Kurzfassung **i**

1 Introduction **1**

1.1 Computer graphics and graphics hardware 2
1.2 Sampling and reconstruction 2
1.3 Filtering applications 3

2 Filtering Basics **4**

2.1 The Fourier transform 4
2.2 Linear convolution 5
2.3 Sampling 6
2.4 Signal reconstruction 7
2.5 Reconstruction and aliasing 9
2.6 Derivative reconstruction 11
2.7 Image processing 12

3 State of the art in reconstruction and hardware-accelerated graphics **15**

3.1 Reconstruction filters in computer graphics 15
3.2 Hardware-accelerated graphics 17
3.3 Relevant features for hardware filtering 22
3.4 Hardware-based filtering 24

4 Hardware-based filtering using textures **25**

4.1 Distribution principle 25
4.2 Distribution principle in hardware 26
4.3 High-resolution filtering vs. image processing 28

5 High-resolution filtering **30**
5.1 Filtering algorithms 30
5.2 Applications of high-resolution filtering 34
5.3 Problems and solutions 41

6 Image Processing **44**
6.1 Filtering algorithms in image processing 45
6.2 Applications in image processing 46
6.3 Problems and Solutions 57

7 Implementation **58**
7.1 Kernel representation 58
7.2 Filtering control 60

8 Summary **62**
8.1 Hardware-based filtering using textures 63
8.2 Filtering algorithms 64
8.3 Applications 65

Conclusions **69**

Bibliography **70**

Acknowledgements **73**

Curriculum vitae **74**

Chapter 1

Introduction

> Machines are to work; human beings to use their minds.
>
> *Pollyanna's Principle [4]*

Computer graphics is a part of computer science dealing with approaches to represent, understand or process graphical data. The abstract term *computer graphics* can be divided into the following two categories according to the methods of processing the data:

- **Image synthesis** generates synthetic images from an underlying scene description. A scene is a virtual world consisting of explicitly described objects. This description can be given analytically in form of mathematical equations, as a set of geometric primitives such as points, lines, polygons, spheres, volumetric datasets, or others. Image synthesis also deals with methods for manipulating and lighting scene objects.
- **Image analysis** reconstructs the information stored in the graphical form, which is represented mostly with real-world scenes in form of digitized photographs, video streams, or volumetric datasets as well. It covers all techniques for quality enhancement, noise reduction, edge detection, segmentation used either for image processing and editing or for pattern recognition and feature extraction. The scene is mostly described by discrete samples of the real world, called pixels (picture elements) in the 2D case, and voxels (volumetric pixels) in the 3D case. Image analysis tries to "really" understand the information stored in these graphical elements.

Each computer graphics category uses a set of input and output devices to manipulate or display the graphics. These devices can be either *vector*- or *raster*-based. Vector-oriented devices such as tablets or plotters are rather specific and are not part of a standard computer setup. In practice we rather use more raster-based devices that represent continuous scenes through discrete samples. Typical representatives of raster devices are monitors or cursor pointers that operate on a set of two-dimensional point arrays. When the internal scene representation is vector-oriented, for example, as a set of polygons, it is necessary to sample them into a set of points in a process called rasterization for displaying it.

Computer graphics is not the only scientific branch that uses samples for the representation of continuous signals. Similar approaches are used as well in signal processing or acoustics, for example. The difference to computer graphics is that signal processing usually is dealing with 1D signals, like sounds or electromagnetic signals, while computer graphics mostly is dealing with 2D or 3D signals. Two-dimensional functions represent images, and three-dimensional samples volumetric objects.

1.1 Computer graphics and graphics hardware

Image synthesis provides a lot of methods how to represent the underlying structures in a graphical way. This process is called *rendering* and it is done in several steps usually called *graphics-rendering pipeline*. The function of the rendering pipeline is to generate a two-dimensional image given by a virtual camera, scene objects, light sources, lighting models, textures, and others. The locations and shapes of the objects in the image are given by their underlying structure and placement of the camera in the scene. Appearance of these objects is affected by material properties, light sources, textures and lighting models.

Software implementations of the rendering pipeline make it possible to fully customize the rendering process. However, the software-based rendering has its drawbacks in terms of performance issues. Therefore there are nowadays some stages of the rendering pipeline implemented in hardware. In the early years of hardware acceleration it was possible to set up the scene objects from the geometric primitives, use a limited number of light sources, customize the camera settings, but the lighting model was hard-wired in the hardware and the customization was strongly limited. The hardware acceleration was implemented only in expensive high-end graphical systems and there was actually no hardware acceleration on the consumer level. Due to the growing importance of the game industry several manufacturers started to produce the first low-cost consumer graphics hardware equipped with hardware acceleration. The first generation of low-end hardware accelerators were in fact simple rasterizers, i.e., only the two last stages were done in hardware. Nowadays, the consumer graphics cards are able to do the transform and lighting stage, which significantly increases the rendering performance. These cards are equipped with a large feature set that makes possible to fully customize the hardware-based rendering pipeline to realistically render the scene at interactive framerates.

Although these features are primarily intended for the game industry, they are usable in other applications as well. A good example is hardware-accelerated volume rendering on consumer graphics cards or hardware-based filtering, discussed in this thesis.

1.2 Sampling and reconstruction

The theory dealing with the discretization of continuous data and its reconstruction to the continuous form is called sampling theory. It tells, how dense to sample the continuous function to be optimally represented by its samples. The theory recommends the *sinc* function to reconstruct the continuous signal without any information loss [33] (in case sampling was done without aliasing). The reconstruction process is described as a convolution of the sampled function with a reconstruction filter. Unfortunately the sinc function is not very suitable for practical use because of its infinite width, therefore several other reconstruction filters have been introduced [37]. Filters can be classified according to their reconstruction quality and their computational complexity. It is obvious that high-quality reconstruction is more complex and requires more computational time. The fastest reconstruction methods are nearest-neighbor approximation (equivalent to the so-called *box* filter) and linear interpolation (*tent* filter). These filters are also natively supported by graphics hardware that means that they application is fast, but the reconstruction quality often leads to artifacts. Such artifacts can be removed using higher-order filters such as piecewise cubic splines [18, 16], for example. Although these advanced filters are not natively supported by hardware, the reconstruction process can exploit the texture hardware features and capabilities to increase performance.

1.3 Filtering applications

The above discussed raster and volume graphics needs the reconstruction, for example, for image resampling tasks like minification or magnification. The volume graphics uses it for rendering purposes as well. The derivative reconstruction can be used for gradient estimation. The reconstruction is in fact just a subclass of general *filtering* operations. Filtering is an abstract term describing the convolution of input data with an arbitrary filter. We focus on two types of filtering – the first is using high-resolution filters, truly simulating the continuous ones for reconstruction tasks and the second are image processing tasks generally using low-resolution filters. Both types have been implemented exploiting hardware-acceleration for real-time processing. In the case of high-resolution filters (Chapter 5) we use cubic filters and a windowed sinc (Chapter 6) of width four. The image processing filtering uses standard filters used in image processing at very high processing speed. For both types of filtering we discuss possible applications that all require real-time performance, so the hardware-based filtering is especially suitable for them. In our implementation we use the general filtering approach proposed by Hadwiger et al. [8, 9, 10] as well as image processing filtering introduced by James [15].

Firstly, in chapter 2, we mention the basic facts and properties of the filter theory to clearly understand this part of the theoretical background. Chapter 3 reviews the state-of-the-art techniques of filtering as well as features of graphics hardware. We will shortly mention other techniques using hardware for filtering applications. Chapter 4 describes the basic concept of a general texture-based filtering algorithm. Chapter 5 reviews applications and algorithms of high-resolution filtering. Image processing filtering and its applications is described in chapter 6. Chapter 7 is dealing with implementation issues. Chapter 8 summarizes the thesis and after that some conclusions are sketched.

Chapter 2

Filtering Basics

This chapter reviews the principal ideas of sampling theory and digital filtering. The sampling theory is dealing with two fundamental tasks, i.e., sampling and reconstruction. Sampling describes how dense the original function should be sampled to be optimally described by its discrete representation. Reconstruction theory describes how to get the continuous function (or an as good as possible approximation) from its discrete samples. An answer to several related questions lies in the frequency domain where spectral analysis is used to examine the spectrum of the sampled data. How to get the corresponding spectrum from a function is described by Fourier analysis.

2.1 The Fourier transform

Most functions can be represented as a sum of sine and cosine waves. In case of continuous non-differentiable functions, the sum consists of infinite waves. The sine and cosine waves used for representation are of different frequencies, amplitudes – called frequency response or spectrum of the function – and phases. Accordingly, this is a representation in the so-called frequency domain. The Fourier transform [5, 29] describes how to transform the spatial signal f to the frequency domain. The following equations dealing with the Fourier transform are taken from [5]. The Fourier transform is given by:

$$F(\omega) = \int_{-\infty}^{\infty} f(x) \cdot e^{-i2\pi\omega x} dx,$$

where ω is a value in the frequency domain. The inverse Fourier transform for reconstructing $f(x)$ from $F(\omega)$ is defined as:

$$f(x) = \int_{-\infty}^{\infty} F(\omega) \cdot e^{i2\pi\omega x} d\omega$$

The Fourier transform generates a complex output for a real spatial input. The real and complex part of the output specifies the amplitude and phase response. The frequency response and the phase is defined by:

$$|F(\omega)| = \sqrt{[Re(F(\omega))]^2 + [Im(F(\omega))]^2}$$

$$\phi(\omega) = \tan^{-1}\left(\frac{Im(F(\omega))}{Re(F(\omega))}\right)$$

These equations are dealing with continuous functions. However in practice, we are often dealing only with samples of the original function f. To transform discrete functions to the frequency domain, we use the so-called discrete Fourier transform (DFT), which is derived from the continuous case:

$$F[\omega] = \sum_{x=0}^{N-1} f[x] \cdot e^{-i2\pi\omega x/N}$$

$$f[x] = \frac{1}{N} \sum_{x=0}^{N-1} F[\omega] \cdot e^{i2\pi\omega x/N}$$

One interesting characteristic of the Fourier transform is its separability. To extend the DFT to two dimensions we can rewrite the functions as:

$$F(u,v) = \sum_{y=0}^{M-1} \left(\sum_{x=0}^{N-1} f(x,y) \cdot e^{-i2\pi ux/N} \right) \cdot e^{-i2\pi vy/M}$$

$$f(x,y) = \frac{1}{M} \sum_{v=0}^{M-1} \left(\frac{1}{N} \sum_{u=0}^{N-1} F(u,v) \cdot e^{i2\pi ux/N} \right) \cdot e^{i2\pi vy/M}$$

In practice the fast version of the Fourier transform called Fast Fourier transform or FFT for short [6, 29] is often used. This version reduces the computational complexity from $O(n^2)$ to $O(n \cdot log(n))$.

2.2 Linear convolution

A fundamental operation in image processing and reconstruction issues is an operator called convolution. It is defined by an integral in the continuous case and as a sum in the discrete case [24, 42]:

$$g(x) = (f * h)(x) = \int_{-\infty}^{\infty} f(x') \cdot h(x - x') dx'$$

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - \lceil m \rceil + 1}^{\lfloor x \rfloor + \lceil m \rceil} f[i] h(x - i) \tag{2.1}$$

Easily described, convolution is a sliding and weighted average of one function with another function providing the weights. The discrete version of the convolution from equation 2.1 describes the reconstruction process, where $g(x)$ is the reconstructed function, $f[i]$ the sampled representation of the original function, and $h(x)$ the reconstruction filter of width $2m$. The continuous convolution as well as the discrete case are illustrated in figures 2.1 and 2.2. Figure 2.1 shows a continuous convolution of two box functions resulting in the tent function. The discrete convolution is illustrated in figure 2.2, where the first function is a discrete function and the second (often referred as the filter or kernel) function is the tent function. The result is a continuous function that approximates the original continuous function linearly. This process is called linear interpolation. One interesting property of the convolution operation is the so called convolution theorem [32]:

> *The spectrum of the convolution of two functions in the spatial domain is equivalent to the product of the transforms of both input signals, and vice versa.*

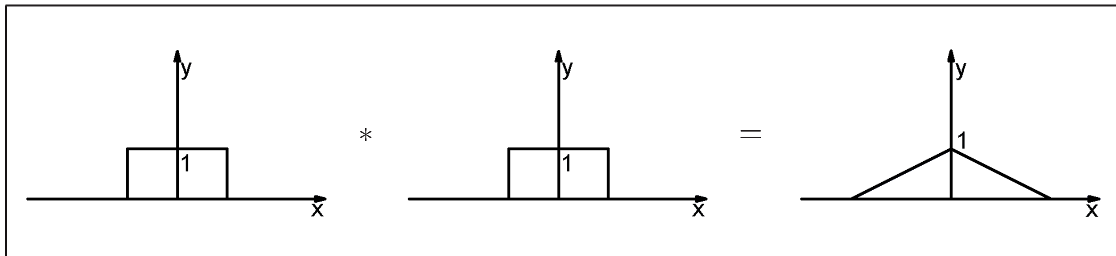


Figure 2.1: Example of a convolution of two box functions that results in a tent function [35].

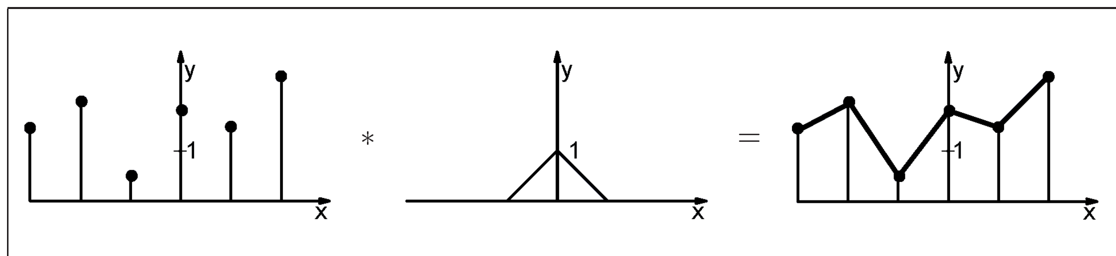


Figure 2.2: Reconstruction process using the tent filter for convolution with a sampled function [35].

This is used in practice when the convolution is a time-consuming operation, e.g., the function representing the kernel is rather wide. The result of the spatial convolution could be obtained by transforming both functions to the frequency domain, calculating the product of these two functions and finally transforming the product back to the spatial domain. Other important properties of convolution are commutativity, distributivity and associativity [24].

2.3 Sampling

The sampling theory uses several important functions for its analytical definitions. The first function is called the impulse function (also known as Dirac impulse) [24]:

$$\delta(x) = \begin{cases} \infty & x = 0 \\ 0 & else \end{cases}$$

A function, which is derived from the Dirac impulse and which is also fundamental in the sampling theory is called *comb* (sometimes also *shah*) function [24, 42], consisting of equally spaced impulses defined by

$$comb_T(x) = \sum_{n=-\infty}^{\infty} \delta(x - n \cdot T),$$

where T is the distance parameter between two adjacent impulses. The process of sampling can be described as the multiplication of the original (continuous) signal with the comb function. The quality of sampling varies according to the value of parameter T. How to set the parameter to be able to retrieve the original function back from its samples is described by the fundamental theorem of the sampling theory, the so-called Shannon theorem [33]:

A function $f(x)$ that is band-limited and sampled above the Nyquist frequency is completely determined by its samples and can be perfectly reconstructed by convolving the set of samples with the sinc function.

The term *band-limited function* means function with finite frequencies. This can be examined in the frequency domain; it should not contain any frequencies outside the particular interval called base band [24]. The *Nyquist frequency* is twice the highest frequency component in the spectrum of the original function [24]. Figure 2.3 describes the correspondence between the spatial and the frequency domain during the sampling process. The sampled function creates an array of spectrum replicas, while the spectrum in the center is denoted as primary spectrum and the other as alias spectra.

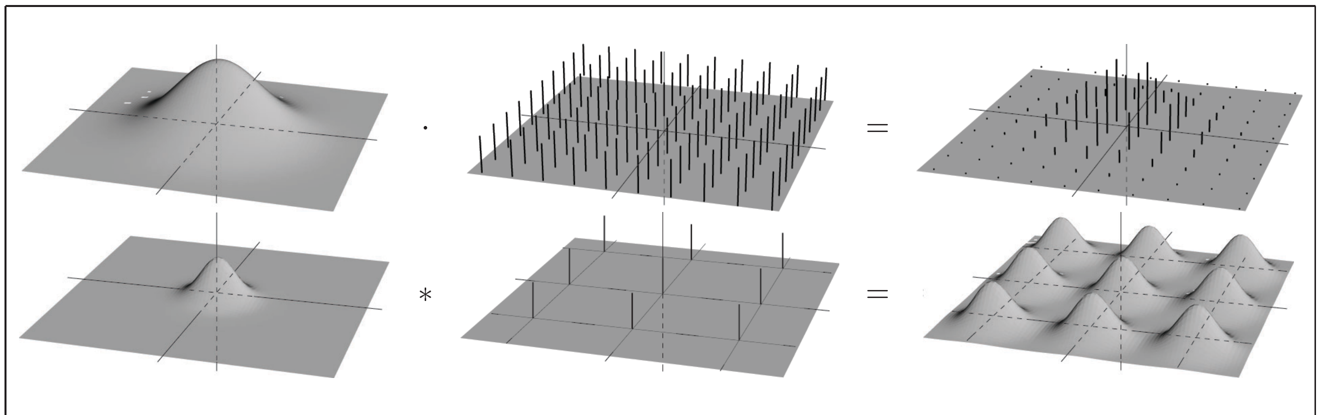


Figure 2.3: Two-dimensional sampling in the spatial domain (top) and frequency domain (bottom) [18].

2.4 Signal reconstruction

The Shannon theorem defines not only how to sample the continuous function, but also specifies the filter kernel which is the optimal one for reconstruction. If we take a look into the frequency domain, the reconstruction (based on the sinc filter) is simply a multiplication of the spectrum and a box function of the same width as the bandwidth of the primary spectrum. This "cuts" off all the alias spectra and leaves only the primary spectrum (untouched). The primary spectrum represents the original continuous function in the frequency domain.

Ideal reconstruction

In order to reconstruct a function in the spatial domain, the reconstruction process is a convolution of the sampled function and the spatial representation of the box filter in a frequency domain as mentioned above. This box filter corresponds in the spatial domain to the *sinc* function and vice versa [32]. The sinc filter is defined as:

$$sinc(x) = \begin{cases} 1 & x = 0 \\ \frac{\sin \pi x}{\pi x} & else \end{cases}$$

This function is illustrated in figure 2.4. The problem that makes reconstruction impracticable using the sinc filter, is its infinite extent.

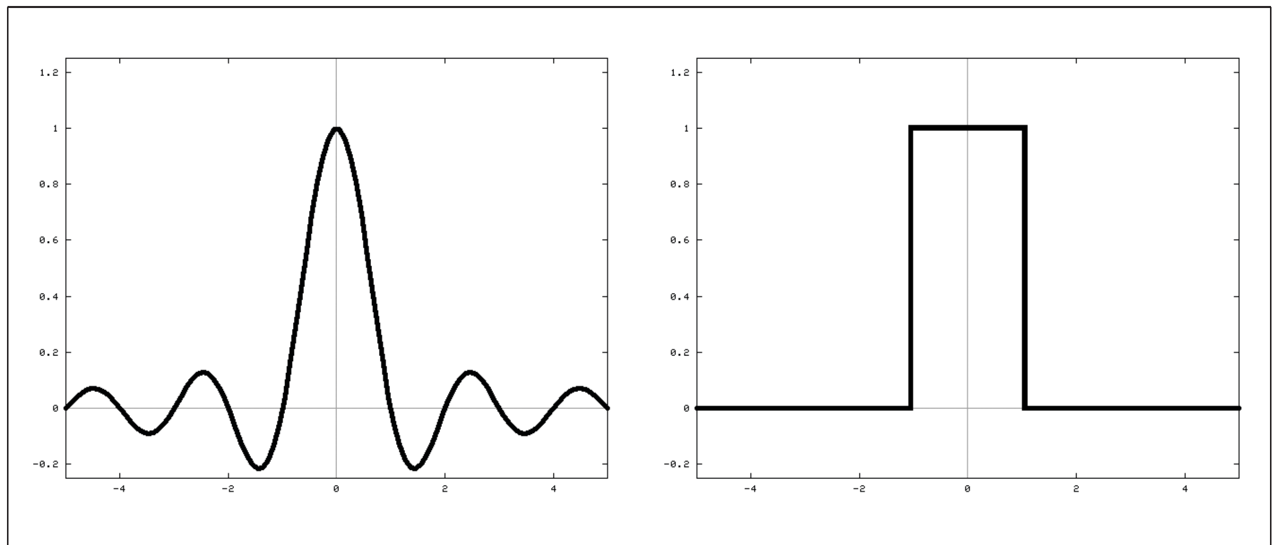


Figure 2.4: Sinc function (ideal reconstruction filter) on the left and its frequency response (box filter) on the right.

Reconstruction filters

In practice, other filters approximating the sinc function are used for signal reconstruction. The choice of useful filters is based on the similarity to sinc on a given interval, while outside this interval the values are equal to zero. Similar to other areas of computer science, the challenge to find a good compromise between computational complexity and quality also comes up here. In general, the most important criteria are the similarity to sinc (evaluated by its frequency response) and the width of the filter.

Nearest neighbor approximation (Box filter)
The most primitive solution which is providing fast but usually not satisfying results is the nearest-neighbor interpolation. It is a convolution with the box filter of width one, but in the spatial domain. This results to a discontinuous, piecewise constant function. The filter as well as the results are illustrated in figure 2.5.

Linear interpolation (Tent filter)
The biggest advantage of linear interpolation is the simple filter kernel, which is the already mentioned tent function. The kernel is of width two that means in the 1D case that only two samples contribute on a reconstructed value. This makes the filtering fast enough, although it introduces visible artifacts. Filtering with the tent function results in a continuous piecewise linear function which is not differentiable at the sample positions. The process is illustrated in figure 2.2.

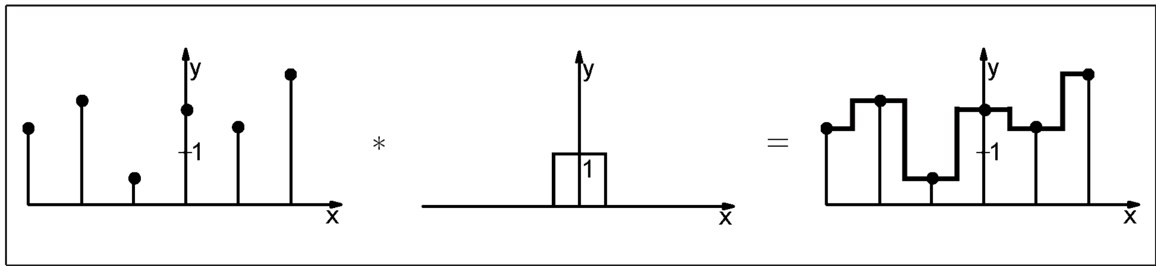


Figure 2.5: Reconstruction process using the box filter for convolution with the sampled function.

Cubic interpolation
The piecewise cubic interpolation is using four samples for reconstruction of one value. This method provides smooth transitions in-between sample points, i.e., the resulted function is differentiable on the whole reconstruction interval. Using suitable cubic functions as filter kernels is already considered as state of the art [16, 18]. This method will be discussed in the next chapter in more detail. Unfortunately this kind of filtering is rather time-consuming due to the computational complexity, especially dealing with large or higher-dimensional datasets.

Gaussian filters
Another approach is to use the well-known Gaussian lobe as a filtering kernel [37]. The Gaussian function is defined as:

$$G(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-(x-\mu)^2/2\sigma^2} \tag{2.2}$$

Although the filtering process generates a smooth continuous function, the disadvantage is that the Gaussian lobe reaches zero only at infinity. In other words, the Gaussian lobe has infinite support. By using suitable μ and σ values the results are similar to other state-of-the-art filtering methods. Another characteristic of the Gaussian lobe is its approximative behavior, which results from the fact that it is positively defined on the whole interval of real numbers. This behavior introduces blurring artifacts. In the next chapters we will see that this also can be an advantage. It simplifies the hardware implementation of the filtering process. The Gaussian lobe is shown in figure 2.6.

Windowed-sinc filter
The idea of windowing is to approximate the sinc function on a given interval by multiplying the sinc with another (limited) function, called window [36]. The window function is defined outside the respective interval to be constantly zero and usually provides smooth transitions on the boundaries of the interval. After multiplication with the sinc, the resulting function has also limited support, but it preserves some of the behaviors of the sinc function. A more detailed description is given in the next chapter.

2.5 Reconstruction and aliasing

The sampling and reconstruction theory describes various aspects that must be considered for high-quality filtering. What happens, when the conditions are not fulfilled, is described in this section. The term “aliasing” is not consistently defined; Mitchell and Netravali [18] define all

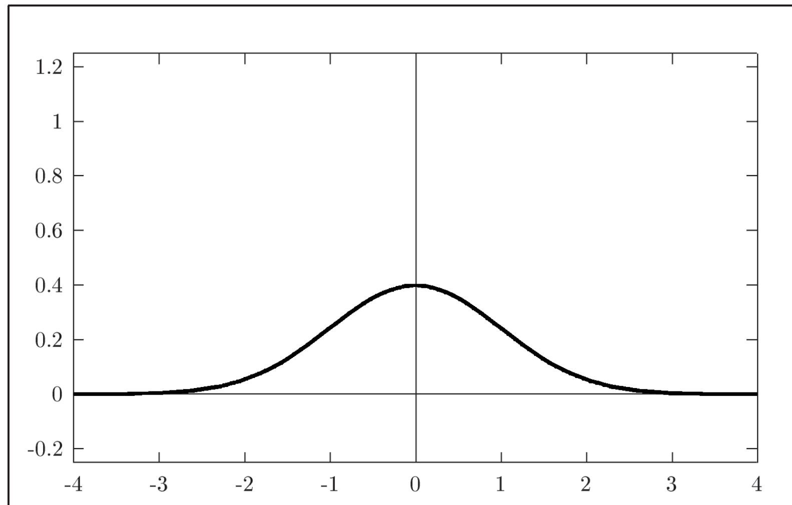


Figure 2.6: Gaussian lobe with $\mu = 0$ and $\sigma = 1$.

artifacts caused by incorrect sampling or reconstruction as aliasing. Other definitions consider only sampling errors as aliasing, reconstruction artifacts are simply reconstruction errors. We use the Mitchell and Netravali definition of aliasing [18], and divide the errors into the following categories:

Pre- and post-aliasing

Both of these aliasing artifacts occur, when the fundamental conditions of the sampling theory are not fulfilled. Pre-aliasing occurs when the signal is not band-limited before sampling or not sampled properly so that the alias spectra overlap in the frequency domain. The solution is either to increase the sampling frequency or to preprocess the function with a low-pass filter.

Post-aliasing occurs due to the incorrect reconstruction when the alias spectra "leak" into the reconstruction. The reconstruction kernel is significantly non-zero at frequencies, also above the Nyquist frequency. The pre- and post-aliasing effect is illustrated in figure 2.7. It shows what happens in the frequency domain with the spectrum due to aliasing. The first problem is that it is impossible to cut off the alias spectra correctly, because they are overlapping the primary spectrum. The other case shows the incorrect setting of the frequency filter width.

Blurring

Blurring also called smoothing occurs due to an incorrect choice of a reconstruction kernel. This effect is strongly visible when approximating kernels are used, because they simply do not reconstruct the original function in the interpolative sense. This means even the reconstructed values at the sample points are not equal to the sampled values.

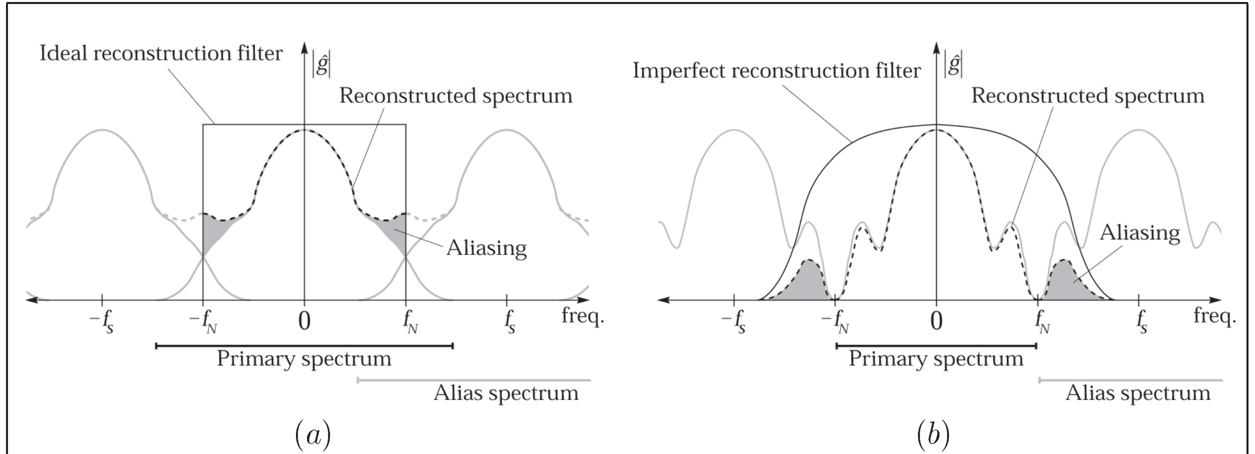


Figure 2.7: Pre-aliasing (a) and post-aliasing (b). [18].

Ringing
Low-pass filtering of step discontinuities results in ringing. This effect is also called Gibbs phenomenon.

Anisotropy
This artifact occurs in the multi-dimensional case when the filter is not spherically symmetric. Anisotropy manifests itself as an asymmetry in smoothing or post-aliasing effect.

2.6 Derivative reconstruction

Until now the so-called function reconstruction was described, when the filtering process reconstructs the function itself. But there are many cases when the first order (or n-th order) information is needed as well. For example, to estimate the gradients of an image, the continuous filter produces much better results than simple edge detectors, i.e., discrete approximations of derivative kernels.

The fundamental property of convolution (or of linear filtering generally) is that the n-th order derivative of the convolution result of a function with a kernel is equivalent to the convolution of that function with n-th order derivative of the kernel, and vice versa. The relationship is defined as:

$$\frac{d^{(n)}(f * h)}{dx} \Leftrightarrow f * \frac{d^{(n)}h}{dx}$$

That means that deriving the classical reconstruction kernels mentioned above, we obtain so-called derivative kernels. In the 2D case, the derivative kernel is computed as a product of the derivative kernel with its functional counterpart. This results into a partial derivation in a particular direction. By rotating this kernel by 90 degrees we obtain the kernel for the next partial derivative. Higher order derivative kernels are obtained analogously to the first derivative kernel. The first derivation of the sinc filter is called *cosc* and has infinite support as well. The difference is that it exceeds the interval $[-1, 1]$ and it is an odd function.

2.7 Image processing

For fast image processing the discussed high-resolution filters are often not effective enough. Therefore several discrete, low-resolution filters were developed. There are two types that are mostly used in image processing filtering tasks. We will partly use the image processing terminology here to achieve optimal correspondence to the addressed literature. So first we define the term *linear operator* which is the image processing term for a filter of width equal to number of samples in one dimension. It could be considered as a low-resolution approximation of a particular continuous filter. Another often used image processing term is *operator mask* that means usually a two-dimensional matrix of linear operator weights. The principle of image processing filtering is illustrated in figure 2.8, where the p array represents two-dimensional sampled data and h the linear operator.

Another category of image processing operators are *non-linear operators*, which do not compute the filtered image using convolution. The algorithm usually takes one value from the operator mask according to some neighborhood conditions. The typical operators are minimum or maximum operators, taking always the smallest or biggest intensity value from the sample's neighborhood. This value is then considered as the output value of the particular pixel. Such operators are not mentioned in the following text, because they are not convolution-based.

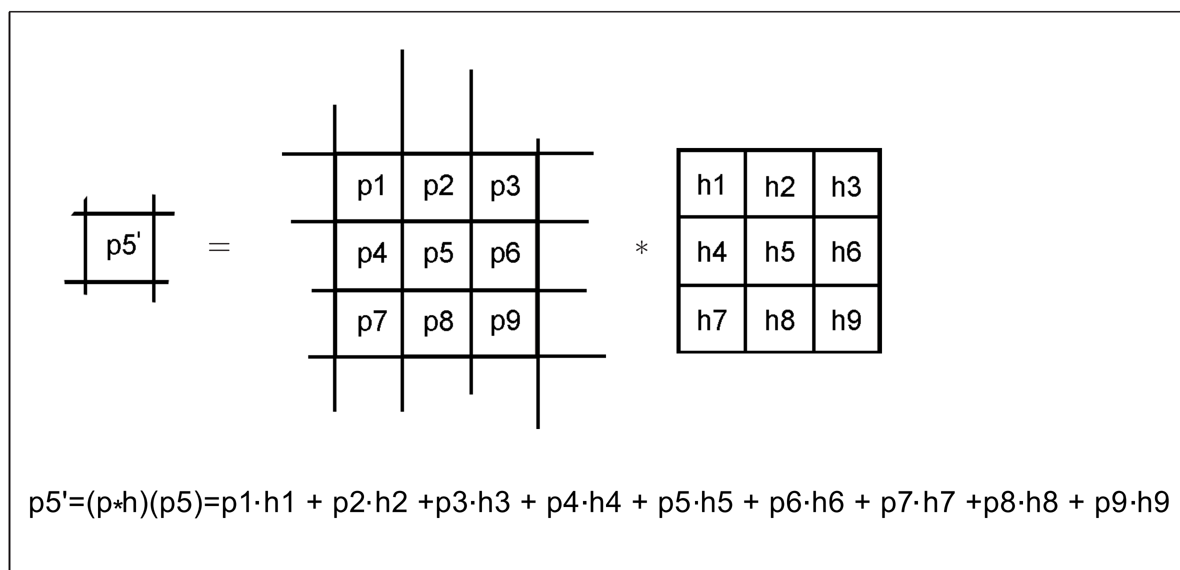


Figure 2.8: Principle of image processing filtering. The p array represents 2D data and h the linear operator. The principle shows computation of the output sample at the position $p5$.

Smoothing filters

An interesting class of linear operators are smoothing filters [34]. They are used for noise suppression exploiting the redundancy of the image data. The operator works on a sample's neighborhood given by its width. The typical representative is the averaging operator, i.e., the box filter. The operator

matrix consists of constant values, whose sum is equal to one. The resulted (smoothed) value is the sum of all (equally weighted) neighboring pixels. Figure 2.9 (a) shows an averaging operator mask.

Another often used smoothing operator is the Gaussian operator [34] shown in figure 2.9 (b). The difference to the averaging operator is that the Gaussian filter has no constant values in the operator mask. The Gaussian operator is a discrete approximation of Gaussian lobe, however generally not of a high resolution.

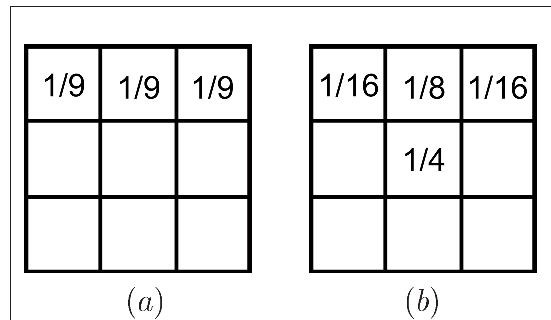


Figure 2.9: Averaging (a) and Gaussian (b) smoothing operators [34].

Edge detection

Another class of linear operators are edge detectors. These filters highlight the contours of objects, or generally highlight such places in an image where there is a big difference in neighboring sample values. This is used in segmentation or computer vision to find searched objects according to their contours. These operators also approximate their continuous counterparts. A typical representative is the Laplacian operator [34]. It approximates the sum of mixed derivatives of second order. Another representative is the Sobel [34] operator, approximating the derivative filtering of the first order. This filter usually consists of two kernels; each kernel approximates the partial derivation in one orthogonal direction. Using two operator masks makes it possible to estimate the gradient direction, not only the magnitude as in the case of the Laplacian operator. Both mentioned operators are illustrated in figure 2.10.

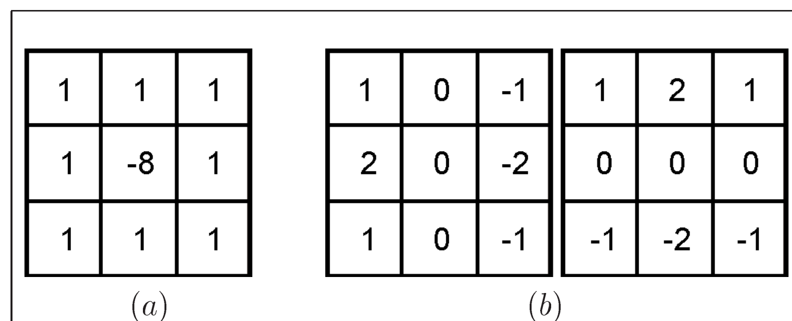


Figure 2.10: Laplace (a) and Sobel (b) edge detectors [34].

Custom filters

The previously mentioned filters are approximations of kernels for either function or derivative reconstruction. The operator mask, however, could consist of any values. For smoothing filters it is typical that the weights are in sum equal to one. This is due to the goal of intensity preservation. The derivative approximations and edge detectors usually have weights summing up to zero. Building a custom kernel, however, need not fulfill any of these conditions. Such filters are mostly used, for example, for special effects in image processing and editing applications.

Chapter 3

State of the art in reconstruction and hardware-accelerated graphics

Due to the limited support of the general filtering process by current hardware there are nowadays only few solutions for effectively performing convolution in hardware. The first section of this chapter describes existing software solutions, the next section gives an overview about current hardware features that are used in existing, hardware-based solutions.

3.1 Reconstruction filters in computer graphics

In the last chapter we have shortly mentioned the state-of-the-art techniques which are used for high-quality reconstruction. One approach is based on cubic interpolation using kernels of width four, another uses windowing functions to limit the sinc support to a finite interval.

Cubic spline kernels for reconstruction

Symmetric cubic spline filters are piecewise cubic polynomials defined on the intervals [-2, -1], [-1, 0], [0, 1], and [1, 2]. Outside these intervals the kernel is equal to zero. The effective width of these kernels is four that means in the 1D case that four sample points are contributing to the reconstructed value. Mitchell and Netravali [18] derived a family of cubic splines with two free parameters, i.e., the very popular BC-splines. The piecewise cubic BC-splines are defined by:

$$h(x) = \begin{cases} (12 - 9B - 6C)|x^3| + & if \;\; |x| < 1 \\ (-18 + 12B + 6C)|x^2| + (6 - 2B) & \\ & \\ (-B - 6C)|x^3| + (6B + 30C)|x^2| + & if \;\; 1 < |x| < 2 \\ (-12B - 48C)|x| + (8B + 24C) & \\ & \\ 0 & else \end{cases}$$

If we set the B and C parameters to particular values, the BC-splines correspond to well-known filter kernels, e.g., $B = 1$ and $C = 0$ corresponds to the cubic B-spline. Mitchell and Netravali have

investigated the characteristics of the BC-splines by changing the parameters B and C [18]. They experimentally found the boundaries of acceptable range for the B and C parameters. Another popular family of cubic splines used for reconstruction issues are cardinal splines derived by Keys [16]. These splines have one free parameter and are defined by:

$$h(x) = \begin{cases} (\alpha+2)|x^3| - (\alpha+3)|x^2| + 1 & if \;\; |x| < 1 \\ \alpha|x^3| - 5\alpha|x^2| + 8\alpha|x| - 4\alpha & if \;\; 1 < |x| < 2 \\ 0 & else \end{cases}$$

The cardinal splines are a subclass of BC-splines, the simple conversion is done setting $B = 0$ and $C = -\alpha$. The interpolative Catmull-Rom spline with $\alpha = -0.5$ (or $B = 0$, $C = 0.5$ for BC-splines) is the most suitable cubic spline reconstruction kernel [16]. Practically this means that the Catmull-Rom spline is the closest approximation of the sinc function of width four when utilizing piecewise cubic splines. Another interesting characteristic is that it is an interpolative filter, so some aliasing artifacts like blurring do not occur.

Windowed ideal reconstruction

The filters from the previous section try to approximate the ideal reconstruction filter using piecewise cubic spline interpolation. Another approach is to use a finite support approximation of the sinc function itself for reconstruction. However simple clipping of the sinc filter to a particular width usually produces significant aliasing effects due to a discontinuity of the filter. To create a band-limited version of the sinc with a smooth transition to zero, a multiplication with another function called "window" is necessary. To classify the quality of the windowing function it is necessary to examine the reconstruction function in the frequency domain. It other words, how close is the frequency response of the new filter when compared to the box filter (which is the frequency representation of the sinc function).

Turkowski [37] proposed the Lanczos window to be a suitable window for reconstruction. Theußl et al. [35, 36] compared it to other windows such as Bartlett, Welch, Parzen, Hann, Blackman, Kaiser, and clipped Gauss window. They concluded that the Kaiser, Blackman, and Gauss windows are especially suitable according to their frequency response. The quality of reconstruction using these filters depends on the width and parameter settings of the filter.

The Blackman window is defined by [35]:

$$h(x,\tau) = \begin{cases} 0.42 + 0.5\cos(\pi\frac{x}{\tau}) + 0.08\cos(2\pi\frac{x}{\tau}) & if \;\; |x| < \tau \\ 0 & else \end{cases}$$

The Kaiser window has an additional parameter α that controls how steeply the function approaches zero [35]:

$$h(x,\tau,\alpha) = \begin{cases} I_0\left(\alpha \cdot \sqrt{1-(\frac{x}{\tau})^2}\right)/I_0(\alpha) & if \;\; |x| < \tau \\ 0 & else \end{cases}$$

I_0 is the zeroth-order modified Bessel function [29]. The Gaussian window has one disadvantage in comparison to the previous two: it actually never reaches zero so we have to truncate it. The Gaussian lobe is defined by equation 2.2.

Another popular method for evaluating filters in spatial domain was introduced by Möller [19, 20]. It is based on a Taylor-series expansion of the convolution sum. The details of this method go beyond the scope of this thesis, the interested reader is encouraged to read the referenced literature [19, 20].

Analogous to function reconstruction, the derivative reconstruction uses a similar approach. However, it uses the derivatives of the above mentioned functions. This is true for both the piecewise cubic reconstruction and windowed cosc. Experiments showed that the Blackman, Kaiser, and Gaussian are also the most suitable windows for the derivative reconstruction [35, 36].

Software-based high-resolution filtering

The term high-resolution filtering addresses filtering techniques which use kernels of much higher resolution as their effective width. These filters are approximating the original continuous kernels much more realistically than their counterparts from image processing. They are mostly used in high-quality reconstruction, reasonably fulfilling the sampling theory described in the previous chapter.

The best results of high-resolution filtering are achieved with the above mentioned cubic splines and the windowed sinc filters. They produce smooth transitions without any visible artifacts. The disadvantage is that they compute the filtered value from 4 samples in the one-dimensional case, from 16 samples in 2D, and from 64 samples in 3D. Existing software solutions provide high-quality results, which require long computational times (far from real-time performance). So there is no possibility to integrate software solutions for high-quality and high-resolution filtering into interactive applications, when filtering is required on-the-fly.

3.2 Hardware-accelerated graphics

In this section we will shortly describe the basic functionality of graphics hardware, and take a look at the history to see how the various graphics boards developed during the last twenty years. We will classify them according to what they do and show the newest features which make the hardware more flexible and customizable.

The rendering pipeline

To understand what the graphics hardware does it is necessary to understand the graphics pipeline, i.e., the process that brings geometric objects onto the screen. The pipeline model classifies five main stages of operation:

- **Generation (G)**: Modeling the scene using a data structure, e.g., a BREP, a hierarchical scene-graph, or a height field.
- **Traversal (T)**: Traversing the scene data.
- **Transformation (X)**: The object coordinates are transformed to the camera coordinate system, then clipping is done according to the viewing frustum. In case of perspective projection the coordinates are divided and finally transformed to the screen coordinates. The lighting step,

which also takes part in this stage, is the calculation of lighting effects at each vertex of each triangle. This includes the color and brightness of each light in the scene and how it reacts with the color and specularity of the objects in the scene.

- **Rasterization (R)**: During this stage the objects, i.e., the polygons are being decomposed into horizontal spans of pixels (or fragments). There are also various tests taking place during this stage such as ownership test, scissors test, alpha test, stencil test, depth buffer test and blending. These tests decide which fragment should be accepted or refused according to customized settings [23].
- **Display (D)**: For the last stage in the 3D graphics pipeline, the display controller reads the information out of the frame buffer and sends it to the driver for display.

The hardware accelerators are primarily classified according to which parts of the rendering pipeline they contribute to. The traditional VGA graphics card does only perform the last stage. It is also labeled as a GTXR-D card. This label tells which parts are processed in the driver (on the left side of the dash) and which are done in hardware (right side). The first and second generations of the consumer graphics cards were of GTX-RD [38, 22] type. That means that besides bringing the scene onto the screen they were doing the rasterization stage as well. GeForce was the first GT-XRD card in the consumer area [22]. At that moment the difference between high-end solutions and consumer graphics cards became smaller and smaller. Several cards from the consumer area are also considered as low cost high-end solutions [22, 3].

The application programming interface

To be able to communicate with the hardware, several graphics libraries were developed. Using these interfaces, the programmer is able to generate the scene, sets the camera view and the lighting conditions, and does everything that somehow influences the rendering process. Two of these programming interfaces are standard nowadays OpenGL and Direct3D.

OpenGL
OpenGL (Open Graphics Library) [23] became a standard due to its definition as an open and free software. Since its introduction in 1992 by SGI, OpenGL has become the industry's most widely used and best supported 2D and 3D graphics application programming interface (API), bringing thousands of professional applications to a wide variety of computer platforms. The OpenGL is designed by various manufacturers adding more and more features to the standard set. Another feature of OpenGL is the portability of its code. Only one characteristic can be considered as a drawback, which is its programming level. It is designed for the C programming language. Consequently, OpenGL does not contain any hierarchy of classes. It works as a state machine. This problem is partly solved in the Open Inventor API (SGI), i.e., an object-oriented layer on top of OpenGL. This API unfortunately is bound only to the official versions of the OpenGL standard, so there is no possibility to use the newest hardware features by using Open Inventor. However, it is possible to mix Inventor and OpenGL code. Due to its stability and portability OpenGL is widely used in professional applications and research.

Direct3D
Direct3D is another standard for the PC platform, based on the Windows operating system. It is a

part of a multimedia framework called DirectX [11] (developed by *Microsoft Corporation*). It is designed as an object-oriented graphics library within the multimedia framework. Direct3D is often used in game industry.

The pioneers from Silicon Graphics

The history of using special hardware for accelerating the graphics pipeline started in the early eighties, when *Silicon Graphics Inc. (SGI)* [31] designed their first machine. Floating-point hardware was just becoming available at reasonable prices, the framebuffer memory was still quite expensive and the application-specific integrated circuits (ASICs) were in fact not available at all. The resulting machines had reasonable transformation capabilities, but very limited capabilities for processing the framebuffer. In particular, smooth shading and depth buffering, which required substantial framebuffer hardware and memory, were not available at interactive rates. Therefore the target capabilities of first-generation machines were the transformation and rendering of flat-shaded points, lines, and polygons. These primitives were not lighted, and hidden-surface elimination, if required, was accomplished by software. Typical representatives of such systems were the Silicon Graphics Iris 3000 (1985) and the Apollo DN570 (1985) [31].

Towards the end of the first-generation period new advances in technology allowed lighting, smooth shading, and depth buffering to be implemented, but only with an order of magnitude less performance than what was available for rendering flat-shaded lines and polygons. Therefore the target capability of these machines remained the same as for the first-generation machines. The Silicon Graphics 4DG (1986) was an example of such an architecture [31].

Because the first-generation machines could not efficiently eliminate hidden surfaces, and could not efficiently shade surfaces even if the application was able to eliminate them, they were more effective at rendering wire-frame objects than solids. Beginning in 1988, the second-generation of graphics systems, primarily graphics workstations, became available. These machines took advantage of reduced memory costs and the increased availability of ASICs to implement deep framebuffers with multiple rendering processors. These framebuffers had the numeric ability to interpolate colors and depths with little or no performance loss, and the memory capacity and bandwidth to support depth buffering with minimal performance loss. They were therefore able to render solids and full-frame scenes efficiently, as well as wire-frame images. The Silicon Graphics GT (1988) and the Apollo DN590 (1988) are early examples of the second-generation machines [31].

Later second-generation machines, such as the Silicon Graphics VGX, the Hewlett Packard VRX, and the Apollo DN10000 included texture mapping and anti-aliasing of points and lines, but not of polygons. Their performances are substantially reduced when texture mapping is enabled. The texture size (of the VGX) and filtering capabilities (of the VRX and the DN10000) are limited [31].

The third-generation design was introduced by RealityEngine graphics in 1993 [2]. Its target capability is the rendering of lighted, smooth shaded, depth-buffered, texture-mapped and anti-aliased triangles. The initial target performance was 500000 triangles per second, assuming the triangles are in short strips, and 10 percent intersect the viewing frustum boundaries [2]. Textures were to be well filtered (8-sample linear interpolation within and between two mipmap levels) and large enough (1024×1024) to be usable as true images, instead of small textures tiled on the geometry. Anti-aliasing produced high-quality images of solids, and worked in conjunction with depth-buffering,

meaning that no application sorting was required. Pixels were filled at a rate sufficient to support 15–30 Hz for most applications.

The successor of the RealityEngine system was InfiniteReality introduced in 1997 [21]. The fundamental design goal of this graphics system was to deliver real-time performance to a broad range of applications. This system was able to render full-screen images at a constant frequency of 60 Hz, so the goal to satisfy the needs of flight- or ground-based simulations as well as virtual reality applications was achieved quite well. The framebuffer size increased to 1280×1024. Another important feature was the possibility to access a "virtual texture memory" without significantly impacting the overall performance. The programming interface to InfiniteReality as well as its precedessors is OpenGL. These graphics boards are classified to be of GT-XRD type.

Silicon Graphics Inc. was the leader on the market of graphics systems for a long time. Typically, they provided the graphics engines mostly for their own MIPS RISC workstations such as Onyx or Onyx2. However, their production started a PC line as well.

Consumer graphics hardware

The importance of the PC platform in the 3D graphics area grew in the last few years by porting a lot of professional tools for 3D modeling and animation to this platform. One of the representatives that tried to achieve the real-time performance of this tools on the PC was *3DLabs* [39].

In the mid-eighties the computer games market, producing more and more realistic games, started to look for hardware-accelerated graphics as well. The companies participating in the first generation of consumer graphics hardware started to produce chips that were not supporting 2D graphics at all. These were exclusive 3D boards and could only be used in conjunction with conventional 2D graphics cards. Although this boards were already equipped with texture memory and a framebuffer the practical usage was limited to playing games. The graphics applications like tools for 3D modeling and animation, were usually not able to take advantage of these early consumer 3D hardware accelerators. The leading company producing the well-known Voodoo boards was *3dfx Interactive* [38]. These cards can be considered as being of GTX-RD type.

In 1998, *NVidia Corporation* [22] was the first competitor of 3dfx that was able to come close in terms of performance, and even surpassed the line of Voodoo Graphics accelerators in terms of features with its Riva TNT graphics accelerator. This chip started the second generation of consumer graphics accelerators. The Riva TNT was the first consumer graphics chip that was able to offer high-performance and high-quality rendering, also supporting OpenGL. In contrast to the Voodoo Graphics, these boards were combined 2D and 3D graphics boards, so they required only a single slot for a full 2D and 3D acceleration. Riva TNT is significantly different to the Voodoo graphics boards but it is still of GTX-RD type.

With the very popular Riva TNT chip NVidia started to merge two application areas using 3D hardware: computer games and professional applications. The following product confirmed this strategy even more. In 1999 NVidia introduced GeForce 256 graphic chip. Introducing the GeForce 256, NVidia started the production of real GT-XRD cards on the PC level [22]. Previous accelerators basically all were rather simple rasterizers, in some way more 2D than 3D, where most of the 3D work had to be done by the driver. The new card fully computed the transform and lighting in hardware. The biggest importance from the programmer's side of view is not only that OpenGL 1.2 is fully supported, but it offers a lot of features and additions to the standard, making the hardware much

more flexible. The features are mostly dealing with the texture mapping hardware. Among them, for instance, hardware-based bump mapping and support of cubic environment maps were available. A feature, which is of biggest importance with respect to flexibility are the Register Combiners as discussed below. A year later, NVidia introduced the GeForce2 chip, equipped with more memory and being faster, but the features remained more or less the same [22].

In 2001, NVidia introduced the GeForce3 graphics chip to the market. The rendering speed increased significantly. The vertex and pixel shaders were added to the hardware implementation of the shading languages. Hardware volume rendering became possible using 3D textures instead of a stack of 2D textures. Multi-texturing became more powerful using four texture units in one rendering pass. The game and animation production companies are also making use of shadow buffers, adding more realism using cinematic-style shadow effects. Recently, NVidia introduced their newest chip called GeForce4. It features the same amount of texture units per single rendering pass, but the performance increases in some cases by one magnitude. The chip features improved vertex shading and a new anti-aliasing implementation. However the pixel shader features are still limited by the Register Combiners' flexibility.

Parallel to the GeForce production, NVidia's biggest competitor, i.e., *ATI Corporation* [3] developed another line of graphics chips, equipped with similar features as the GeForce, called Radeon. Starting the production of the Radeon family boards, ATI became the second company developing consumer hardware accelerators equipped with a lot of additional features giving the developer broad application possibilities. Radeon was the first consumer graphic card supporting 3D textures.

Its successor, i.e., the last chip of ATI called Radeon 8500, is able to use up to six texture units in a single rendering pass. It supports 1D, 2D, and 3D textures as well. Another important feature in terms of quality is its internal precision. Radeon 8500 features 12 bit precision, 3 bits more than the GeForce3. However, this chip is still significantly slower than the GeForce3. The card is equipped with both vertex and fragment shader (another term for pixel shader). A more detailed description, as well as a comparison to GeForce3 is discussed in section 3.3.

An open question are improvements in the framebuffer and the internal precision, which is still a big limitation in many application areas. As there are two leading companies on the market of consumer graphics cards we can expect a fast development towards higher speed and a wider spectrum of applications.

Special-purpose hardware

A completely different approach was used in the design of another graphics board by *Mitsubishi Electric* introduced by Pfister et al. [28]. The rather expensive board called VolumePro is designed as a PC-card for a PCI slot. The card was primarily intended for hardware-based volume rendering using a ray casting algorithm [17] and parallel projection at framerates up to 60 Hz. The new versions of the card mainly differ in bigger memory allowing to process bigger datasets. The most significant limitation of this hardware is the narrow range of application. The card was designed for doing volume rendering, it is rarely used for anything else.

3.3 Relevant features for hardware filtering

The feature-set of consumer graphics hardware has reached a level that offers a lot more than just simple scene rendering. Beside the astonishing performance improvements, the programmer is able to significantly customize the rendering process according to his or her needs. The following features are described which are necessary for hardware-based filtering. We shortly sketch what they are used for especially in our context. A more detailed explanation of their usage is given in the following chapters 4, 5, and 6.

Multi-pass rendering

The idea of multi-pass rendering is to perform more rendering passes during the main loop instead of transferring the content of the framebuffer to the display device immediately after each pass. The results of each pass are combined together according to a blend function. The simplest operation is addition of the new results to the existing content of the framebuffer. It is possible to set the blend equation to subtraction or to other operations that use the alpha-channel content of the framebuffer and/or the last rendering pass to decide how pixel values are mixed together. Operations which are dependent on the alpha channel content of one or both operators (new results and existing framebuffer content) are so-called alpha-compositing operations. A typical example of multi-pass rendering is to render the same object in more passes with different material and lighting settings to achieve a more realistic appearance. However, multi-pass rendering has its limitations related to the framebuffer precision which is still 8 bit per channel. Anther important feature is to copy the content of the framebuffer to a texture and to re-use the subresult for the next processing. Nowadays, hardware features also “virtual framebuffers” to avoid the time-consuming reading from the framebuffer. The fastest approach to store rendering subresults is to render directly to the texture.

Multi-pass rendering is the most important feature for hardware filtering, allowing to compute the final pixel value from subresults of more passes.

Multi-texturing

The multi-texturing approach is similar to multi-pass rendering, allowing to process more textures in a single pass. The number of textures is limited by the number of texture units provided by the current hardware. The advantage of multi-texturing when compared to multi-pass rendering is usually a higher speed but also the fact that the internal precision of the hardware is often better than the framebuffer precision. This practically means that the result of the computation is not influenced that much by the precision. The better quality of multi-texturing as compared to multi-pass rendering is not the only difference in functionality. Multi-pass rendering is generally intended for blending of two arbitrary scenes, while multi-texturing allows to use more textures (and operations among them) on the same geometry within a single rendering pass.

The old multi-texturing model was quite hard-wired, the programmer was strongly limited by the exact ordering of the texture units during the multi-texturing pipeline. Thanks to features of nowadays hardware there are more or less no limitations in the texture order anymore.

For high-resolution filtering also the multi-texturing feature is fundamental. The reason is that the object to be filtered is stored in one texture and the filter kernel in another. For multiplying the kernel with the source at least two textures are necessary.

Programmable shaders

Programmable shaders are a powerful way to describe the interaction of surfaces with light. A well-known shading language is, e.g., RenderMan introduced by Pixar [12]. The RenderMan shading language is very flexible in managing the rendering process, however, it is still a software solution. As graphics hardware nowadays evolves beyond the traditional inflexible pipeline, hardware designers are looking for programmable models to support the next generation of real-time applications. The shader operates on each vertex or pixel individually, so it is possible to customize the rendering pipeline for each element individually. Of course, some parts of the scene still can be rendered with the fixed rendering pipeline to increase performance. There are two types of shader, i.e., vertex and pixel shaders. Although they seem similar from the end-user side of view, the concepts are rather different. The vertex shader is a part of the transformation stage, while the pixel shader controls the rasterization stage.

A vertex shader can be considered as a black box with vertex data being fed into the box and different vertex data coming out. Every vertex that goes in comes out once, but it may have changed while it was in the vertex shader. Consequently, vertex shaders do not create or delete vertices, but simply operate on them, change the values of the data which describe the vertex. The outgoing vertex can have a different position in space, changed color, transparency, or texture coordinates as compared to the input. All of these vertex changes, computed for one vertex at a time, can create a special effect for the overall object.

Analogously to the vertex shader, pixel shaders (or fragment shaders) operate on pixels. The principle of using pixels shaders, however, is similar to vertex shaders. Replacing the fixed graphics pipeline, the programmer can better influence the appearance of materials like simulating facial hair, producing various procedural effects, bump mapping, or Phong shading.

Comparing the two leading companies on the PC market, i.e., NVidia and ATI, ATI offers more flexibility with respect to pixel shader with their ATI Radeon 8500. Even the newest NVidia GeForce4 chip does not change this fact. There are still limitations in the texture unit order, the programmer is restricted to use only a hard-wired ordering.

For the hardware-based filtering methods programmable shaders are a tool for comfortable texture and color processing. It also allows to add the result of one multiplication of two operands (textures and/or colors) to another, before storing the resulting values in the framebuffer.

3D textures

The classical approach of texturing an object deals with surfaces. To each vertex a texture coordinate is assigned referring to the underlying source image (texture) which simulates the surface characteristics of the material. Slightly in contrast, the real world objects are no surfaces but three-dimensional objects. Consequently the characteristic of real-world objects are approximated by 3D functions instead of 2D ones. Classical representatives are marble or wooden objects, which are hard to render

realistically when 2D textures are used. There are a lot of visible artifacts appearing with 2D textures. For example the visible texture repetition or the fact that two textures do not perfectly fit together on the edges of a particular object. This is easy to solve by using a 3D texturing approach. Another application area are volumetric objects like fog or smoke, where the traditional 2D texturing must involve a lot of tricks to realistically simulate such objects [26, 27]. 3D texturing is also necessary in volume visualization, like visualizing tomographic or magnetic imaging scans [30, 40]. There are already approaches to simulate 3D data with a stack of 2D slices, but the implementation is rather complicated.

3D textures are necessary for hardware filtering when filtering 3D objects. This is mostly the case when visualizing medical data or objects textured by solid textures.

OpenGL imaging subset

The OpenGL imaging subset is an extension of OpenGL and allows a simple convolution between the source image and a kernel [23]. Unfortunately it does not provide at all means for high-resolution filtering. It is designed for the basic needs of image processing applications like smoothing or edge detection. The computation is not done in texture hardware, it works with the basic pixel operations set. Using this feature is not very effective, because the kernel size is limited by the hardware and only one or two-dimensional convolutions are supported natively. The processing speed on consumer hardware is also rather confusing, the computation is probably implemented in the driver.

Due to its limitations, this feature was not used in our implementation, it was used as quality and performance reference.

3.4 Hardware-based filtering

The only filter supported natively in hardware for high-resolution filtering is the tent filter, i.e., for linear interpolation. This is a sufficient solution only in the case of high sampling frequency or minification during viewing, otherwise it produces significant artifacts. The big advantage of natively supported linear interpolation is the real-time performance required by interactive applications. Nowadays, graphics hardware allows to filter linearly also 2D and 3D functions on-the-fly. The resulting performance depends, of course, on the filtered texture size and the mesh complexity as well.

Although current graphics chips are equipped with a lot of features, one problem still remains, i.e., the insufficient precision range of the current hardware. Therefore there are just few approaches to use the hardware for filtering. Hadwiger et al. [8, 9, 10] has proposed an hardware-based approach exploiting texture hardware features for high-resolution high-quality filtering tasks. This thesis is based on their approach, which is described in chapter 5. James [15] designed an image processing approach for simple convolution tasks using the texture hardware as well. This method is also used in our implementation and is described in chapter 6.

Beside these methods another approach was proposed by Hopf and Ertl [13] using the SGI graphics hardware [31]. They are using the OpenGL imaging subset of OpenGL and texture hardware to extend the natively supported 2D convolution to 3D [13]. A year later they introduced hardware-accelerated non-linear image processing with erosion and dilation operators [14].

Chapter 4

Hardware-based filtering using textures

> There is nothing easier than complicate the job, however nothing is more difficult than making the job easier.
>
> *Meyer's Law [4]*

In the previous chapters we have introduced the basic filtering operation, i.e., convolution. The usual (standard) implementation is based on the *gathering principle*. This means that the final value of a particular output sample is computed by gathering weighted neighboring input samples and adding them simultaneously. In other words, the convolution operation is processed in a serial way, i.e., one output sample after the other.

The gathering principle is the straightforward way to implement convolution and it can take advantage of parallelizing the process. Processing more output values in one time – on a multiprocessor system or on a network workstation farm – can increase the performance almost linearly. The parallel algorithm can divide the image or volume into several parts, which are then processed each on a separate processor. After each subpart is filtered, the final result is composited from the subresults.

This "classical" implementation is illustrated in figure 4.1(a). The tent filter results in a portion from the first sample summed with a corresponding portion of the second one. On a single processor system this kind of implementation is this especially efficient, when only one or just few particular output values are computed.

4.1 Distribution principle

The gathering principle as discussed above is the most intuitive approach, it is easy to implement in software. However there is another possibility of how to achieve filtering using another evaluation order. Instead of computing one output value after another, we can compute the contribution of particular input samples to all corresponding output values at a time. This looks at first quite complicated and usually there is no reason to use this principle in software implementations.

The distribution principle is illustrated in figure 4.1(b). For comparison to the gathering principle we use the tent filter again. The input sample distributes its partial contributions to a given resampling grid. In case of a tent filter of width two, we sum these partial contributions with contributions of

another input sample and get the resulting resampling points. The idea of reordering the evaluation of the convolution sum is used in all splatting-based techniques [41].

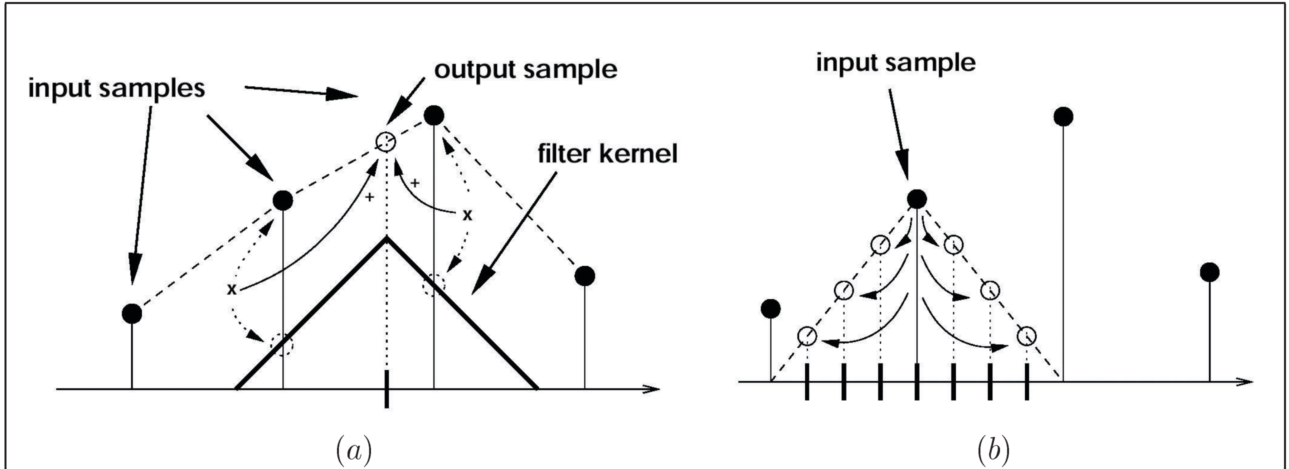


Figure 4.1: The Gathering principle (a) computes the output value at a time, while the distribution principle (b) distributes the input sample to all corresponding output values.

4.2 Distribution principle in hardware

The reason, why we discuss the distribution principle at all, is its power when we use graphics hardware in the filtering process. A graphics chip is designed to perform primitive mathematical per-fragment operations for all fragments (almost) simultaneously. This does not mean that it is in parallel. This depends the internal hardware implementation, however, due to a powerful performance it can be considered to be parallel from the outside. The most flexible part of the graphics chip is the texture mapping hardware. Operations like addition or multiplication of RGBA channels are the most simple representatives of the instruction set. This is in fact all what a simple filtering process needs.

Lets take a look at the distribution principle once more to understand its hardware version. The input function is stored in one texture and the filter kernel in another one. The kernel is thus also stored in sampled form. However, the number of samples can vary according to the quality of sampling. It is obvious that using a high sampling frequency of the kernel increases the quality of the resulting filtering operation. The kernel texture is scaled to the texel size of the input texture times the kernel width. Kernel width describes the number of contributing input samples. For example the tent filter computes output values from two input samples in the 1D case that means it is of width two, while the cubic spline kernels are of width four. To be able to perform the same operation for all input samples in one time, we have to divide the kernel into several parts to cover always only one input sample at a time. Let us call such parts *filter tiles*. For illustration we use the tent filter again. As mentioned above this filter is of width two. Instead of taking the whole filter kernel, we first only take the left tile of it. This tile is scaled exactly to the width between two input samples. To compute the “left” contribution of each input sample we shift the input function texture to the left by one half of the sample distance. After this, each input sample covers exactly the left tile of the filter kernel. Now we

have the input function in one texture and the kernel tile (of width one) in the second one. We take advantage of another hardware feature, i.e., texture repetition. The kernel texture is repeated to cover the whole input function. We set the numerical operation between these two textures to multiplication and render it to the framebuffer. This subresult is the left tile contribution of each input sample to the output resampling points. We repeat this process with the right tile – shift the input texture to the right by one half of the sample distance from its original position – set the framebuffer blending function to addition and render the other contribution to the framebuffer again. The whole filtering result is now stored in the framebuffer. The distribution of the left and right tile contribution is illustrated in figure 4.2.

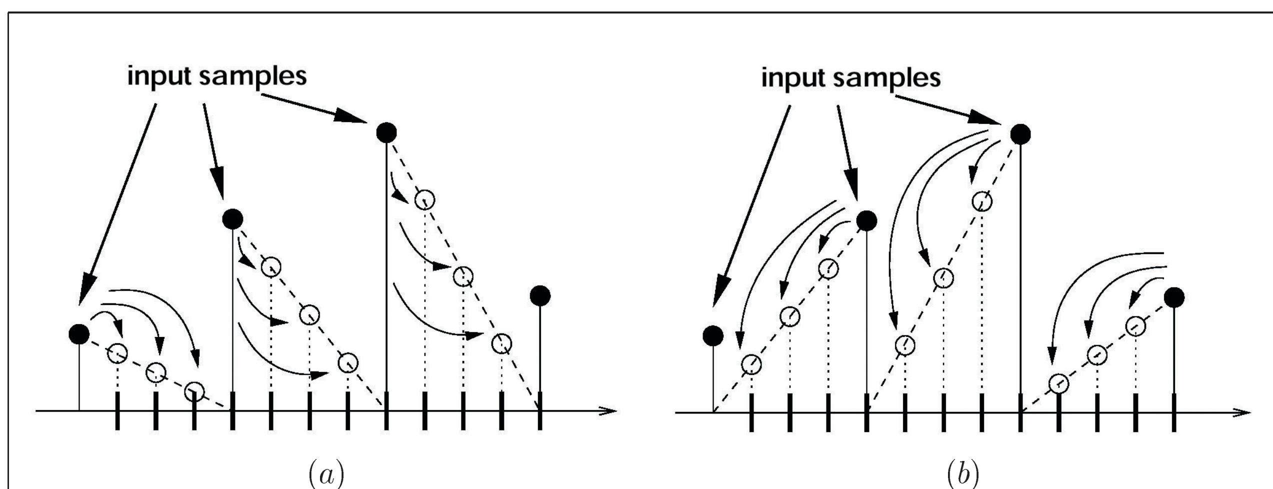


Figure 4.2: Distribution of left (a) and right (b) contributions on the resampling points. The distributed portions of the left and right contributions are summed together to get the correct values at the resampling points.

The 1D tent filter example explains the general filtering approach using arbitrary filter kernels. For better illustration we explain the filtering idea with a pseudocode dividing the process into the following logical parts.

Global setup

In this part, the first important task is the filter kernel computation and its division into several texture tiles of a width equal to the distance between two samples. The next task is to generate lookup tables that completely describe the filtering process (see section 7.2 for a detailed explanation). The lookup tables assign shifting coordinates and filter tiles to each filtering pass. The shifting coordinates are analogous to the example of 1D filtering using the tent filter as discussed above. They are used to shift the input texture to cover the correct image part with the corresponding filter. However, in the 2D and 3D case with wider kernels the situation is a little bit more complicated. There are shifts not only half the sample width in each direction, but 16 or 64 various shifts to assign a correct sample neighborhood to the corresponding filter tile. Section 7.2 describes the filtering process from the implementation point of view and illustrates the shift lookup tables as well. In the case of symmetric filter kernels, we can generate one filter tile from another one by simple mirroring operation. Another lookup table describes how to mirror and which tile to use for a particular filtering pass. This lookup

table is generated in this stage as well. Another part of the global setup is an optional software simulation of the filtering process to see, whether there are any complications in the numerical precision using hardware filtering.

Per-pass setup
This part of the filtering process selects a filter tile for the respective filtering pass. This tile is assigned to a texture unit, similar to the source texture. The only difference is in the texture setup. While the source texture is set to clamping, the filter texture is set to repetition. This is used to be able to filter each sample value of the input texture at once with the same filter tile.

Vertex setup
The name of this stage comes from the hardware part, which is used in this stage. Vertex shaders are used to shift the texture coordinates of the input texture according to the filtering pass so that the input sample is multiplied with the corresponding filter tile. Beside this, it scales the filter tile texture to the size of the input sample distance. After this, when the texture is set to repetition from the previous filtering stage and the input texture is correctly shifted, each filter tile replica should cover exactly one input sample.

Pixel setup
The final stage is also called according to the hardware part where it is processed, namely the pixel shader. It assigns both textures to shader registers and computes the multiplication of these two registers. The product of the multiplication is a set of outgoing fragments which represent the subresult of a particular filtering pass. This subresult is then stored in the framebuffer. The blend function must be set to addition to sum all the filtering passes together to achieve the desired filtering.

We will use pseudocode in the next chapters to illustrate various issues of the implementation. The general filtering approach could be described by the pseudocode in table 4.1. The pseudocode uses the operator *shift_j*, denoting that the input texture is not actually indexed differently at each output sample, but instead the input texture is shifted according the the particular rendering pass j.

4.3 High-resolution filtering vs. image processing

The filtering process as described above, i.e., linear interpolation with the distribution principle in fact is resampling not reconstruction. The difference is that reconstruction generates a continuous function from the sampled data, while resampling produces another sampled function that represents the same continuous function as the original sampled data. So reconstruction can be viewed as the first part of resampling. The second part is then the sampling process with another sampling frequency.

Another filtering approach is used in image processing and convolution-based operations. In general, image processing only affects already existing samples. The so-called linear operators are neither reconstruction nor resampling filters at all, they do not generate any new resampling points. The approximation of the continuous filter is very rough, because the convolution mask has the same number of samples as its width.

The difference between a 1D continuous signal, a high-resolution filter, and the image processing operator is sketched in figure 4.3. The original Gaussian lobe often used as smoothing operator in image processing applications is represented by three or five samples in each dimension (figure 4.3 (c)).

global setup:	
input texture:	*as is, all formats (mono, RGB, RGBA, etc.);*
filter tiles:	*generate 1D, 2D, or 3D tile textures;*
per-pass setup:	
input texture:	*setup texture clamping;*
filter tiles:	*select tile corresponding to pass j;*
	setup tile repetition;
vertex setup:	
input texture:	texcoord[TEX_UNIT0] = shift_j(texcoord[TEX_UNIT0]);
filter tiles:	texcoord[TEX_UNIT1] = texcoord[TEX_UNIT0]*tile_size;
pixel setup:	
input texture:	reg0 = SampleTex(TEX_UNIT0);
filter tiles:	reg1 = SampleTex(TEX_UNIT1);
	out = Multiply(reg0, reg1);

Table 4.1: General filtering approach divided into several stages. This approach is denoted as *std-2x* algorithm in chapter 5.

Each sample represents one weight for the neighboring samples. Comparing this to high-resolution filtering (figure 4.3 (b)), where one tile of the filter kernel is represented by usually a high number of samples, the continuous kernel (figure 4.3 (a)) is better approximated. In the case of image processing, we are dealing with only one sample pro tile. This means that the filter tile is constant throughout singular tiles. When we take a look at the general filtering approach in hardware in case of linear operators, we will multiply the input texture with the kernel texture of a constant value. This is equivalent to multiplying it with a color value. Instead of setting the color always to white we use the color to represent the convolution weights. This saves us some texture traffic time – caused by the kernel textures transfer – and texture units to be able to process more filtering passes in a single rendering pass. This difference in the filtering concept is the reason why we discuss high-resolution filtering and image processing in separate chapters 5 and 6.

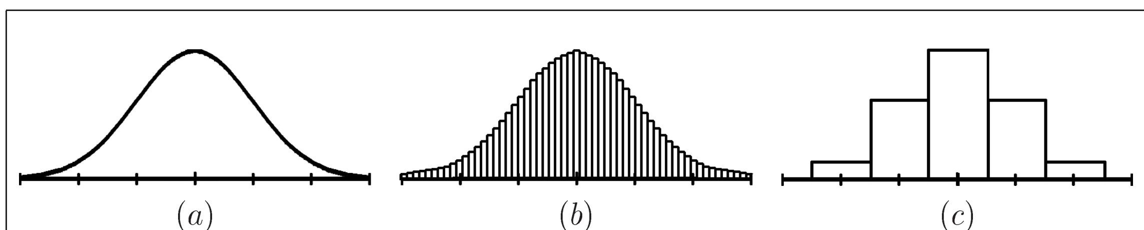


Figure 4.3: Gaussian lobe: continuous function (a), high-resolution sampling (b) and image processing Gauss filter (c).

Chapter 5

High-resolution filtering

In the previous chapter we have introduced hardware-based filtering as an approach which is based on basic texture operations. It was explained using a simple example of linear interpolation. In this chapter we will describe high-resolution filtering in more detail, showing the filtering approach based on filters of width four, i.e., piecewise cubic spline filters and windowed sinc using the Kaiser window, known from previous chapters. Although we use kernels of width four, the filtering approach works for kernels of arbitrary width. A width of four provides on current graphics cards the best quality/performance ratio.

High-resolution filtering deals with continuous filters which are sampled using a variable sampling frequency, however assuming, that the number of samples is much higher than the filter width. To involve the hardware into the filtering process it is necessary to split the filter kernel into several tiles according to its width. The tiles are then stored in separate textures. As we are dealing with filters of width four, a one-dimensional filter is divided into four texture tiles. The advantage of the filters we use for reconstruction is that they are separable in higher dimensions. That means, e.g., the two-dimensional kernel can be computed as a cartesian product of two one-dimensional kernels. The filters that we use have also another interesting characteristic of reconstruction kernels, i.e., symmetry. This allows to (implicitly) assess one part of the filter kernel by mirroring another part. In our implementation we use three types of reconstruction kernels: a cubic B-spline, a Catmull-Rom spline, and a windowed sinc with a Kaiser window. All these filters are separable and symmetric as well. This is important to know when discussing various filtering algorithms in the next section.

5.1 Filtering algorithms

The filtering process based on the concept from the previous chapter uses in fact two textures, one for the sampled input function and another for the sampled filter kernel. In case when the hardware allows to use more than two textures in a single rendering pass, we can perform more than one filtering passes within a single rendering pass. This has two significant advantages. First, due to a higher internal precision, the computation does not suffer from the quantization artifacts that much. The second advantage of using more textures in one rendering pass is of course the increased performance. In some cases the performance increases almost linearly.

Although quantization artifacts are usually not so visible at a first glance, they immediately become visible when changing from four to two textures. The human observer recognizes a grid-like

structure over the resampled function. Using four textures means that there are two filtering passes within one rendering pass summed together internally in hardware and this sum is added to the framebuffer. Using only two textures there is one filtering pass per one rendering pass and all filtering contributions are summed in the framebuffer.

The algorithms described below can be expanded to use more texture units if available. The algorithms are illustrated using pseudocode to clearly see the differences between them. We use the same style of pseudocode as in the previous chapter (see table 4.1).

Standard filtering

In this case, no optimization is done in order to make use of kernel characteristics like symmetry or separability. It implies that the tile textures have the same dimensionality as the input sample function. This approach has the highest demands on texture memory as well as the highest texture traffic.

The first task of the standard filtering algorithm is the computation of the filter kernel and its division into several texture tiles of a width equal to the distance between two samples. Next, the lookup tables are generated to manage the filtering process. The first lookup table determines the pass order to be able to easy change the processing order. The second lookup table assigns the filter tile to the particular filtering pass and the third table stores shift coordinates for each pass. The shift coordinates are used to shift the input texture to cover the correct texture part with the corresponding filter tile. All this is done in software as a preprocessing step. The per-pass stage selects a filter tile for the respective filtering pass. This tile is assigned to one texture unit, and the source texture to another. While the source texture is set to clamping, the filter texture is set to repetition to multiply all source texels at once. After this is done, the source texture is shifted according to the filtering pass and the filter tile texture is scaled to the size of the input sample distance. The input and filter tile textures are multiplied together and the result is stored in the framebuffer. This is repeated for all filtering passes to compute all contributions which are summed together in the framebuffer.

The algorithm is denoted as *std-2x*, while “std” means standard and the “2x” notation describes the number of texture units used. Extended versions are called *std-4x* and *std-6x*, depending on hardware capabilities. It works for 1D, 2D, and 3D convolutions as well.

Symmetric filter kernels

The symmetry of a filter kernel can be exploited such that some tiles of the filter kernel are generated by mirroring others. We can make use of symmetry both across axes and diagonals. When the filter kernel is not separable or the hardware does not have enough texture units in order to exploit separability, the symmetry property highly reduces the texture memory consumption. In the 2D case, using a kernel of width four in each dimension, we only need to store three out of 16 texture tiles. Even more effective, in the 3D case only four out of 64 texture tiles are stored, saving more than 93% of memory resources as compared to the standard case. Another optimization is to save the kernel texture in compressed form. Although this causes an additional significant reduction of memory consumption the hardware needs some time for the decompression.

It is important to note that mirroring a texture tile is not as simple as just swapping texture coordinates, mainly because mirroring needs to be pixel-exact. Although on some hardware it is often

sufficient enough to mirror the texture coordinates in the vertex shader, it is generally not enough to guarantee the pixel-exactness that means that the filter tile texture covers the source texture exactly for each pixel. In fact the vertex shader guarantees only the exactness on a vertex basis. Therefore it is necessary to mirror the tiles in the pixel shader.

The algorithm which exploits filter symmetry is derived from the general filtering approach. The only difference is in the per-pass setup. This algorithm does not increase the overall performance of the filtering process, and it can be combined with any other filtering algorithm, therefore we do not use any special notation for this kind of optimization. Table 5.1 shows this process expressed using pseudocode.

global setup:	
input texture:	*as is, all formats (mono, RGB, RGBA, etc.);*
filter tiles:	*generate 1D, 2D or 3D tile textures;*
per-pass setup:	
input texture:	*setup texture clamping;*
filter tiles:	*select tile corresponding to pass;*
	setup tile mirroring;
	setup tile repetition;
vertex setup:	
input texture:	texcoord[TEX_UNIT0] = shift_j(texcoord[TEX_UNIT0]);
filter tiles:	texcoord[TEX_UNIT1] = texcoord[TEX_UNIT0]*tile_size;
pixel setup:	
input texture:	reg0 = SampleTex(TEX_UNIT0);
filter tiles:	reg1 = SampleTex(TEX_UNIT1);
	out = Multiply(reg0, reg1);

Table 5.1: Filtering with the *std-2x* algorithm exploiting kernel symmetry.

Separable filter kernels

Exploiting another characteristic of filter kernels, i.e., separability, we can decompose a high-dimensional kernel into several ones of lower dimensionality. Storing lower dimensional kernels instead of higher dimensional ones significantly reduces memory consumption and texture traffic. Instead of using 2D or 3D filter kernels, we assess them as two or three one-dimensional ones. In the 3D case there is another possibility: in order to save texture units involved in the filtering computation the texture is stored in one one-dimensional and one two-dimensional texture. These textures are multiplied on-the-fly in the pixel shader to generate the corresponding 2D or 3D filter kernel tiles. It is obvious that instead of, e.g., one two-dimensional texture with 64 samples in each direction only one or two one-dimensional textures with 64 samples in total are stored. We call this algorithm *sep-3x*, expressing by “sep” the separated computation and by “3x” the number of texturing units used in the filtering algorithm for 2D as well as 3D case. There are always two texture units reserved for the

decomposed filter and one texture unit for the input sample function. In the 2D case both texture units represent the one-dimensional textures, while in the 3D case we represent one-dimensional texture with one texture unit and the two-dimensional texture with another one. These two textures are multiplied at first, generating the corresponding filter tile which is used in the convolution with the input sample function.

Although in theory the *sep-3x* algorithm is numerically equivalent to the *std-2x*, this is not completely true in practice. The *std-2x* algorithm uses kernels which are generated in a preprocess in software using floating-point precision, while in the pixel shader on-the-fly computation uses only the internal precision given by hardware. The comparison between the above mentioned algorithms is definitely a performance/quality trade-off. The quality of the *sep* algorithm significantly increases when using its *sep-6x* version. Table 5.2 describes the filtering approach described in this section.

global setup:	
input texture:	*as is, all formats (mono, RGB, RGBA, etc.);*
filter tiles:	*separable; only 1D (3x [2D] / 4x [3D]), or 1D and 2D (3x [3D]);*
per-pass setup:	
input texture:	*setup texture clamping;*
filter tiles:	*select tiles corresponding to pass;*
	[*setup tiles mirroring;*]
	setup tiles repetition;
vertex setup:	
input texture:	texcoord[TEX_UNIT0] = shift_j(texcoord[TEX_UNIT0]);
filter tiles:	texcoord[TEX_UNIT1] = texcoord[TEX_UNIT0]*tile_size;
	texcoord[TEX_UNIT2] = texcoord[TEX_UNIT0]*tile_size;
/* *sep-4x* */	texcoord[TEX_UNIT3] = texcoord[TEX_UNIT0]*tile_size;
pixel setup:	
input texture:	reg0 = SampleTex(TEX_UNIT0);
filter tiles:	reg1 = SampleTex(TEX_UNIT1);
	reg2 = SampleTex(TEX_UNIT2);
/* *sep-4x* */	reg3 = SampleTex(TEX_UNIT3);
/* *sep-4x* */	reg2 = Multiply(reg2, reg3);
	reg1 = Multiply(reg1, reg2);
	out = Multiply(reg0, reg1);

Table 5.2: Filtering with *sep-3x* or *sep-4x* algorithm and exploiting the separability of the filter kernel as well as symmetry (optional).

Separable and symmetric filter kernels

When the filter has both properties, symmetry as well as separability, we are able to combine both above mentioned algorithms. Typically, all kernels used for high-quality reconstruction exhibit such

properties. The pseudocode in table 5.2 also includes the symmetry extension (illustrated with angle brackets in the per-pass setup). When applicable, the process of per-pixel mirroring is performed on the decomposed kernels. When these are mirrored on a per-pixel basis the result of the multiplication of the two or three decomposed lower-dimensional kernels guarantees pixel-exactness as well.

Pre-interleaved monochrome input

This algorithm differs from the others mentioned above in its way of making use of another aspect than kernel properties as input. The algorithm expects a monochrome and preprocessed input sample RGBA texture. The idea is to make use of hardware-based per-pixel dot products. Instead of using a monochrome texture we use the RGBA texture format. Each color channel stores the same monochrome texture but using different texel offsets. In other words the texture is interleaved with itself four times. We demonstrate the interleaved texture using an example: The original texture is stored in the R channel. The G channel represents the same texture, but shifted to the "left" by one sample distance that means that the G channel stores the value toward the "right" from the current one. B and A channels represent the next two neighbors toward the "right" from the original sample value. Note that the interleaved texture is exactly four times the size of the corresponding monochrome texture. After the interleaving process each pixel stores four samples from the original sample function. Doing the same with the filter kernel, i.e., pre-interleaving it we are able to use the per-pixel dot product operation to fold four passes into a single pass. Adding the other filtering algorithms into this process as well as using all available texture units the performance increases tremendously. We denote the algorithm as *dot-2x*, the versions making use of more texture units by *dot-4x* and *dot-6x*, and those exploiting filter separability by *spd-3x*, *spd-4x*, and *spd-6x*. We describe the *dot-2x* version of the algorithm using pseudocode in table 5.3. The *spd* versions are a mixture of *sep* and *dot* pseudocode.

5.2 Applications of high-resolution filtering

This section reviews the application areas of the high-quality and high-resolution filtering. Three different filter kernels are used in our implementation in order to exploit their filtering characteristics: cubic B-spline, Catmul-Rom spline and Kaiser-windowed sinc of width four. We present results using surface texturing as an example in 2D and solid texturing as an 3D example.

global setup:	
input texture:	*monochrome, interleaved in RGBA;*
filter tiles:	*1D, 2D, or 3D, also interleaved in RGBA;*
per-pass setup:	
input texture:	*setup texture clamping;*
filter tiles:	*select tile corresponding to pass;*
	[*setup tile mirroring;*]
	setup tile repetition;
vertex setup:	
input texture:	texcoord[TEX_UNIT0] = shift_j(texcoord[TEX_UNIT0]);
filter tiles:	texcoord[TEX_UNIT1] = texcoord[TEX_UNIT0]*tile_size;
pixel setup:	
input texture:	reg0 = SampleTex(TEX_UNIT0);
filter tiles:	reg1 = SampleTex(TEX_UNIT1);
	out = DotProduct4(reg0, reg1);

Table 5.3: Filtering with the *dot-2x* algorithm, exploiting the per-pixel dot product, computing four passes within a single pass, as well as exploiting the kernel symmetry (optional).

Surface textures

Surface texturing is the most often used way for using texture hardware in consumer application areas as well as in professional modeling tools. The hardware natively only supports one type of high-resolution filtering, namely bi-linear filtering. This method produces clearly visible artifacts, especially when the sampling frequency of the input function is not high enough. However this is the next better solution after nearest neighbor interpolation. Our method preserves both high-quality reconstruction as well as interactive framerates. As illustration of high quality we show the results of various filters comparing them to linear interpolation. In figure 5.1, the Catmull-Rom spline and Kaiser-windowed sinc provide similar results. The cubic-B spline generates a little bit a blurry effect due to its approximative behavior. But in consumer oriented applications, where the main goal is to provide smooth functions this is often not considered as an artifact. Using this filter in professional applications, e.g., medical visualization can cause that the low-pass filtering removes small structures and therefore is unpracticable. However, there are possibilities for using approximative filters in interpolation oriented tasks also. We will discuss this in another section of this chapter.

Higher-order filtering is especially suited for simulating effects which have a low frequency property: The typical representatives are lightmaps for real-time display of radiosity lighting solutions [1]. This solution is often used in games, storing the whole lighting of a particular level in a texture. The lightmaps are generally sampled with low sampling frequency. Reconstructing the lightmap using a bi-linear filtering causes strongly visible artifacts (similar to those seen on figure 5.1 (a)). Using the bi-cubic filtering, especially the cubic B-spline, we get improved lighting effects at interactive framerates.

The tables 5.4 and 5.5 show the framerates of the filtering process using two different

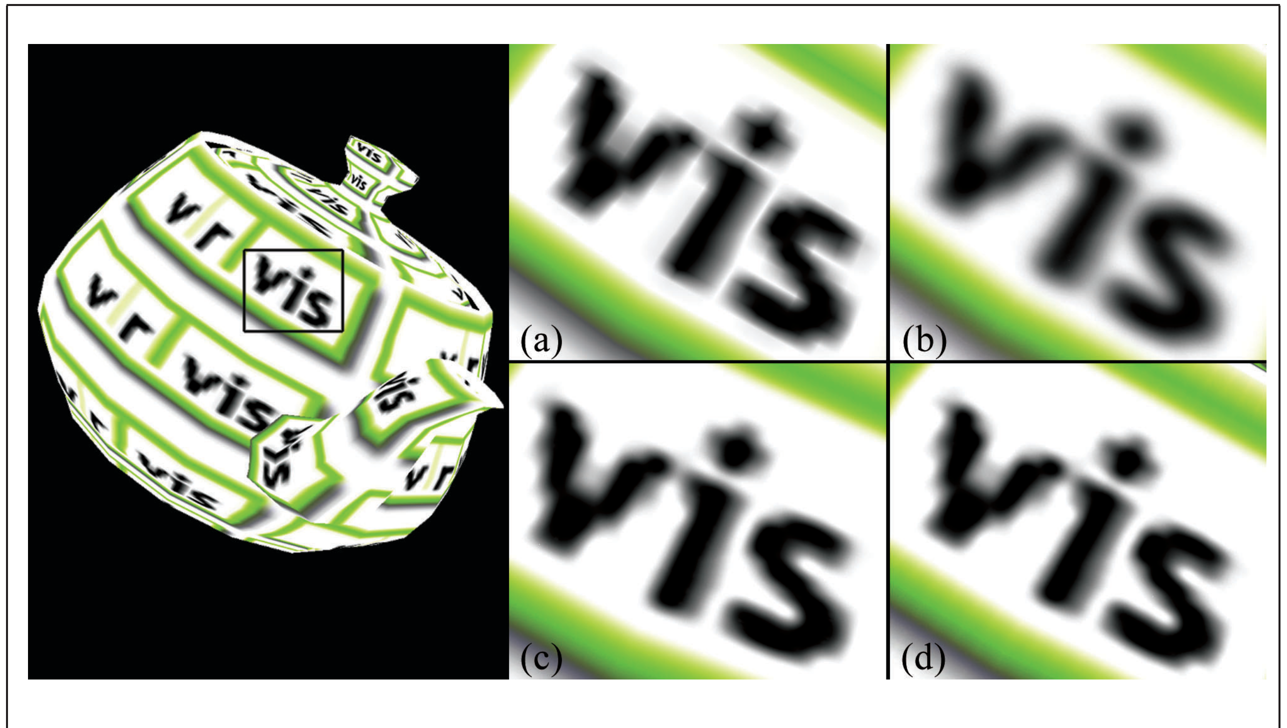


Figure 5.1: Surface-textured teapot using a standard tent (a), a cubic B-spline (b), a Catmull-Rom spline (c) and a Kaiser-windowed sinc filter (d).

state-of-the-art graphics chips, namely NVidia GeForce3 and ATI Radeon 8500. The tables compare various discussed filtering algorithms and the approximate number of pixels being rendered. Note, that GeForce3 has generally better performance, while the Radeon 8500 makes it possible to use up to six texture units in a single pass. Beside that the Radeon card has better internal precision and provides numerically better results. For this benchmark we used a fully textured object with about 500 visible triangles. The source texture of 64×64 samples was repeated on the object several times (see figure 5.1).

Solid textures

Making use of three-dimensional texturing allows to simulate effects which are hard to implement when using traditional surface textures only. The first group of effects are natural materials like wood or marble because of their internal structure [26, 27]. Imagine a vein of color that runs through a marble statue which cannot be described with standard 2D texture because it runs through the center of the statue and emerges on the other side. A similar property has wood with its annual rings. Another group of objects are volumetric objects like volumetric fog or 3D lightmaps often sampled at a low resolution. Storing them in acceptable resolution means high demands on the texture memory, which is even much more significant than in the case of 2D texturing.

Another, nowadays very popular area of application of 3D textures is hardware-based volume rendering [30, 40]. To generate acceptable results, the reconstruction stage of this process should provide high-qualitative results as well. Especially in medical visualization, where the quality of

# pixels	*std-2x* (16)	*std-4x* (8)	*dot-2x* (4)	*dot-4x* (2)	*sep-3x* (16)	*spd-3x* (4)
60k	55	105	190	275	55	190
180k	55	57	125	108	55	188
260k	50	36	80	60	46	150
900k	25	15	28	19	22	70
1200k	20	18	28	13	15	55

Table 5.4: Framerates (in fps) of NVidia GeForce3 surface texturing, using different filtering algorithms. In parentheses is the number of rendering passes.

# pixels	*std-2x*	*std-4x*	*dot-2x*	*dot-4x*	*sep-3x*	*sep-6x* (8)	*spd-3x*	*spd-6x* (2)
60k	24	46	90	167	24	46	90	167
180k	24	34	71	78	24	46	71	83
260k	17	19	45	45	24	46	55	62
900k	9	9	20	19	23	35	26	27
1200k	8	9	16	15	16	24	55	32

Table 5.5: Framerates (in fps) of ATI Radeon 8500 surface texturing. In parentheses is the number of rendering passes.

representing the dataset must follow strict conditions. The mostly used type of filtering is tri-linear interpolation. Using higher-order filters should improve the reconstruction quality. This is another application area, where the high-quality filtering is especially well-suited for.

For tri-cubic filtering we use the same filter kernels as in the two-dimensional case, however the filter kernels are three-dimensional. In figure 5.2 we see the comparison between tri-linear and tri-cubic filtering. The dataset consists of 128^3 samples. The tri-linear case yields so called staircase artifacts, which are completely removed in the tri-cubic case. For the tri-cubic filter a cubic B-spline was chosen, especially suited for such types of reconstruction.

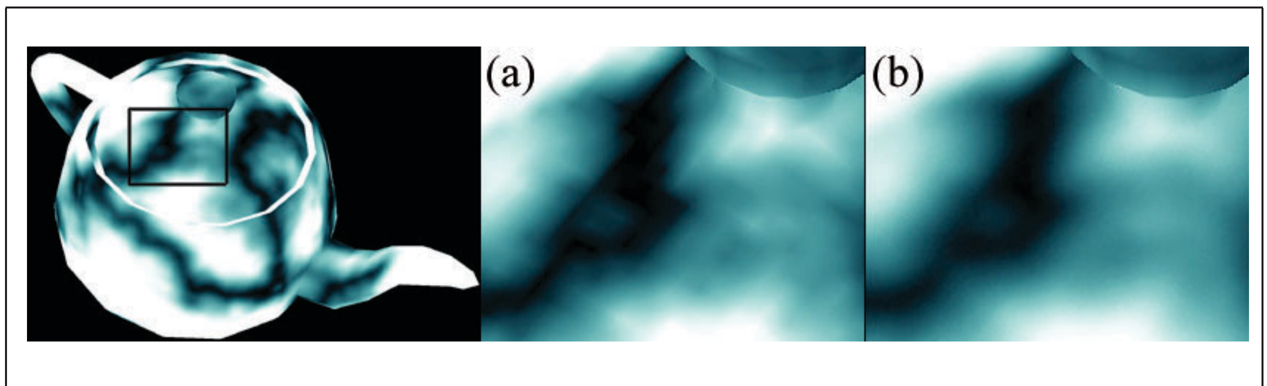


Figure 5.2: Solid-textured teapot using a standard tent (a) and a cubic B-spline filter (b).

The performance issues of the GeForce3 and Radeon 8500 are described in the following tables 5.6, 5.7. The test dataset has the same resolution as the wooden material described above, i.e.,

128^3 samples. From the benchmarks, it is visible that Radeon 8500 performs well when drawing just few pixels. With a growing number of rendered pixels the GeForce3 provides better framerates.

# pixels	*std-2x* (64)	*std-4x* (32)	*dot-2x* (16)	*dot-4x* (8)	*sep-4x* (64)	*spd-4x* (16)
60k	19	21	64	66	21	76
180k	6.5	6.8	20	20	14	50
260k	4.2	4.5	11	11	8.7	34

Table 5.6: Framerates (in fps) of NVidia GeForce3 solid texturing.

# pixels	*std-2x* (64)	*std-4x* (32)	*dot-2x* (16)	*dot-4x* (8)	*sep-4x* (64)	*spd-4x* (16)
60k	23	26	71	71	59	90
180k	4.2	4.5	15	16	30	30
260k	2.8	2.3	8	9	18	40

Table 5.7: Framerates (in fps) of ATI Radeon 8500 solid texturing.

Animated textures

High-quality reconstruction is even more important when using animated textures as compared to the static case. Linear interpolation produces much more visible artifacts, because the underlying interpolation grid, appears as a kind of static layer beneath the moving texture. Higher-order filtering like cubic interpolation completely removes these artifacts. So the animation process is not disturbed by strong artifacts, preserving realism of the animation. This grid effect is most visible in animations with rotating objects. We classify the animations into three main categories according to their generation stage.

Pre-rendered animation

The first category computes the animation frames in a pre-processing step, e.g., using any type of application suited for such animation generation. The frames are stored in separate textures and the animation is done by simply rendering a different texture each pass, providing the illusion of an animation. This type is very sensitive with respect to memory consumption, therefore the natural solution is to store the animation frames in lower resolution. These low resolution textures usually are reconstructed using linear interpolation, resulting in the already mentioned artifacts.

Figure 5.3 shows the difference between bi-linear and bi-cubic filtering of one frame from a texture animation used in a spacecraft multi-player game called Parsec [25]. The input texture is given in various resolutions: 256×256, 128×128, 64×64, and 32×32. The first two texture resolutions can be considered as acceptable, but the lower resolutions strongly pronounce the difference of the filtering methods used for reconstruction. Another example is a pre-rendered animation of fire stored at a 64×64 resolution. This animation simulates a torch fire by a popular trick in game industry. Instead of using “real” volumetric fire mapped on a 3D object, we use two perpendicular quads intersecting each other. This trick is based on the variable viewing angle preservation that means that the fire appears to be three-dimensional from each angle. The snapshot from this animation in figure 5.4 top shows the difference between bi-linear and bi-cubic interpolation once more.

Procedural CPU animation

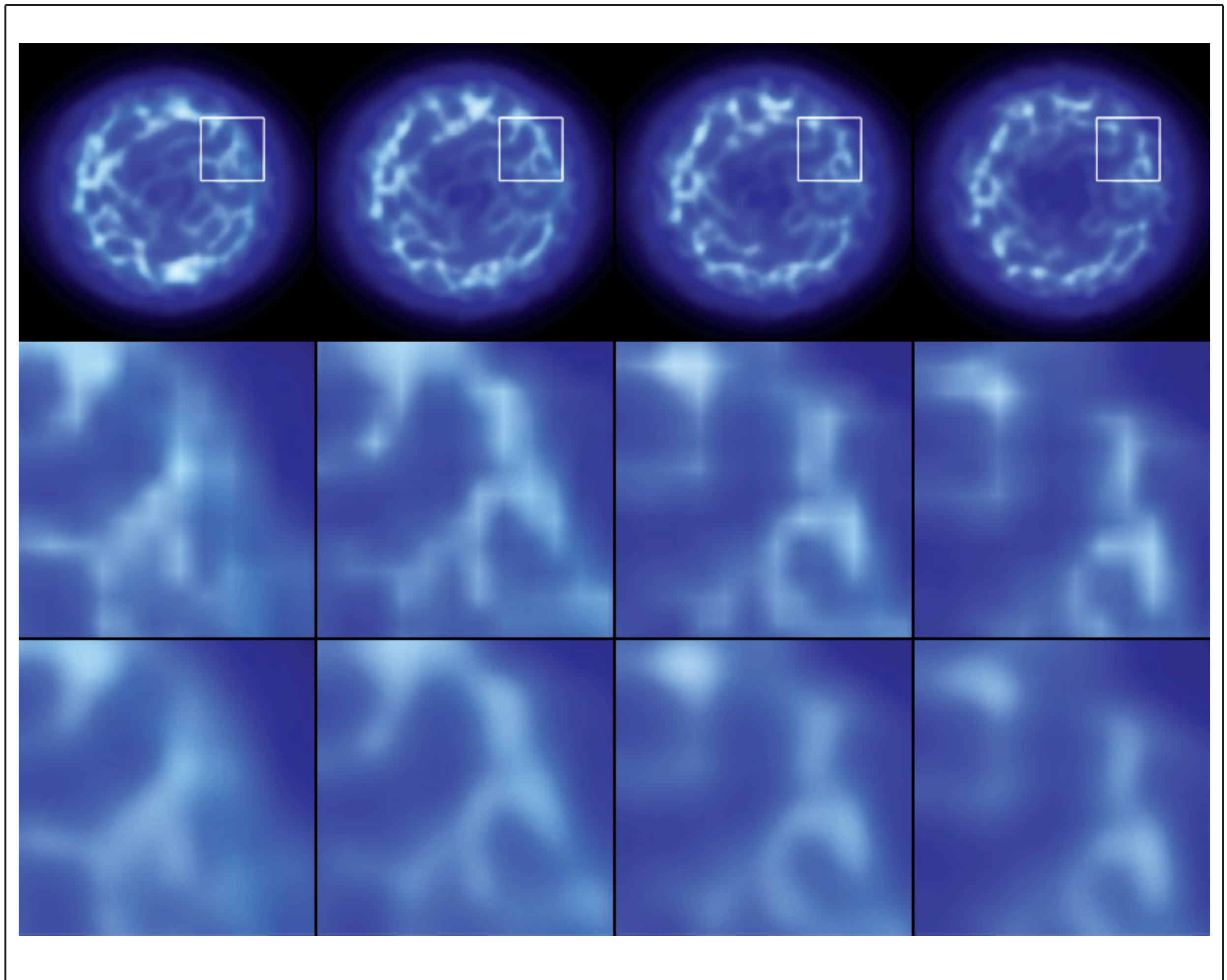


Figure 5.3: Pre-rendered animation from the space combat game Parsec [25] (first row) filtered with a standard tent (second row) and a cubic B-spline filter (third row).

Another category of animations are procedural animations. At first, we are discussing animations with frames generated on the processor. Typical procedural animations are smoke, gas, or fog as well as moving or forming marble [7]. For generating the textures on-the-fly and preserving the interactive framerate, it is necessary to reduce the computation time as much as possible. Note, that in this case also the texture transfer from the main memory to the graphics chip is a rather time-consuming operation. This type of animation has no high demands on the texture memory capacities. However, in order to achieve a sufficient framerate we are forced to minimize the number of computations to a minimum. So also this leads to low resolution frames which consequently need to be reconstructed at as good as possible quality. Cubic interpolation suffices due to low computational demands and fast high-quality reconstruction, achieving interactive framerate of a realistic animation.

Procedural GPU animation

A similar category from the "family" of procedural animations includes those which are computed on the graphics chip itself. Although the graphics processor does not offer a whole lot of flexibility,

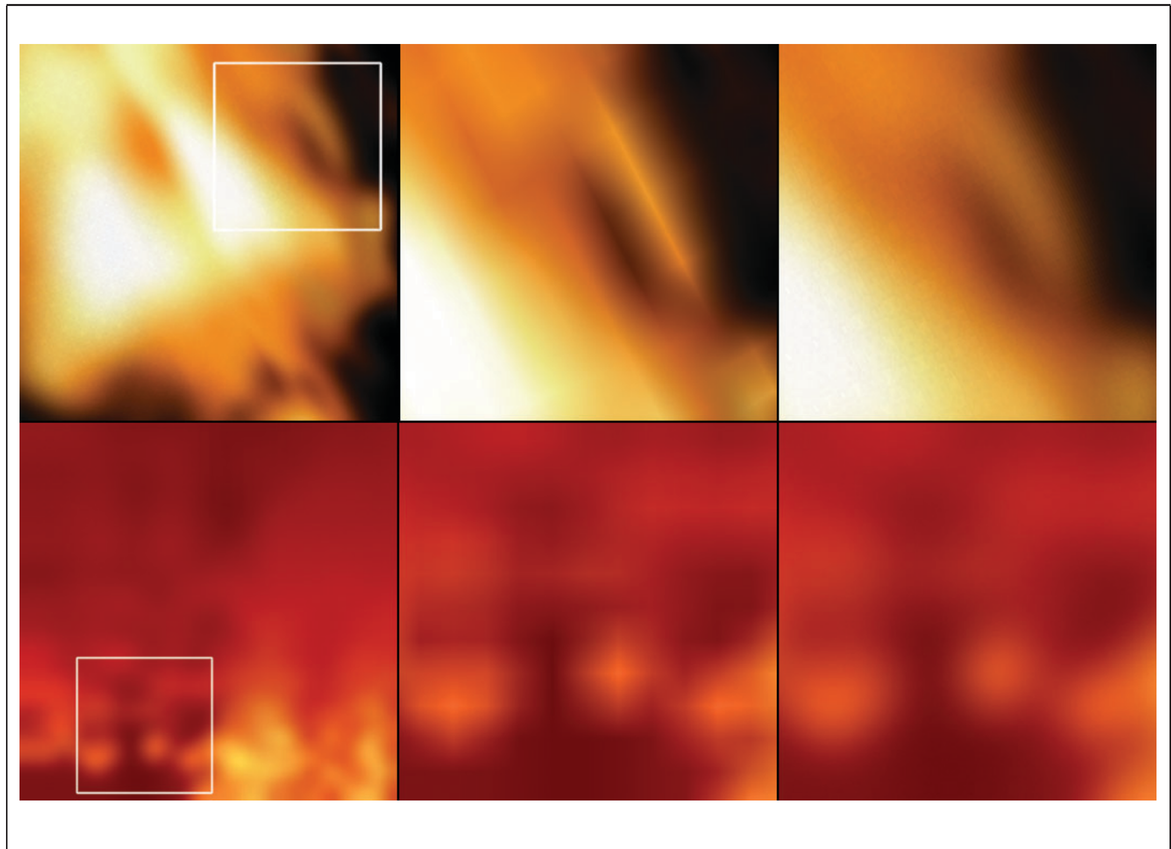


Figure 5.4: Pre-rendered animation of fire (top) compared to procedural GPU animation (bottom). The figure shows the animation frame, maginfied regions filtered with a standard tent and a cubic B-spline filter (from left to right).

and therefore only a limited range of animations are possible when compared to those which are generated on the main processor, there is one significant advantage: we do not need to transfer the sampled function to the graphics memory, because it is already there. This saves some part of the computation time. But in general the necessity of interactive framerates requires to reduce the computation time.

One nice example how to make use of hardware for procedural animation was introduced by James [15] shown in figure 5.4 bottom. The algorithm generates a fire animation based on basic texture and image processing operations. It generates source "embers" using the well-known Game of Life algorithm. This is implemented in hardware making use of the dependent textures feature. The embers are then blurred in hardware using basic linear operators. The blurring (or smoothing) approach will be described in detail in the next chapter. After blurring, the algorithm shifts the blurred embers upwards achieving the impression of moving fire particles. This shifting is done by a simple increment of the v texture coordinate. The process is repeated in an endless loop producing always new animation frames. After the frame generation the high-quality filtering approach can be applied. The resulting animation is comparing again the bi-linear and bi-cubic reconstruction process. The algorithm uses a 32×32 texture resolution.

Derivative reconstruction and applications

Up to now we have only discussed high-resolution function reconstruction. Another issue is derivative reconstruction. Interestingly there is no derivative reconstruction natively implemented in hardware as in the case of function reconstruction. Therefore the reason to implement a hardware-based filtering is clear. Compared to software solutions, this approach provides sufficient framerates even in the 3D case and produces much better results than, for example, a Sobel edge detector or central differences. Beside that, Sobel and central differences are not high-resolution filters at all but belong to the image processing field.

Derivative reconstruction can make use of all the filtering algorithms described at the beginning of this chapter only with some small changes. One difference regards kernel symmetry: a reconstruction kernel for first-order derivatives usually is generated as multiplication of one one-dimensional derivative kernel and one (in 3D two) one-dimensional function kernel(s). The derivative kernel is the derived function kernel. For the 2D case we need two such kernels to estimate the partial derivatives of both dimensions, for the 3D case we need, of course, three kernels. From the above it is clear that derivative kernels fulfill the separability property as well. The problem is the range of the filter kernel,i.e., the extreme values are above 1 and below -1. We have partly solved this problem with biasing and scaling the values to the hardware range, but we are still not satisfied with the results. Therefore we have to implement some techniques to increase the quality of derivative filtering.

5.3 Problems and solutions

The implementation of the filtering process in hardware is dealing with several problems which usually do not appear in software solutions. This chapter reviews general particular problems and shows the way how they can be solved. The only problem also appearing in software implementations is dealing with approximative filters. The biggest problem, however, still is the framebuffer range and precision.

Approximative filter kernels

In our implementation we use one filter kernel of approximative behavior, i.e., the cubic B-spline filter. We have already discussed the problem of smoothing artifacts in some application areas. This problem, however, can be solved in a preprocessing step.

The approximative spline kernels do not operate on the function points as the interpolative ones do, but on a set of so-called control points. The function points are located within the convex hull of these control points. The task is to find appropriate control points from the function samples. After solving this problem, the approximative curves (based on the pre-processed control points) represent the function as the interpolative curves with function samples. We will show how to compute the

control points using a 1D example. The relation between control points and function samples for a cubic spline is given by the following matrix equation [42]:

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \cdot \\ \cdot \\ f_{n-2} \\ f_{n-1} \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 4 & 1 & & & & & \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & \cdot & & & & \\ & & & \cdot & & & \\ & & & & 1 & 4 & 1 \\ & & & & & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \cdot \\ \cdot \\ c_{n-2} \\ c_{n-1} \end{bmatrix}$$

The function points are denoted as f points and the control points as c. Labeling the three matrices F, K, and C we rewrite this relation:

$$F = K \cdot C$$

From this it is obvious, that the coefficients in C could be computed from the sampled data points F and the inverse of the tridiagonal matrix K:

$$C = K^{-1} \cdot F$$

Although the computation of the control points can be a time consuming process, it is necessary to do this only once, in a preprocessing step. After this is done the cubic B-spline should produce similar results as interpolative kernels, e.g., the Catmull-Rom spline kernel. Filtering with cubic B-splines has a lot of advantages, the “dot” and “spd” acceleration works well with this filter, because it is always positive and does not complicate the filtering process with negative values, which always appear in interpolative curves.

Framebuffer-related challenges

Using interpolative filters for functional reconstruction we have to deal with shortcomings of current graphic hardware, which is its limited value range, i.e., the $[0, 1]$ interval. The question is, how to solve the filtering process, when the filter contains negative values. Although signed textures already became available, we are always forced to store the subresult in the framebuffer which has the range in the $[0, 1]$ interval. Fortunately we can subtract the values from the framebuffer. This makes possible to use filters with negative values, such as Catmull-Rom splines or windowed sinc. To solve the problem of filtering with negative tiles, we have to order the filtering passes in a way, that the framebuffer values never result in values below zero in the computation. Clamping means clipping, e.g., if the sum of two operands exceeds one the result is equal to one. To reduce the clamping error on both boundaries of the effective interval, we mix the order for producing optimal results. The filtering passes must be ordered to minimize the clamping effect. In some cases it is necessary to split a particular filter tile into two parts, because the tile has too high values. The sum of these parts represents the original filter tile. We generate one new tile called split-tile, storing one part in the original tile and the other part in the split-tile. We reorder the filtering passes so that we insert one filtering pass using a negative tile between these two tiles.

Derivative reconstruction requires much more “coding hacks” than the zeroth-order case. First, the result of function reconstruction is naturally within the interval $[0, 1]$ as the sampled function is also. This is not the case for gradient information, i.e., results of first order derivative reconstruction. The

interval is extended to the range $[-1, 1]$ or even more, this depends on the given filter kernel. Therefore the filtering subresults are scaled and biased to the $[0, 1]$ interval. Naturally, zero is represented by the value 0.5. These scaling and biasing must be processed carefully, in order to minimize numerical artifacts. Another problem is a sign change within one filter tile. Tiles often have both positive and negative values. This can be solved in a similar way as with split-tiles (mentioned above): we divide the tile into two tiles, one original and one split-tile. The tile values are divided into positive and negative values. The negative values are transfered to the split-tile. Instead of the negative values we insert zeroes into the original tile. After this we filter the same corresponding sample in two passes with the different filter tiles.

An unresolved problem remains the precision of the framebuffer. In the case of derivative filtering this is especially significant because we need to scale the values to the small interval $[0, 1]$ represented by 8 bits. So we represent an interval of double length with only 8 bits. This is not at all sufficient for high-quality reconstruction. This problem should be solved by hardware manufacturers in their future solutions.

Chapter 6

Image Processing

In this chapter we will describe another filtering approach, suitable for image processing tasks. We will not deal with high-resolution kernels, but with simple linear operator matrices. We have already discussed the difference between high-resolution filtering and the image processing principle. However, the general filtering approach also describes image processing, in practice. The difference is that instead of textures for filter tile representation simple color values are used. The number of resampling points is the same as the number of input samples which means that using linear operators we are only processing existing values to change them without generating any new samples.

Using color instead of entire textures has a significant impact on the performance. Instead of using two textures – one for the input function and one for the filter kernel – we only need one texture for a single filtering pass. Exploiting the hardware capabilities, this allows to merge more filtering passes into a single rendering pass.

A significant difference to high-resolution filtering is also the variable filter width. All algorithms from the previous chapter are implemented for kernels of width four. In the case of image processing, we are able to filter input with arbitrary filter widths. This is done by using lookup tables which completely describe the filtering process. Using arbitrary filter kernel widths is a performance/quality trade-off. Either we accept results strongly influenced by the limited framebuffer precision or we involve some quality improvements that are, especially at this time, rather time-consuming operations.

Another interesting issue is the possibility to use the OpenGL imaging subset for such image processing tasks. This is not really efficient at the moment, the filter width is limited by the hardware and the framerates are far from sufficient. The implementation of the imaging subset does not seem to be hard-wired in hardware yet, it is probably only a software implementation in the driver. But this kind of filtering approach has its advantages as well. It is pretty simple to implement a filtering operation and there are no problems with the framebuffer precision at all. The question is, when it will be implemented in graphics hardware to provide acceptable performance at arbitrary kernel widths.

In the following sections we will deal only with texture-based image processing, describing the methods using pseudocode, and showing the differences to the previous mentioned algorithms. The next section deals with a couple of application areas and the last part describes some problematic issues of the filtering process.

6.1 Filtering algorithms in image processing

The algorithms presented in this section are analogous to their high-resolution counterparts. Theoretically the only difference is the sampling rate of the filter kernel. But this has a significant impact on the whole filtering process. In the case of image processing we do not make use of symmetry or separability, because we do not use texture memory for the kernel representation and the number of values within a typical linear operator is often only a couple of floats. The whole mirroring and on-the-fly kernel computation are not present here. So for image processing tasks we use the *ips* (image processing standard) acceleration for any type of input texture, and *ipd* (image processing dot) approach for monochrome textures interleaved in RGBA. These are analogous versions when compared to the *std* and the *dot* algorithms for high resolution filtering.

Standard algorithm

The image processing *ips* algorithm is derived from the general *std* approach. Our implementation is able to use any type of linear operator. The operator is defined by its dimension, weights stored in an array and a shift table that assigns texture coordinate offsets to the weight values. The process using one texture unit takes the first value from the weight array and sets the color to this weight value for each color channel. Then the input texture is shifted according to the offsets given for the corresponding weight value. After this the texture is mapped on a quad and multiplied with the color value. This is performed in a loop for all filter weights summed together in the framebuffer. The other image processing *ips* variations simply fold more filtering passes into one rendering pass. This has the same advantages as high-resolution filtering, i.e., speed and quality obtained from the multi-texturing principle. The *ips-1x* algorithm is illustrated in table 6.1.

global setup:	
input texture:	*as is, all formats (mono, RGB, RGBA, etc.);*
linear operator:	*generate 2D operator matrix;*
per-pass setup:	
input texture:	*setup texture clamping;*
linear operator:	*setup color value* COLOR0 *corresponding to pass;*
vertex setup:	
input texture:	texcoord[TEX_UNIT0] = shift_j(texcoord[TEX_UNIT0];
pixel setup:	
input texture:	reg0 = SampleTex(TEX_UNIT0);
linear operator:	reg1 = COLOR0;
	out = Multiply(reg0, reg1);

Table 6.1: Filtering with *ips-1x* algorithm.

Pre-interleaved monochrome input

The general difference between the image processing *ipd* algorithms and the *dot* versions from high-resolution filtering is the possibility to use filters of variable width. The *dot* implementation is strongly based on the filters of width four, however it can be extended to broader widths as well. The image processing *ipd* version on the other hand allows to process images with filters of arbitrary width. This complicates the algorithm a bit in the case when the filter width is not a multiple of four. This is due to four color channels, where each channel stores the same image but shifted by a particular offset. We handle the filtering for an arbitrary width in the following way.

Each RGBA sample stores the original value in the R channel, other channels store values towards the "right" from the current one. Filtering with kernels of width of a multiple of four is analogous to the high-resolution algorithm, i.e., folding four passes into one. The case of other filter widths utilizes the next greater multiple of four as compared to the kernel width. The filtering process is performed regularly again as the kernel width is a multiple of four. The trick is that the values of the filter kernel above the original width are set to zeroes and therefore have no influence on the computation. The *ipd* image processing implementation is described in the table 6.2.

global setup:	
input texture:	*monochrome, interleaved in RGBA;*
linear operator:	*2D operator matrix also interleaved in RGBA;*
per-pass setup:	
input texture:	*setup texture clamping;*
linear operator:	*setup color value* COLOR0 *corresponding to pass;*
vertex setup:	
input texture:	texcoord[TEX_UNIT0] = shift_j(texcoord[TEX_UNIT0];
pixel setup:	
input texture:	reg0 = SampleTex(TEX_UNIT0);
linear operator:	reg1 = COLOR0;
	out = DotProduct4(reg0, reg1);

Table 6.2: Filtering with *ipd-1x* algorithm.

6.2 Applications in image processing

The range of application of image processing operations covers the whole image processing and pattern recognition fields. These operations are classified as so-called local pre-processing methods and are used in video surveillance to detect movements or identify subjects according to certain properties. Other areas where convolution plays a significant role are computer vision or automatic segmentation. For all of them the performance issue of the process is very important. These approaches need to process input, for example, obtained from CCD cameras, in real-time, otherwise the process loses effectivity or does not make sense any more.

Other application areas of hardware-based image processing solutions are filters for image editing and graphics design or stream and movie editing applications, where there is need to filter a huge amount of frames within a limited time. Such processes, called often post-processing, can not only be used in conjunction with already existing streams, but they can be also build right into the rendering process to render a typical scene in a non-photorealistic way, for example. These areas are discussed in more detail in the following sections.

Smoothing

We have already introduced the smoothing operators in the filter basics chapter 2.7. These filters are used for noise suppression, i.e., “cutting” off the high frequencies. We will introduce just two, often used smoothing operators. The first type is simple averaging. A given neighborhood of a sample is summed and divided by the number of involved samples. In practice, we use a linear operator with constant weights (in sum equal to one). When we perform filtering in hardware we have to set the weight, i.e., color value, only once and process all filtering passes using the same value. This is the simplest smoothing operator analog to a box filter with larger width than one. Other filters are more sophisticated. The Gauss smoothing operator preserves the significance of the sample in the center of the convolution sum. The convolution matrix contains non-constant values which also sum up to one. All filtering operators discussed here consider an 8-neighborhood of the processed pixel that means that all eight surrounding samples are considered as neighbors.

The smoothing operation often needs to operate on a wide neighborhood to smooth the values or remove noise artifacts sufficiently. Obviously a wider neighborhood needs more filtering passes and due to the framebuffer precision is prone to summation artifacts. The second problem is discussed separately in the last section of this chapter, so first we take a look at the performance issues of the filtering process. Table 6.3 shows framerates for all algorithms with various kernel sizes. The number of real rendering passes depends on the kernel size and the number of textures used in a single pass and whether texture interleaving was used or not. The input image was sampled at a 512×512 resolution for the listed measurements.

kernel width	img. subset	*std-1x*	*std-2x*	*std-4x*	*dot-1x*	*dot-2x*	*dot-4x*
3	3.26	120.00	180.00	185.00	275.00	307.00	314.00
5	2.86	50.00	74.00	75.50	118.00	169.00	155.00
7	2.48	28.00	40.50	41.70	90.00	131.15	121.00
9	1.95	17.00	25.50	26.40	50.00	73.66	76.45
11	-	12.00	17.55	17.99	41.48	61.30	55.96
13	-	8.20	12.61	13.55	26.86	41.75	43.15
15	-	6.24	10.03	10.05	22.39	36.68	35.40
17	-	5.00	7.66	7.96	16.76	28.17	25.27
19	-	3.95	6.30	6.61	15.02	23.42	24.12

Table 6.3: Framerates (in fps) of NVidia GeForce3 smoothing operation, using different filter kernel widths.

Important to note is that the framerates do not describe the quality of the filtering process. Filter kernels larger than seven do not provide sufficient quality, so it is necessary to include a quality

improvement called hierarchical summation into the filtering process. The results then are comparable to software filtering, but the overall performance is decreased, i.e., the framerates are at about half when compared to the low-quality approach. The most "expensive" operation is copying the subresult into the texture. We will describe the hierarchical principle later in the last section of this chapter.

Figure 6.1 shows the same image filtered with various kernel sizes and different approaches using averaging. The first row shows results using OpenGL imaging subset, the second row shows results of the *ips-1x* algorithm with rather poor results for bigger kernels. The third row shows *ips-2x* approach with hierarchical summation. The significant quality improvement was caused by making use of higher internal hardware precision and by hierarchical summation.

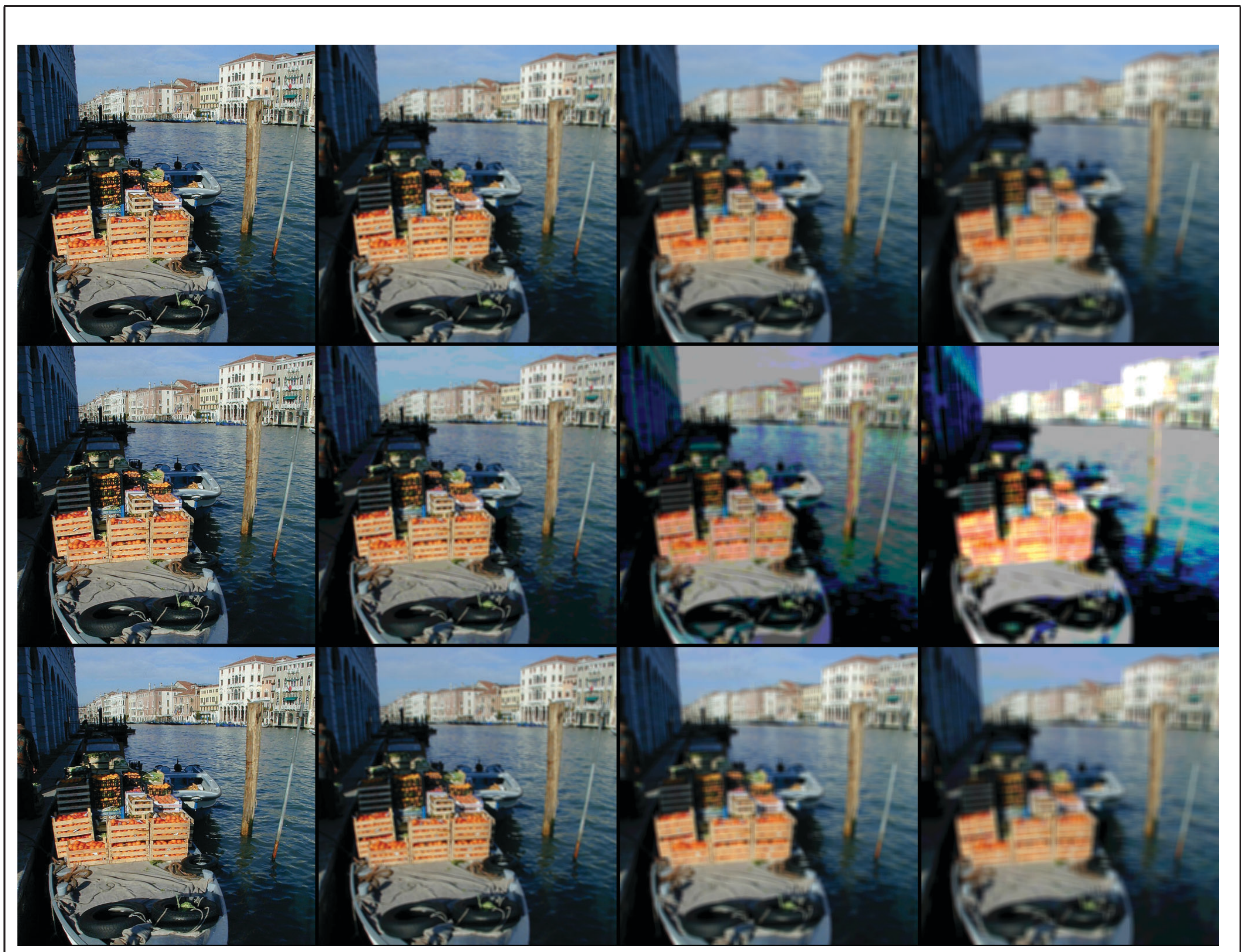


Figure 6.1: Averaging operation with operators of width 1×1, 5×5, 9×9 and 13×13 using OpenGL imaging subset (first row), texture-based filtering (second row) and texture-based filtering exploiting hierarchical summation (third row).

Edge detection

Another important task also based on the convolution operation is edge detection. It is used in a wide area of pattern recognition applications. A good example is the automatic identification of persons.

This process involves a lot of sophisticated techniques based on relations between features of the human face, but every of these applications first needs to preprocess the image to be able to find such features. This can be done using edge detection for segmenting parts of the image to find boundaries between. To process this efficiently within a limited time it is necessary to reduce the time of each step to a minimum. The usage of graphics hardware for such a process can improve the performance significantly.

In our implementation we use two basic types of edge detection operators. Each kernel has width three in each direction, so the mask consists of 9 weights, and therefore the filtering process is accomplished very fast, with acceptable results, and without any time-consuming quality improvements. The first example is the Laplacian filter, based on an approximation of second derivative kernels. The Laplacian filter uses only one kernel to detect differences between image pixels in all directions. Using one kernel for all directions leads to very short processing times. We use two types of Laplacian filters shown in figure 6.2. The first one is implemented "as is", the only difference is that the middle pixel contribution, which is negative is subtracted from the previous contributions at the end. This means all positive contributions are summed in the framebuffer and the last filtering pass subtracts the negative contribution of the middle pixel from the framebuffer. This process is very fast on the one side, however it has quite low intensity response of the boundaries of image segments. One possibility to improve edge visibility is to scale the resulted image values by a suitable constant. This approach has two main drawbacks, i.e., the scaled image significantly suffers from quantization artifacts and the scale operation requires additional processing. To perform the scale operation in hardware (which is generally the fastest solution) the image has to be copied from the framebuffer to a texture and scaled in the pixel shader. Therefore we have implemented another approach using another version of the Laplacian kernel. Although this consists of more filtering steps (see section 7.2), the performance as well as quality is much better than with the low-response Laplacian version with additional scaling.

Figure 6.2 shows that the weights also sum up to zero, but the partial sum of some of these weights exceeds the hardware range of $[0, 1]$. Therefore we split the negative contribution of the middle weight into eight separate contributions. Instead of processing 9 filtering passes, we process 16. After each positive filtering pass we insert one negative. This can cause some numerical errors, but comparing the results visually to a software implementation, there is no significant difference. The Laplacian filter is popular because of its simplicity and fast processing. It also has its drawbacks, i.e., the filter approximates the gradient magnitude only, so there is no possibility to estimate the gradient direction.

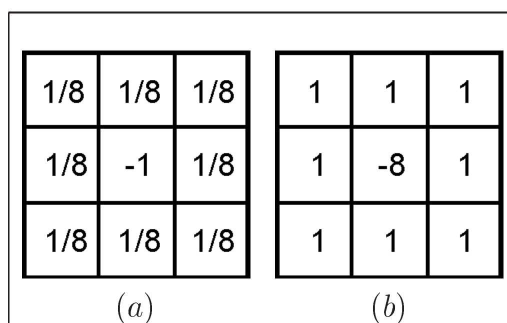


Figure 6.2: Laplacian with low edge response (a) and high edge response (b).

For the computation of gradient directions another category of edge detectors is used. This cate-

gory is based on an approximation of the first derivative, also called compass operators. Such a filter consists of two or more kernels according to the desired response of respective edges in the image. A typical and often used representative of compass operators is the Sobel operator. We use four types of Sobel operators, classifying them according to the intensity and direction of the response of the edges. These Sobel variations are shown in figure 6.3. The first one (a) takes into account the orthogonal edge responses, but only in one direction of each orthogonal dimension. That means, that the final image does not include negative gradient responses because the negative values are clamped to zero and they are represented by black. It is easy to implement it similarly to the first mentioned Laplacian filter. When we order the filtering passes correctly, the partial weight sums do not exceed the hardware interval. This approach has the same disadvantage as the first Laplacian filter case, i.e., the edge response is not strong enough. The next Sobel variation (b) solves this problem with stronger weights, but we need to split the weights greater than one to more filtering passes similar to the second Laplacian filtering implementation. This solves the problem of low response, but it has the same problem with invisible "negative" edges.

To detect edges for bidirectional (orthogonal) response we need two kernels for each orthogonal dimension. The "negative" edge responses become "positive" after rotating the operator matrix by 180 degrees. The filtering results are then summed together in the framebuffer to have the one dimensional edge response completely covered in the final image. The last Sobel version is an extension of the previously mentioned one in such a way that it takes into account also the diagonal response of edges. It covers all eight possible edge responses within a 3×3 kernel. Although each additional filter kernel slightly decreases the filtering performance, all Sobel variations perform within a well-acceptable time.

Framerates of all mentioned edge detectors are compared in table 6.4. The input image was the same as used for the smoothing operation benchmarks, with resolution of 512×512 pixels. We do not provide any benchmarks of the *ipd* algorithm because it works only for four folded filtering passes of identical sign. This is not possible in the case of higher weights, where we switch the filter weight sign for every pass to store the subresults within the hardware interval. The *ips* versions perform (due to a small number of filter weights) very well. For extremely time sensible applications Laplacian filters are the most suitable ones, when also gradient direction information is required we recommend Sobel filters. Although the "low" versions are also included in the benchmark table, they are not really useful for edge detection without any contrast enhancement applied as post-process. But that as mentioned above costs additional time.

edge detector	# kernels	img. subset	*std-1x*	*std-2x*	*std-4x*
Laplacian low	1	3.18	125.00	165.00	190.00
Laplacian high	1	3.18	77.00	118.00	120.00
Sobel low	2	1.38	65.00	106.00	133.00
Sobel high	2	1.38	66.50	97.00	94.50
Sobel high	2×2	0.70	35.30	50.00	51.15
Sobel high	4×2	0.35	16.91	23.30	24.10

Table 6.4: Framerates of NVidia GeForce3 edge detection, using different edge detecting operators.

We present the results of the above mentioned edge detectors also produced by the OpenGL imaging subset. A visual inspection does not exhibit any difference between the methods. In figure 6.4

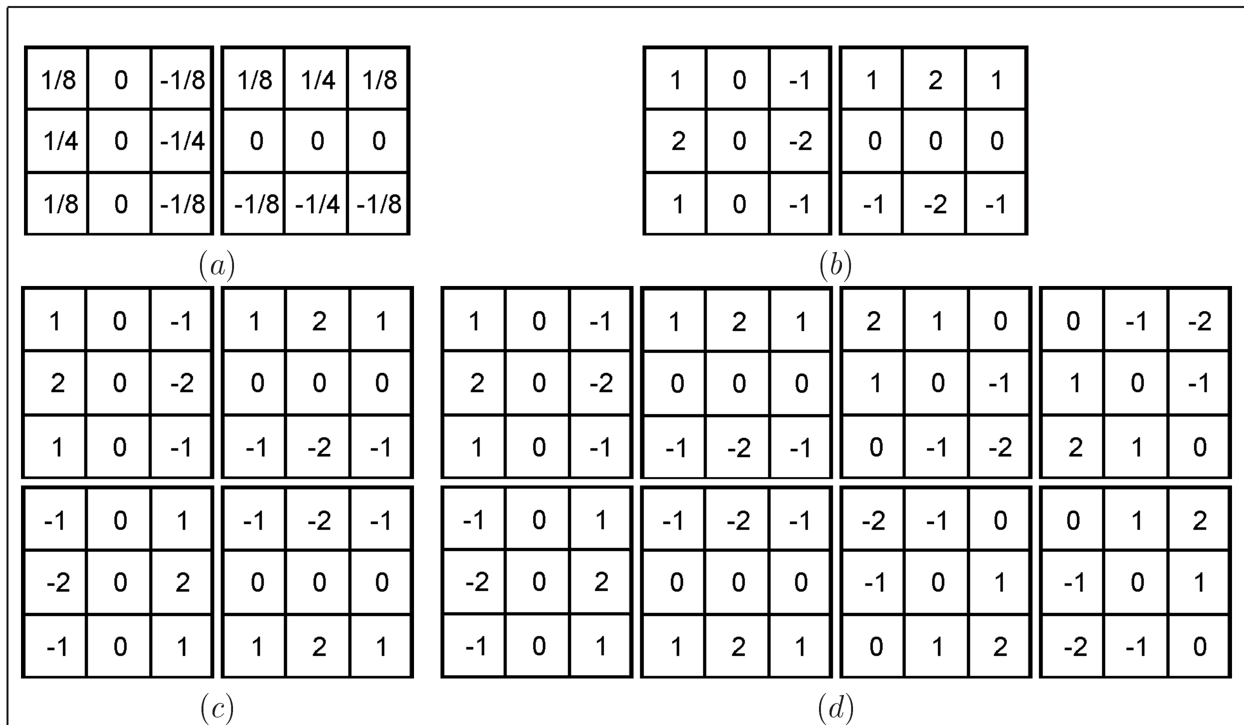


Figure 6.3: Sobel with low edge response (a) and high edge response (b), bi-directional response (c) and bi-directional, orthogonal, and diagonal response (d).

we see the results of all mentioned filters. The highest filtering quality is provided exploiting the maximum number of texture units in one rendering pass.

Artistic rendering techniques

Besides image processing applications, the area of image editing applications is of big importance as well. Most significant are areas dealing with “moving” pictures. To process a huge amount of stream frames within a limited time, or in real-time is either not possible with software or additional hardware is required. Although we also use hardware for the filtering process, the hardware is the graphics card, i.e., part of the standard equipment of each computer system. In this section we introduce some of so-called non-photorealisitic rendering techniques, that exploit hardware capabilities for real-time processing. These techniques combine image processing filtering with other standard hardware operations.

Painting with enhanced edges
This technique consists of more filtering steps combined together using the hardware’s alpha test. First the whole image is processed by a Sobel operator using the eight kernels type. The result is stored in a texture. The next step is smoothing the original image to cut off the high frequencies of noise. The result of this step is also stored in a texture for further processing. Thirdly we render either the inverted original image to the framebuffer, or fill the whole image with a particular color. This rendering step can be considered as background rendering and its contribution to the final image

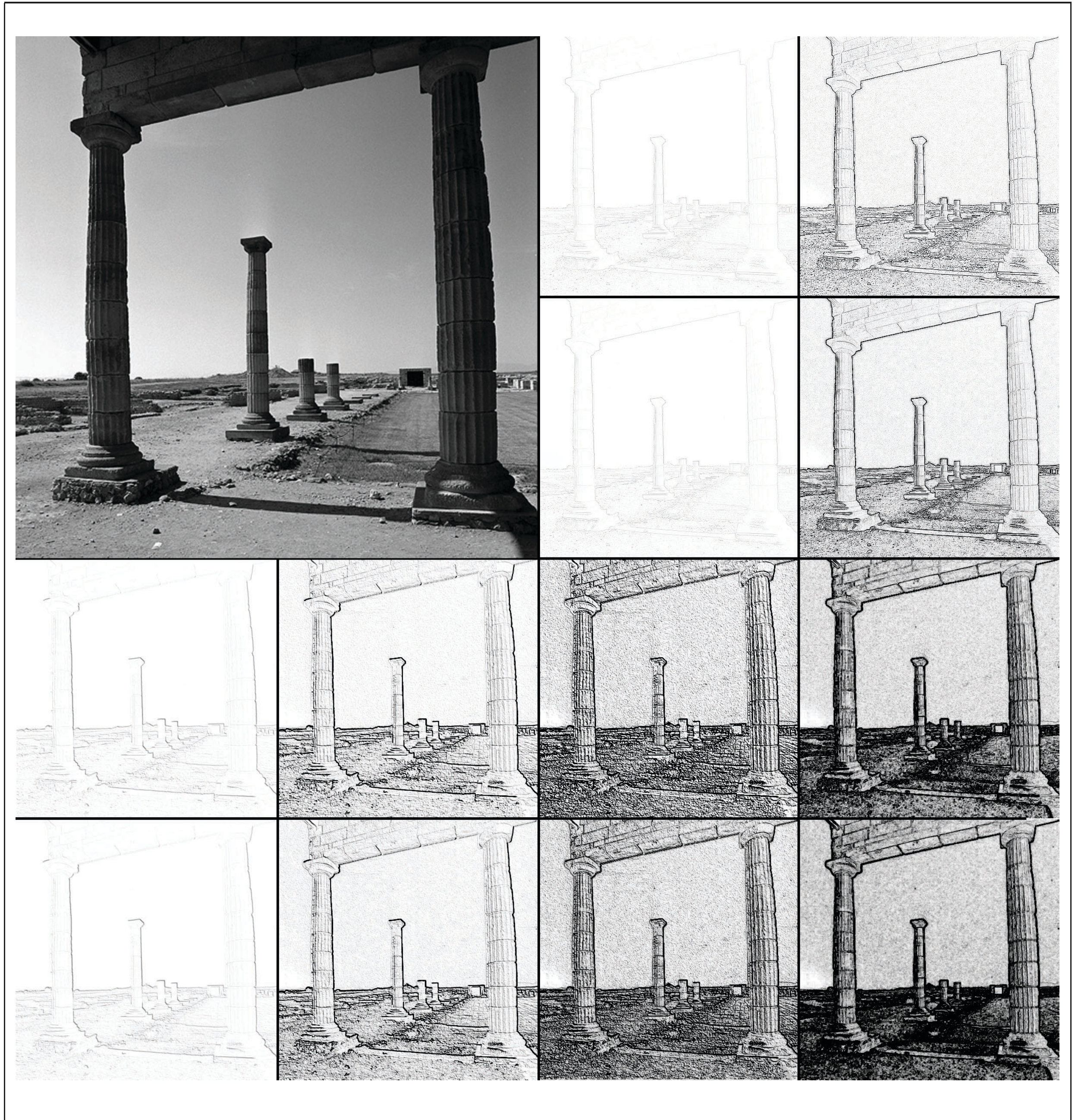


Figure 6.4: Edge detection using Laplacian low and high (upper two rows), Sobel low, high, high 2×2, and high 4×2 edge detector variations (below). Upper images are filtered using the OpenGL imaging subset and below the corresponding texture-based filtering result is given. The resulting colors are inverted.

is only visible along edges. The last part of the algorithm puts the Sobel-filtered and smoothed original textures into the pixel shader. The Sobel texture is put into the alpha channel of one register which is then used in the decision, whether to overwrite the framebuffer pixel with the smoothed original image or not. In other words the smoothed texture is used for all those image parts, where the Sobel result is below a particular threshold, that means outside the edges. The background color

only remains along the edges. The impression of painting is achieved firstly by contour lines and by comparably large areas of equal color, which is the result of cutting off the high frequencies by the use of smoothing. If we turn off smoothing we get the original image with highlighted edges. This can be probably used in video surveillance or automatic person identification as well. The method works at acceptable framerates (ca 20 fps on an 512×512 RGB image) making it possible to use it in real-time applications. The technique is illustrated in figure 6.5.

Figure 6.5: Painting with enhanced edges.

Filter combination on a pre-masked image
The method proposed in this section is used for images, which store a segmentation mask in the alpha channel. Each image "layer" or segment uses a particular alpha value. A filter type is assigned to each layer beforehand. These layers are separately filtered and the result is stored in the framebuffer. We use the alpha test to decide which filtered segment will contribute to a given pixel position. The figure 6.6 shows an original image and the result image which consists of several, differently filtered segments. This method, however, does not deal with segmentation itself, it assumes a pre-segmented image as input.

Pointilism-style paining
The pointilism algorithm is based on a randomly generated mask, used during the stencil test to decide which texel from two textures will be set at a particular position. This method is very simple, however, it produces very nice results. We combine two textures according to the stencil test. The randomly generated mask is the only computation done on CPU, the filtering and combination is done on the graphics chip. We combine a smoothed version with the original image. Due to the random mask it generates a pointilism-style effect. The biggest differences of pointilism-style images

Figure 6.6: Filtering of pre-masked image.

when compared to their originals are appearing along edges, which are represented rather by points than exact lines. The difference is clearly seen on a filtered photograph in figure 6.7.

Anisotropic filtering
We have shortly discussed anisotropy as an aliasing effect due to non-symmetric filter kernels. Even tough this effect is undesired in function reconstruction it can be used as nice, non-photorealistic effect in artistic image editing. It generates a similar effect as a brush stroke in case of an elliptical filter kernel. The brush strokes are oriented always in the same direction as the kernel. Figure 6.8 shows the original image, and the filtered result.

Post processing after rendering

The idea of post-filtering was already discussed, because it is fundamental for some application areas. With post-filtering we refer to filtering of arbitrary input, either video streams or synthetically rendered frames. The filtering is directly integrated into the application process (which is a real-time application). Post-filtering of video streams can be used in real-time applications of pattern recognition or in the movie industry. Filtering rendered synthetic images can lead to non-photorealistic rendering in real-time. This case exploits the fact, that the processed image is already in the graphics hardware and transfer time is saved. We show two examples, the first is filtering a video stream to illustrate the applicability to video surveillance and the second example is post-filtering of synthetic images in screen space. We use the high-resolution filtering application to produce the frames, which are then once more filtered in hardware. If we use an edge detector we clearly see the difference in

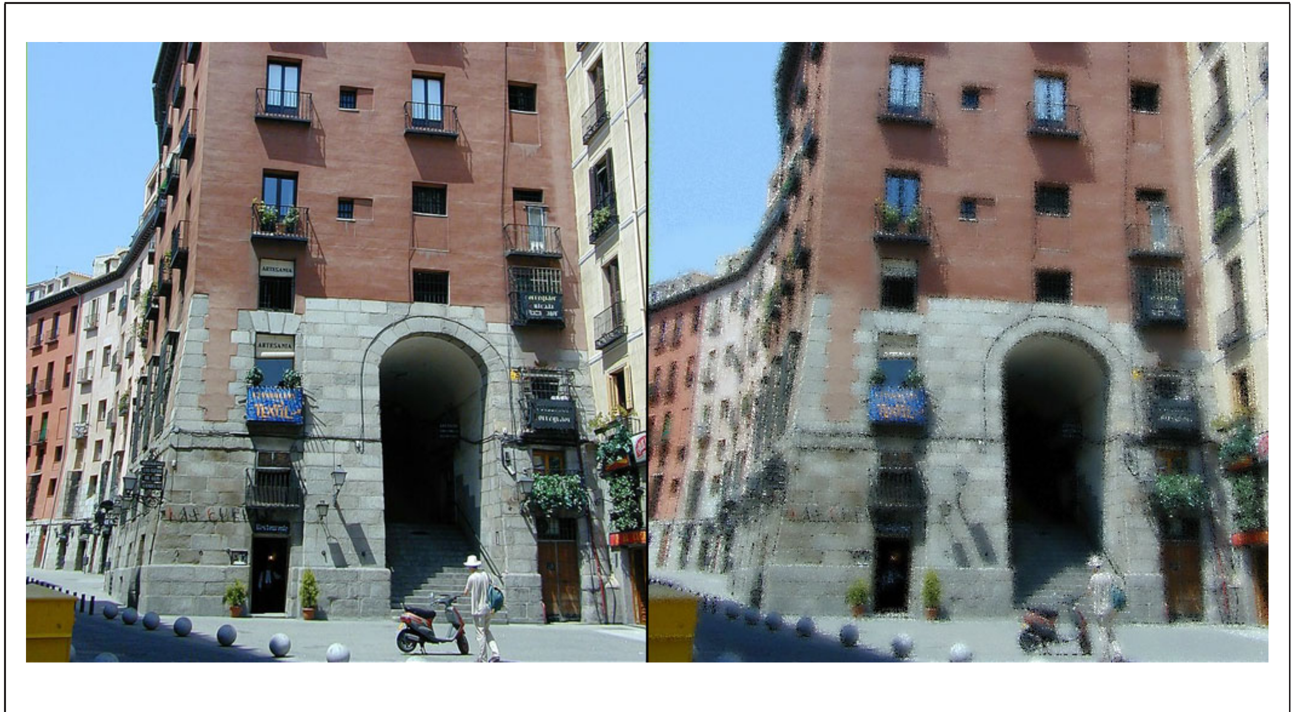


Figure 6.7: Pointilism-style painting.

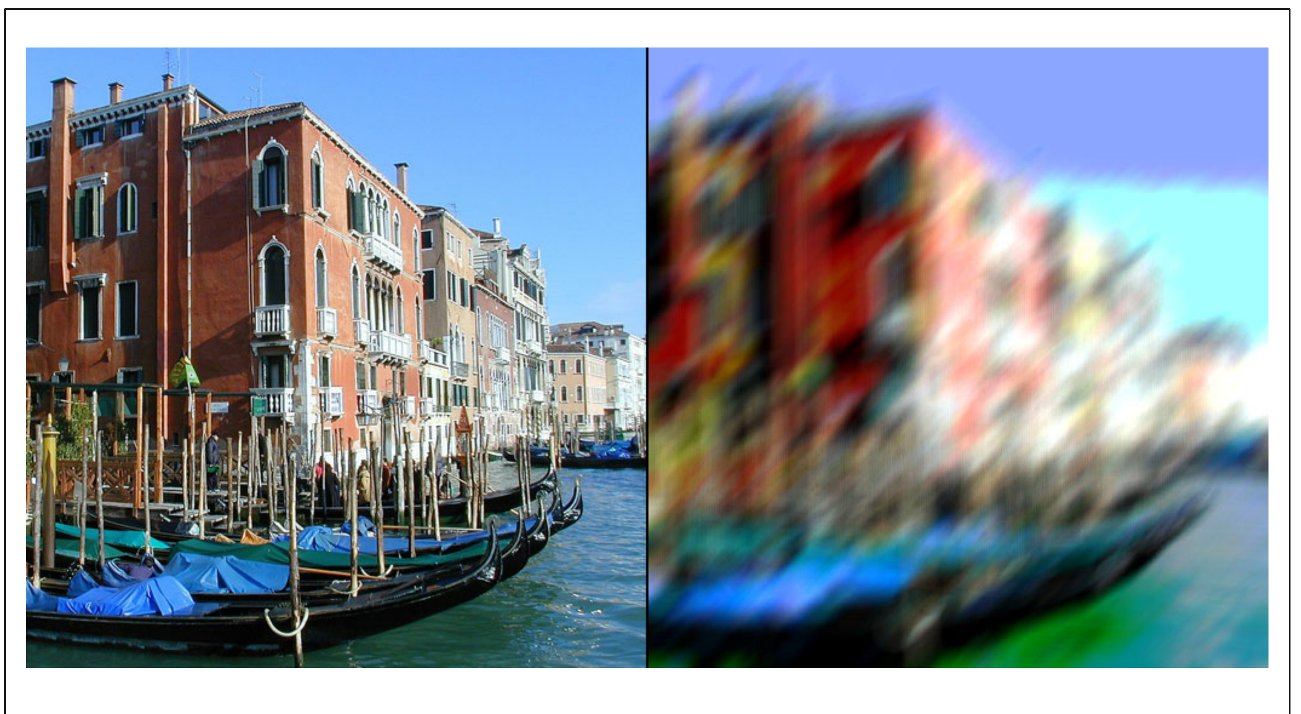


Figure 6.8: Anisotropic filtering.

quality when the original texture is mapped on the object using linear interpolation or higher-order filtering with a cubic B-spline filter. Some frames of both post-processing examples are seen in figure 6.9.

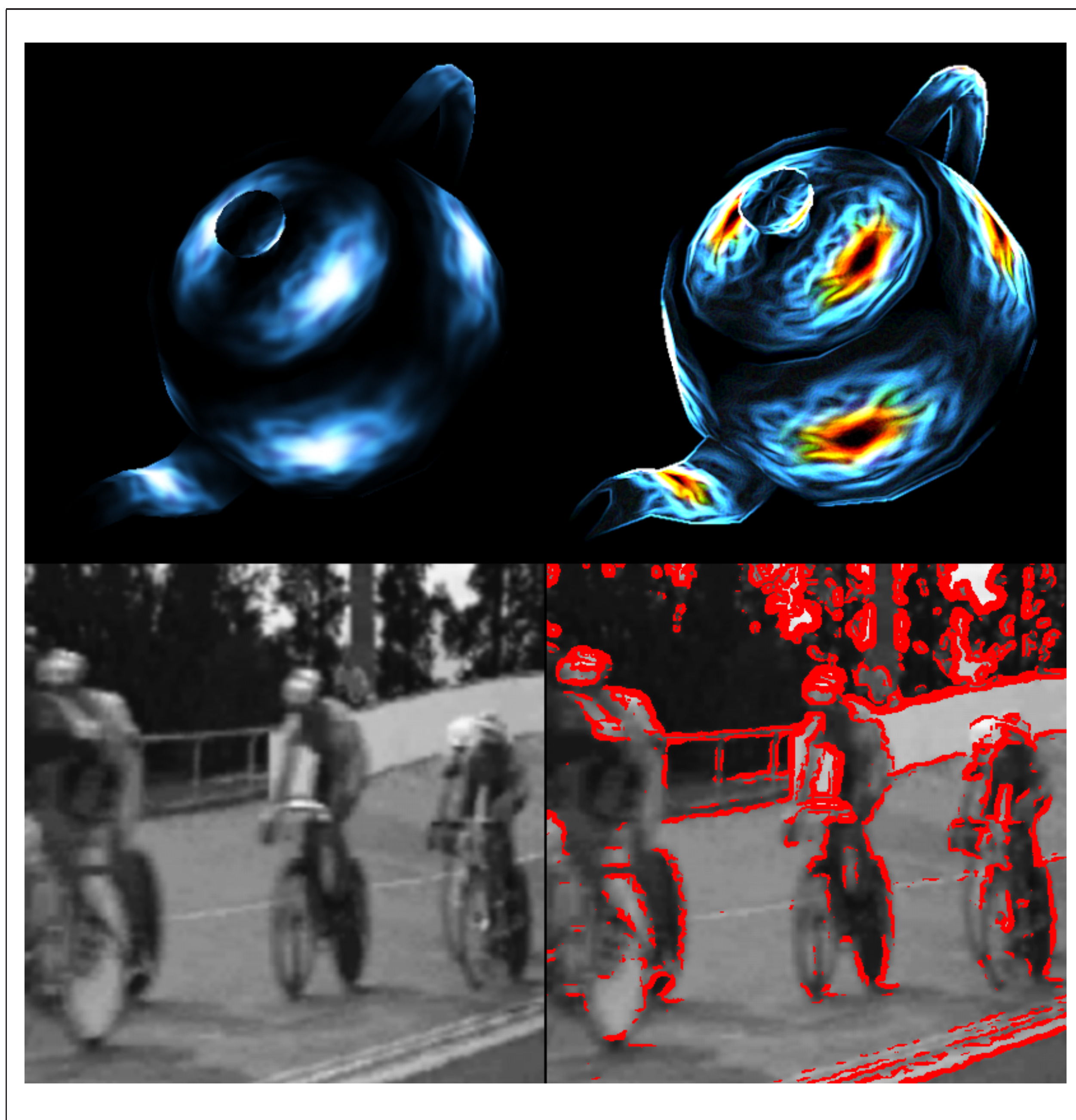


Figure 6.9: Built-in real-time post filtering techniques.

6.3 Problems and Solutions

This section reviews fundamental shortcomings of current hardware, which make the hardware-based image processing more complicated and which have an impact on the overall performance.

Framebuffer-related problems

The shortcomings of the framebuffer complicate the processing similar to high-resolution filtering. However, while in the high-resolution case the first problem was the framebuffer range and then precision, in image processing we have most problems with framebuffer precision. The problem with its limited range can be solved relatively easy. The low eight bit frambuffer precision causes that filtering with filters of bigger width than seven in each direction becomes unusable without any additional quality improvements. This problem is solved using hierarchical summation. We will explain the process using an example. If we use a kernel of width seven, we have 49 filter passes. In case of averaging each weight is the reciprocal of 49. The hardware represents this value with eight bits, so the sum of all filtering passes is not equal to 1. This leads to visible intensity shifts. So instead of filtering with this value we filter a particular number of passes (for example four), with weights scaled to four times the reciprocal of 49 into a special buffer which has the same properties as the framebuffer, called pbuffer. Rendering into the framebuffer is performed with the summed result of these four passes as input texture and reciprocal of four as weight. After this, the framebuffer contains the filtering subresult of four passes with much higher quality as filtering directly into the framebuffer. This is done for all filtering passes in a loop. Additionally the filtering process can exploit the internal hardware precision by using all possible texture units in a single rendering pass.

The framebuffer range problem is solved by simply splitting weights into two or more weights so that each one is within the hardware interval. We reorder the filtering process by inserting negative filtering contributions between splitted positive contributions and vice versa to keep possibly intermediate computations within the hardware interval. As all such filter kernels are of low width it is easy to hard-code the order of filtering passes, i.e., to specify the shifting and weight lookup tables, and achieve good quality.

Rendering to a texture

This problem is not of comparable importance as the problems with framebuffer shortcomings, it is more or less a problem of current driver versions. Nowadays graphics boards support so called direct rendering into textures. But the OpenGL driver implementation of NVidia graphics chips does not support it. All the image processing filtering was coded for NVidia hardware. At the moment rendering is not directly done into a texture. We render into the framebuffer or pbuffer and then the rendering result is copied into the texture. This is a time-consuming operation, which is also often used in reducing the framebuffer precision problem solution, i.e., in hierarchical summation. Therefore, the filtering time is almost twice as long as without hierarchical summation. This problem will be solved hopefully soon with new NVidia drivers.

Chapter 7

Implementation

> The only language to which all programmers are perfectly familiar with, are bad words.
>
> *Troutman's 6. Law of Programming [4]*

The implementation of the filtering methods presented in this thesis was coded in the C programming language. We are using the OpenGL application programming interface to communicate with the hardware. Due to the procedural nature of OpenGL we do not use any object-oriented hierarchical structures. For some helper functions, to compute for example framerates, we use a framework called OFVis. In the next parts we will discuss some sections of the implementation code to describe basic principles from the programmer's side of view.

7.1 Kernel representation

In this section we present various ways of storing the filter kernel information. The high-resolution filters are using a structure `kernelinfo_s` defined in table 7.1. The record kernel stores the kernel

```
struct kernelinfo_s {
  float*  kernel;          // kernel memory
  int     dimensions;      // number of dimensions
  short   numtiles;        // 1 == not tiled
  short   numsplittiles;   // 0 == no tile split
  int     size;            // numtiles * tilenumfloats
  int     tilelen;         // tile length one dimension [entries]
  int     rownumfloats;    // tile length one dimension [floats]
  int     slicenumfloats;  // tile length two dimensions [floats]
  int     tilenumfloats;   // tile length all dimensions [floats]
};
```

Table 7.1: The `kernelinfo_s` structure.

values in a one-dimensional array. Record `dimensions` determines whether it is a one-, two-, or three-dimensional kernel. If the filter is of width two or greater, we have to split it into several tiles. The number of such tiles is given by the `numtiles` record. To reduce numerical clamping error, we split some tiles in two to allow interleaving and put some negative filter passes between them. The number of "split" tiles is given by record `numsplittiles`. The records `size`, `tilelen`, `rownumfloats`, `slicenumfloats` and `tilenumfloats` describe the length of the tiles and the size of the kernel. To generate a higher-dimension (separable) kernel we create it as a product of two lower dimensional ones. This is done either in the preprocess or on-the-fly.

The image processing filters are always two-dimensional, we do not use any "split" tiling and the values and the processing order is either explicitly given or is not important. We store the image processing filters in lookup tables. There are three types of kernel generation. The first one is shown by the use of the Gaussian smoothing operator, which is generated on-the-fly by the function `Gaussian` in table 7.2, computing a value for each kernel element.

```
GLfloat Gaussian (int x, int y) {
   GLfloat tmpwidth =
      ((2*x - filterwidth)/((float)filterwidth))*4*SIGMA;

   GLfloat tmpheight =
      ((2*y - filterheight)/((float)filterheight))*4*SIGMA;

   return ((1/(2*M_PI*SIGMA*SIGMA)) *
      pow(M_E,-(tmpwidth*tmpwidth+tmpheight*tmpheight)/
      (2*SIGMA*SIGMA)));
}
```

Table 7.2: The `Gaussian` function.

Another type of filter generation is to read it from a file. We use the *Targa* file format to store the kernel information. This has the disadvantage that we cannot store filters with negative values. These types of filters are used as anisotropic filters, for example, comfortably designed in an image editing and processing program. In case of RGB or RGBA textures we are able to design custom filters for each kernel separately, which can produce various interesting effects.

The last type of image processing filter generation is the explicit storage in a constant array. We show the example of two Laplacian kernels, one with low edge response (due to low weights) and the second one with high edge response. Both are stored to be used with the OpenGL imaging subset (`laplace0`) and with texture-based filtering (`laplace1`). The first array is stored "as is", while the second has reordered filter passes to stay in the hardware range of [0, 1]. Filtering passes are processed one after another exactly as they are stored in arrays shown in table 7.3. Note, that the high-response Laplacian filter has the contribution of the middle weight stored in eight -1 values. These are mixed with positive contributions of other weights to stay within the hardware range. The ordering of filter weights have to be synchronized with the shift lookup table, which is the task of the filtering management.

```
GLfloat laplace0[9] =
   {0.111, 0.111, 0.111, 0.111, -0.888, 0.111, 0.111, 0.111, 0.111};
GLfloat laplace1[9] =
   {0.111, 0.111, 0.111, 0.111, 0.111, 0.111, 0.111, 0.111, -0.888};


GLfloat laplacehigh0[9] =
   {1, 1, 1, 1, -8, 1, 1, 1, 1};
GLfloat laplacehigh1[16] =
   {1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1};
```

Table 7.3: Laplace "hard-coded" arrays of low and high edge response.

7.2 Filtering control

The filtering process is managed via lookup tables. We show at least the most important ones used for high-resolution filtering which manages surface filtering. The following arrays in table 7.4 are used for direct management of each filtering pass.

```
GLint      surf_permut_to_passid[];
GLint      surf_passid_to_tileid[];
GLint      surf_passid_to_tilemirr[];
GLint      surf_passid_to_texofs_x[];
GLint      surf_passid_to_texofs_y[];

GLint      surf_tile_to_tilesign[];
GLint      surf_tile_to_tileswap[][2];
```

Table 7.4: High resolution filtering control arrays.

The `surf_permut_to_passid[]` table determines the pass order. This is done due to the possibility to vary filter passes to achieve low numerical clamping error. The arrays `surf_passid_to_texofs_x[]` and `surf_passid_to_texofs_y[]` are lookup tables that store input texture offsets for each filter pass. The corresponding filter tile number is stored in `surf_passid_to_tileid[]`. When the filter symmetry property is exploited, we use additional arrays to control the mirroring and swapping of the filter tile, i.e., `surf_passid_to_tilemirr[]`, `surf_tile_to_tilesign[]`, and `surf_tile_to_tileswap[][2]`.

The image processing filtering process is controlled only by two lookup tables, because we do not care about symmetry and the offsets are stored in a single one-dimensional array. In the case of filters with positive weights only, we compute the shift offsets from the distance of the particular weight to the middle weight. In case of 3×3 edge detectors we "hard-code" the input texture offsets in the

same order and style as the filter weights. The only difference is that we store two values instead of one. The shift arrays are shown in table 7.5. Note that every second pair in the shift lookup table of the Laplacian high edge response variant is $0, 0$. This corresponds to the splitted middle weight.

```
GLint laplace_order[18] =
   {-1, -1, -1, 0, -1, 1, 0, -1, 0, 1, 1, -1, 1, 0, 1, 1, 0, 0};

GLint laplace_high_order[32] =
   {-1, -1, 0, 0, -1, 0, 0, 0, -1, 1, 0, 0, 0, -1, 0, 0,
   0, 1, 0, 0, 1, -1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0};
```

Table 7.5: Laplace “hard-coded” shift arrays of low and high edge response.

Chapter 8

Summary

One of the fundamental tasks of computer graphics as well as image and signal processing is how to process sampled data to get some desired result. Signal processing in general is dealing with one-dimensional (sampled) functions such as audio signals. It is trying to reconstruct the original continuous function from its discrete representation. Image processing applications are processing mostly two-dimensional (sampled) functions, representing images. This area does not reconstruct the original function, but rather tries to process the given samples to emphasize some features. Computer graphics reconstructs the continuous function from one-, two-, and three-dimensional samples. The two-dimensional sampled functions are raster images, as in the image processing case. Three-dimensional samples represent, for example, either computer tomography (CT), or magnetic resonance (MRI) datasets, that describe the interior of human bodies, or other volumetric objects that can only be sufficiently described in 3D. CT and MRI scans are used in medical visualization to improve the medical diagnosis process. Typical examples of other 3D objects are semi-transparent objects like fog, smoke, clouds, or objects that are hard to represent without describing their internal structure. This is the case with marble and wooden objects.

The above mentioned one-, two- and three-dimensional datasets are represented by discrete samples. These samples must fulfill particular conditions of sampling theory, which is dealing with two fundamental tasks: sampling and reconstruction. Sampling describes how dense the original function should be sampled to be sufficiently described by its discrete representation. Reconstruction theory describes how to get the continuous function from its discrete samples [24, 42]. The reconstruction process usually is defined as a convolution of the discretely sampled function with a reconstruction kernel. This kernel could be continuous, but in practice we also use discrete filters of high sampling resolution. If the original function was band-limited before it was sampled and if sampling was done with an appropriate sampling frequency we can perfectly reconstruct it using the *sinc* function as filter kernel [33]. The problem of the sinc filter is that it is unusable in practice due to its infinite support. Therefore several approaches have been introduced to perform high-quality reconstruction, based on approximations of the sinc filter on a limited interval. Keys [16] derived a family of cardinal splines suitable for reconstruction purposes, classifying the Catmull-Rom splines as numerically most accurate. Mitchell and Netravali [18] derived another category of cubic splines called BC-splines that are very popular as reconstruction kernels as well. They have classified various types of these splines according to the B and C parameters and determined boundaries of admissible values. Theußl et al. [35, 36] classified various windowing functions that limit the sinc extent to a particular interval, showing that the Kaiser, Blackman, and Gauss windows feature best properties in the frequency do-

main. Turkowski [37] presents a general overview of filter kernel types, concluding that the windowed sinc kernels are best.

The basic filtering operation, i.e., convolution, can be implemented in various ways. The standard implementation is based on the *gathering principle*. This means, the final value of a particular output sample is gathered from the weighted neighboring input samples simultaneously. In other words, the convolution operation is processed in a serial way, one output sample after another. Although this approach seems to be efficient in general, we will show another possibility of how to access the filtering principle using another evaluation order. Instead of computing one output value after another, we can compute the contribution of a particular input sample to all corresponding output values. The difference is, that instead of "gathering" the contributions for each output sample, we "distribute" the contributions of the input samples, therefore we call it *distribution principle*. These distributed input sample contributions are summed up for each output sample. This approach is used, for example, in all splatting-based rendering techniques [41]. Both convolution principles are illustrated in figure 4.1.

Filters providing high-quality reconstruction are usually implemented in software. The algorithms require long computation time, which is far from real-time, to process the whole dataset. This makes it impossible to integrate them into an interactive application with on-the-fly resampling.

The highest-quality reconstruction filter supported natively by hardware is the tent filter, yielding linear interpolation. The hardware is able to use the tent filter in 2D as well as 3D, achieving bi- and tri-linear interpolation. The filter provides acceptable results in the case when the sampling frequency of the data is high enough. The performance issue is the big advantage of this type of interpolation. However, in the case of lower sampling frequency, the filtering results in strongly visible artifacts.

Although current graphics chips are equipped with a lot of features, the problem of insufficient precision remains. Therefore, there are just few approaches possible for how to use the hardware for filtering. The methods are mostly dealing with image processing methods such as the algorithm introduced by James [15]. This approach exploits the texture hardware for image processing convolution. The only method using hardware for high-resolution filtering is proposed by Hadwiger et al. [8, 9, 10]. The image processing convolution by James can be considered as a subset of this general framework. More detailed description of both algorithms are presented in the next section.

Beside these methods another approach was proposed by Hopf and Ertl [13] using SGI [31] graphics hardware. They are using the OpenGL imaging subset [23] and texture hardware to extend the natively supported 2D convolution to 3D. A year later they introduced non-linear hardware-accelerated image processing with erosion and dilation operators [14]. The reason why we do not use the OpenGL imaging subset is its insufficient performance on consumer hardware, as well as its restriction to image processing tasks.

8.1 Hardware-based filtering using textures

Graphics chips are designed to perform primitive mathematical per-fragment operations for all fragments simultaneously. This makes it possible to accommodate the distribution filtering principle in hardware. The method requires graphics hardware with at least two texture units. The sampled input function is stored in one texture and the discretized filter kernel in another one. The kernel texture is scaled to cover exactly the contributing input texels. To be able to perform the same operation for all input samples at once, we divide the kernel into several parts, called *tiles*, that always cover only

one input texel width. These filter tiles are repeated in hardware, so all input texels are covered by the same filter tile. We illustrate the method using the example of the tent filter. The tent filter is of width two, i.e., it uses two input samples as contributions for each output sample. Instead of taking the whole filter kernel, we take its left tile first. This is scaled exactly to the input texel width. To compute the "left" contribution of each input sample, we shift the input texture to the left by one half of the input texel width. After this, each input sample covers exactly the left tile of the filter kernel. Now we have the input function in one texture and the kernel tile of width one in the second one. The kernel texture is set to repetition, so the kernel replicas cover the whole input texture - each input sample. We set the numerical operation between these two textures to multiplication and render it to the framebuffer. This subresult is the left tile contribution of each input sample to the output resampling points. We analogously repeat this process with the right filter tile, set the framebuffer blending function to addition and render the right contribution to the framebuffer again. The whole filtering result is now stored in the framebuffer. The distribution of the left and right tile contribution is illustrated in figure 4.2.

This simple example explains the general filtering approach using arbitrary filter kernels. For better illustration we explain the filtering idea with a pseudocode dividing the process into the following logical parts.

- **Global setup** is the preprocessing step done in software. At first we generate the filter kernel and the lookup tables that control the filtering process. The lookup tables store information about which filtering step uses which filter tile and how the input texture should be shifted.
- **Per-pass setup** selects the filter tile according to the corresponding filtering pass. This tile is assigned to a texture unit, similar to the source texture. The difference is only in the texture setup. While source texture is set to clamping, the filter texture is set to repetition.
- **Vertex setup** stage realizes the shift operation of the input texture and scaling of filter tiles is performed. For performance reasons the vertex shaders hardware is involved in this step.
- **Pixel setup** stage is also called according to the hardware part where it is processed, namely pixel shader. It assigns both textures to shader registers and computes the multiplication of these two registers.

The general filtering approach illustrated on the tent filter is described by the pseudocode in table 4.1. The pseudocode uses the operator *shift_j*, denoting that the input texture is not actually indexed differently at each output sample, but instead the input texture is shifted according to the particular rendering pass.

8.2 Filtering algorithms

The general filtering algorithm uses two textures simultaneously. In the case when the hardware is able to use more than two textures in a single rendering pass, we can perform more filtering passes in one rendering pass. This increases the processing performance in some cases almost linearly. It leads also to qualitatively better results due to exploiting the hardware internal precision for the addition as well. The precision of the framebuffer is 8 bits, while the internal precision of the best consumer cards

of today is up to 12 bits. All the algorithms described below can be to expanded to take advantage of more available texture units or they can be combined together to achieve optimal results. Each algorithm has its own notation - the first three letters identify the filtering algorithm and the rest describe the number of used texture units within the rendering pass.

- **The standard algorithm** is in fact identical with the general filtering approach and is denoted as *std-2x*. It uses general filter kernels without exploiting any filter characteristics and it is shown in table 4.1.
- **Symmetric filter kernels** can be used to reduce the hardware memory consumption. Instead of storing whole kernels, we just store a subset and generate the rest by mirroring existing parts. The algorithm does not have any performance influence, therefore we do not use any special notation, it is used for memory saving purposes, which is especially significant in the 3D case. The symmetric kernel extension is shown in table 5.1.
- **Separable filter kernels** allow to store the filter kernel in lower dimensional parts, which are multiplied on-the-fly producing the respective filter kernel. The texture traffic is much lower in this case, which has significant impact on the performance, but it involves more texture units than the standard case. We denote this algorithm by *sep-3x*. It is shown in table 5.2.
- **Pre-interleaved monochrome input** is used to exploit the per-channel dot product in the pixel shader. This method assumes monochrome and interleaved textures as input. Each pixel stores its “own” value in the R channel, the other three channels are reserved for the next three pixel values toward the right from the current one. The filter kernel stores the sampled filter in the same style. This allows to compute four contributions to the current output sample accessing only one input sample. The notation for this algorithm is *dot-2x*, or combined with separable kernels *spd-3x*. This algorithm is described in table 5.3.

The filtering process discussed up to now was a generalization of all convolution-based filtering methods. However in the special case of image processing, where the input and output sample grid is exactly the same, we can use James’ algorithm. Instead of using texture tiles for representing the filter kernel, we use singular color values. Image processing filters are very rough approximations of the underlying continuous functions. They represent each part or tile only by one value. Therefore we can substitute the filter texture with color values. The difference between a continuous, a high-resolution filter, and an image processing filter is shown in figure 4.3. The reason of using color values instead of textures is saving texture traffic as well as texture units, which consequently can be used to fold more filtering passes into a single rendering pass. We show the analogous algorithm to the general filtering approach for image processing tasks in table 6.1. The “dot” algorithm can be used for image processing tasks as well, but the algorithms exploiting filter kernel properties are irrelevant, because the whole image processing filter is stored in a lookup table in just a couple of float values.

8.3 Applications

This section reviews possible applications of the hardware-based filtering. Firstly, we address high-resolution filters and then some image processing. The biggest importance of the hardware filtering principle is its generality, i.e., it is possible to use any type of filter to achieve high-quality results in real-time.

Surface textures

The first application area of high-resolution filtering is surface texturing. We use higher-order filters of width four, namely cubic B-splines, Catmull-Rom splines, and Kaiser-windowed sinc of width four. Using such filters is especially effective, when the input texture is sampled at low frequency. Reconstructing it using hardware-native linear reconstruction results in visible artifacts. Software-based higher-order reconstruction would not achieve sufficient filtering performance. The typical low texture resolution representatives are lighting effects, such as lightmaps [1]. We show the results of various filter types in figure 5.1. The performance of the state-of-the-art graphics cards using various filtering algorithms is shown in the tables 5.4 and 5.5. The tested texture is of 64×64 resolution.

Solid textures

In the case of solid textures we are dealing with the same higher-order filters as in the 2D case, however they are three-dimensional filters. This texturing approach is used in cases, when the two-dimensional description does not provide acceptable results. Some examples are marble and wood materials [26, 27]. To be able to store more 3D textures in the hardware memory, we have to store them in low resolution. To obtain high-quality results from such datasets, higher-order filtering must be performed. Another application area of growing importance is volume rendering [30, 40]. To avoid linear reconstruction artifacts, the kernels of higher-order can be involved again. Tables 5.6 and 5.7 show the performance of various algorithms on a 128^3 dataset. This dataset is shown in figure 5.2 comparing tri-linear to tri-cubic interpolation.

Animated textures

High-quality reconstruction is even more important when using animated textures. Linear interpolation produces artifacts, which are stronger visible in moving pictures. This is because the underlying interpolative grid appears as static layer beneath the moving texture. Higher-order filtering like cubic interpolation completely removes such artifacts. This effect is most visible in animations with rotating objects. We divide the animation into the following three types according to their generation stage.

- **Pre-rendered animation** are set of frames which are computed in a preprocessing step. To be able to perform real-time animation, we have to store all frames as textures, which poses extreme requirements to the hardware memory capacity. The solution can be to store the frames in lower resolution and reconstruct them using higher-order filters.
- **Procedural CPU animation** produces frames which are generated on-the-fly. Each frame, once it is computed, is transferred to the graphics hardware and displayed as a mapped texture. The transfer and generation stage are the most time-consuming operations. Generating and transferring lower-resolution textures significantly improves the performance.
- **Procedural GPU animation** is a similar technique as the procedural CPU animation but has one advantage. The frames are produced in the graphics hardware and do not need to be transferred to graphics memory, because they are already there. However, such animations are much more limited in the generation stage than CPU-generated animations. The reason of

low-resolution sampling is the same as in the previous case, i.e., to speed-up the generation process.

We show the difference between tri-linear and tri-cubic filtering on a pre-rendered animation frame of variable texture resolution in figure 5.3. Figure 5.4 shows a fire animation, the first one is a pre-rendered animation (top), the other one is generated on the GPU (bottom).

Derivative filtering

The previous applications used various function reconstruction filters. The generality of the filtering algorithm allows to implement derivative reconstruction in hardware as well. This allows to compute gradients on-the-fly, which are mostly computed in a preprocessing step in real-time applications. Although the simplest software solution, i.e., central differences, computes gradients in a short time, it produces visible staircase artifacts. This effect is significantly reduced by using higher-order and high-resolution kernels for derivative reconstruction.

Smoothing

Smoothing is the first example in the image processing area. The high-resolution filtering implementation used filters of fixed width of four, although it can be extended to arbitrary widths. The image processing filtering is extended to filters of arbitrary width. The typical filters of variable width are smoothing operators used for noise reduction by cutting off the high frequencies in the image. We use two types of smoothing filters, i.e., simple averaging and a Gaussian filter, which is based on the Gaussian lobe function. Table 6.3 shows performance of the smoothing process on an image with 512×512 resolution. Filtering with kernels of width seven and higher results in strong visible quantization artifacts. Therefore we are using quality improvement, i.e., hierarchical summation, that provides acceptable results. The framerates, however, are about half of those from the benchmark table. We show the difference of filtering with and without quality improvements in figure 6.1.

Edge detection

The second type of image processing filters are edge detectors. These are applicable in almost all pattern recognition and computer vision areas, usually requiring real-time performance. This is hard to achieve in software, without exploiting any hardware. Our implementation uses two types of edge detectors, i.e., a Sobel and a Laplacian operator. The Sobel filter approximates first-order derivatives and the Laplacian filter second-order derivatives. The Laplacian filter uses only one convolution mask for the filtering process and is faster than the Sobel filter. It describes the magnitude of the edge response only, so it is not suitable for gradient direction estimation. The Sobel filter consists of two or more kernels, for each dimension at least one. These filters have more variations. The benchmark table 6.4 shows filtering performance on an image of 512×512 resolution for all of them. The Laplacian filter has two variations with low weight values and high weight values. This has an impact on the resulting edge visibility. Similar to the Laplacian one, the Sobel filter is also presented with low and high weight values. Other Sobel versions use more than two filter kernels to improve the edge detection process. Also these filters give an acceptable performance. The results of the edge detectors using OpenGL imaging subset and texture based filtering is shown in figure 6.4.

Artistic rendering

Beside fundamental convolution-based operations, there are other arbitrary filters used in image editing and processing applications. We show that exploiting graphics hardware and combining various features, we are able to implement non-photorealistic rendering techniques with real-time performance.

- **Painting with enhanced edges** is a technique, that combines the result of edge detectors and smoothing in the pixel shader to create customizable painting-like results. If we turn off the smoothing we get the original image with enhanced edges, which can be used for pattern recognition purposes as well.
- **Filter combinations on a pre-masked image** assume a segmented image as input. We use the alpha channel as segmentation mask. We filter the image several times using various kernels and combine all together using the alpha test of the source image.
- **Pointilism-style painting** uses a randomly generated noise mask during the stencil test to decide which texel from two textures will be set at a particular position. The generation of the noise mask is the only part of the process done on CPU. The original input image is combined with a smoothed one to create a pointilism-like effect.
- **Anisotropic filtering** uses non-symmetric kernels to simulate one-directional brush strokes.

All these effects can be seen in figures 6.5, 6.6, 6.7, 6.8.

Post-filtering

All the image processing filters mentioned above can be integrated in various applications. The idea of post-filtering is to integrate such filtering directly into a particular process that generates images. But instead of transferring them immediately to the display, we process them with our image processing filters. This could be considered as “screen space” processing. The typical application, for example, is integrating the non-photorealistic rendering technique in a standard renderer, or to filter the output from a CCD camera for video surveillance purposes. These examples are shown in figure 6.9.

Conclusions

The hardware-based filtering approach – dealing with high-resolution filters for reconstruction and linear operators for image processing tasks – shows that the hardware can be involved not only in processing the rendering pipeline in hardware, but for a lot of other tasks that somehow process the graphical information. This work shows a subset of features on filtering applications, but these features can be exploited in any other application area as well. The most flexible part of the graphics hardware turns out to be vertex and pixel shaders.

It is clearly visible that the hardware manufacturers focus not only on the real-time rendering itself, but on other graphical time-critical applications. A good example is the OpenGL imaging subset support. This is still implemented in the driver on low-end graphics cards [22]. But the high-end solutions [31] can process convolution in real-time [13]. The internal precision and range of the graphics chips offer a wide spectrum of applicability as well. However, the framebuffer is still limited to the range of $[0, 1]$ represented by eight bits. This will be changed in the near future.

Unfortunately there are a lot of features that are still not standardized in the OpenGL API, and therefore are hardware specific. Therefore the same functionality, e.g, implemented in the pixel shader, has to be implemented for each graphics chip separately. This is quite bothering from the developer's point of view, but it is natural that the standard is always "one step back".

The hardware still offers nearest-neighbor and linear interpolation reconstruction. Using high-order reconstruction filters implemented in hardware is a good compromise between performance and quality especially when the temporal aspect is taken into the account. This is clearly visible an the animated textures example. The linear interpolation produces much more visible artifacts, because the underlying interpolation grid appears as a kind of static layer beneath the moving texture.

The shortcomings of the framebuffer, i.e., precision and range, introduce clamping errors on the filtered images when the partial sum of the filtering contributions exceeds the admissible interval. The low precision leads to quantization artifacts which are especially visible in the image processing tasks. The internal precision of vertex shaders is also not sufficient enough on some hardware and does not guarantee pixel-exactness. Therefore a lot of simple operations have to be done in the pixel shader.

Bibliography

[1] M. Abrash. Quake's lighting model. In *Graphics Programming Black Book*, pages 1245–1256. Coriolis Group Books, 2001.

[2] K. Akeley. RealityEngine graphics. In *Proceedings of SIGGRAPH '93*, pages 109–116. ACM, 1993.

[3] ATI web page. http://www.ati.com/, 2002.

[4] A. Bloch. *The Complete Murphy's Law*. Price Stern Sloan, Los Angeles, 1990.

[5] R. N. Bracewell. *The Fourier Transform and its Applications*. Price Stern Sloan, Los Angeles, 1999.

[6] J. W. Cooley and J. W. Tukey. Algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[7] D. S. Ebert. Animating solid spaces. In *Texturing and Modelling: A Procedural Approach*, pages 163–191, 1994.

[8] M. Hadwiger, T. Theußl, H. Hauser, and M. E. Gröller. Hardware-accelerated high-quality filtering of solid textures, technical sketch. In *SIGGRAPH 2001 Conference Abstracts and Applications*, page 194, 2001.

[9] M. Hadwiger, T. Theußl, H. Hauser, and M. E. Gröller. Hardware-accelerated high-quality filtering on PC hardware. In *Proceedings of Vision, Modeling, and Visualization 2001*, pages 105–112, 2001.

[10] M. Hadwiger, I. Viola, and H. Hauser. Fast convolution with high-resolution filters. Technical Report TR-VRVis-2002-001, VRVis Research Center, Vienna, Austria, 2002.

[11] DirectX home page. http://www.microsoft.com/directx/, 2002.

[12] Pixar Corporation home page. http://www.pixar.com/, 2002.

[13] M. Hopf and T. Ertl. Accelerating 3D convolution using graphics hardware. In *Proceedings of IEEE Visualization '99*, pages 471–474, San Francisco, 1999.

[14] M. Hopf and T. Ertl. Accelerating morphological analysis with graphics hardware. In *Proceedings of Vision, Modeling, and Visualization 2000*, pages 337–346, 2000.

[15] G. James. Operations for hardware-accelerated procedural texture animation. In *Game Programming Gems 2*, pages 497–509. Charles River Media, 2001.

[16] R. G. Keys. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(6):1153–1160, 1981.

[17] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[18] D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer graphics. In *Proceedings of SIGGRAPH '88*, pages 221–228, 1988.

[19] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. Evaluation and Design of Filters Using a Taylor Series Expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–199, 1997.

[20] T. Möller, K. Mueller, Y. Kurzion, R. Machiraju, and R. Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *IEEE Symposium on Volume Visualization*, pages 143–151, 1998.

[21] J. S. Montrym, D. R. Baum, D. L. Dignam, and Ch. J. Migdal. InfiniteReality: A real-time graphics system. In *Proceedings of SIGGRAPH '97*, Annual Conference Series, pages 293–302. Addison Wesley, 1997.

[22] NVIDIA web page. http://www.nvidia.com/, 2002.

[23] OpenGL web page. http://www.opengl.org/, 2002.

[24] A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing*. Prentice Hall, Englewood Cliffs, 1975.

[25] Parsec - there is no safe distance web page. http://www.parsec.org, 2002.

[26] D. R. Peachey. Solid texturing of complex surfaces. In *Proceedings of SIGGRAPH '85*, pages 279–286, 1985.

[27] K. Perlin. An image synthesizer. In *Proceedings of SIGGRAPH '85*, pages 287–296, 1985.

[28] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. In *Proceedings of SIGGRAPH '99*, pages 251–260. ACM Press, 1999.

[29] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.

[30] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop 2000*, 2000.

[31] SGI web page. http://www.sgi.com/, 2002.

[32] R. W. Shafer and L. R. Rabiner. A digital signal processing approach to interpolation. *Proceedings of IEEE*, 69(6):692–702, 1973.

[33] C. E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 29(4):10–21, 1949.

[34] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis and Machine Vision*. PWS Publishing, 1995.

[35] T. Theußl. Sampling and reconstruction in volume visualisation. Master's thesis, Institute of Computer Graphics, Vienna University of Technology, Vienna, Austria, 1999.

[36] T. Theußl, H. Hauser, and M. E. Gröller. Mastering windows: Improving reconstruction. In *Proceedings of Symposium on Volume Visualization '00*, pages 101–108, 2000.

[37] K. Turkowski. Filters for common resampling tasks. In *Graphics Gems I*, pages 147–165. Academic Press, 1990.

[38] 3dfx Interactive, Inc. web page. http://www.3dfx.com/, 2002.

[39] 3DLabs Corporation web page. http://www.3dlabs.com/, 2002.

[40] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH '98*, Annual Conference Series, pages 169–178. Addison Wesley, 1998.

[41] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH '90*, pages 367–376, 1990.

[42] G. Wolberg. *Digital image warping*. IEEE Computer Society Press, 1990.

Acknowledgments

This work has been done in the scope of the basic research on visualization (`http://www.VRVis.at/vis/`) at the VRVis Research Center in Vienna, Austria (`http://www.VRVis.at/`), which is funded by an Austrian research program called Kplus.

Special thanks to Markus Hadwiger, Helwig Hauser, and Meister Eduard Gröller for patient supervision. Thanks to Thomas Theußl and Jiří Hladůvka for help with theoretical background. Thanks to Robert Tobler for nice pictures I use in my thesis (see photo gallery `http://ray.cg.tuwien.ac.at`) ...

... and of course special special thanks to my parents for the general supervision and support.

Curriculum vitae

Personal data

Ivan Viola
born on June 25th, 1977, in Bratislava, Czechoslovakia.

Address:
Kráľovské údolie 25, 81102 Bratislava, Slovakia
Phone: +421 (903) 770625, +43 (699) 12618865
eMail: viola@cg.tuwien.ac.at

Education

since 10/1997	Vienna University of Technology, Computer Science, Vienna, Austria
10/1995–06/1997	Slovak Technical University, Faculty of Electrical Engineering and Informatics, Bratislava, Slovakia
09/1992–06/1995	High school, Gymnasium Vazovova, in Bratislava, Slovakia
09/1986–06/1992	Elementary school Drieňová, in Bratislava, Czechoslovakia
09/1983–06/1986	Elementary school Odborárska, in Bratislava, Czechoslovakia

Jobs

09/2001–01/2002	Teaching assistant, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria.
03/2001–06/2001	Teaching assistant, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria.
since 11/2000	Technician and web designer, Art Film Festival, Trenčianske Teplice, Slovakia.
09/2000–01/2001	Teaching assistant, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria.
07/2000–08/2000	Holiday job, Siemens AG, in Vienna, Austria.
02/2000	Holiday job, Siemens AG, in Vienna, Austria.
07/1999	Holiday job, Siemens AG, in Vienna, Austria.
07/1998–09/1998	Holiday job, Siemens AG, in Vienna, Austria.

Languages

English	fluently
German	fluently
Slovak	natively

Hobbies Photography, film art, music, bass guitar, books, jogging, snowboarding, trekking