Fast and Flexible High-Quality Texture Filtering With Tiled High-Resolution Filters

Markus Hadwiger* Ivan Viola* Thomas Theußl† Helwig Hauser*

* VRVis Research Center
Vienna, Austria
† Institute of Computer Graphics and Algorithms
Vienna University of Technology, Austria

Abstract

Current graphics hardware offers only very limited support for convolution operations, which is primarily intended for image processing. The input and output sample grids have to coincide, making it impossible to use these features for more general filtering tasks such as image or texture resampling. Furthermore, most hardware employs linear interpolation for texture reconstruction purposes, incurring noticeable artifacts. Higher-order interpolation via general convolution is able to remove most of these artifacts. However, real-time applications currently do not consider higher-order filtering due to lack of hardware support. We present algorithms for extremely fast convolution on graphics hardware, which are able to deploy high-resolution filters, i.e., filters that are sampled at a much higher resolution than the width of the filter kernel. This framework can be used for general convolution tasks, but is especially suited to substituting the native bi-linear or tri-linear interpolation currently used for texture magnification, while still achieving frame rates of up to 100 frames per second for full screen filtering with bi-cubic interpolation.

1 Introduction

Convolution is the fundamental operation of linear filtering, where a filter kernel is passed over the input data, and for each desired output sample a weighted average of input samples that are covered by the filter kernel is computed. This process can be used for various tasks, ranging from image or volume processing to filtering for resampling purposes. Unfortunately, current hardware supports only convolutions where the input and output sample grids coincide. In such a scenario, a convolution kernel consists of a relatively small number of weights located at integer locations, where for each output sample being calculated each of these weights corresponds to exactly one input sample. Naturally, such convolution filters cannot be used when the output sample grid has no restrictions with respect to location or size, i.e., in general image or texture resampling. In this case, high-resolution filters are needed, i.e., filters that have not only been sampled at integer locations, but at a much higher resolution than the width of the filter kernel. They are a high-resolution, albeit still discrete, representation of the original continuous filter kernel. For all theoretical purposes, however, we consider such a kernel as actually being continuous; in practice, we also employ reconstruction for the filter kernel itself.

Especially in texture mapping, the input data in principle has to be subjected to two filtering operations in order to resample the discrete texture space onto the likewise discrete screen space. First, the texture has to be reconstructed from the samples stored in the texture map using a reconstruction filter. After it has subsequently been warped to screen space according to the viewing projection, a prefilter has to be applied, before finally sampling onto the output pixel grid [4]. Naturally, in practice certain trade-offs have to be made for performance reasons. Where in theory the filtering process is the same for both magnification and minification, i.e., where a single texel covers many pixels or vice versa, these cases are usually distinguished and handled separately. In the case of magnification the ideal resampling filter is dominated by the reconstruction filter, whereas in the case of minification it is dominated by the warped prefilter. Current graphics hardware tackles this problem by using a combination of MIP mapping [25], and bi-linear or tri-linear interpolation. The former is used to approximate the prefilter and provide the input to the second stage, i.e., the reconstruction filter that performs linear interpolation in texture space. While most previous work has focused on improving the prefiltering stage [3, 10], the framework presented in this paper can be used to significantly improve the reconstruction stage. In fact, although we are focusing on magnification, it is possible to combine our algorithms with MIP mapping in order to handle the case of minification, and attain a combination of prefiltering and reconstruction.

The main contribution of this paper is two-fold. First, we present a general framework for very fast convolution with basically arbitrary high-resolution filter kernels in graphics hardware, whose applicability will increase even further with near-future hardware supporting higher frame buffer precision and range, and more simultaneous textures. The hardware itself is not required to support convolution operations natively at all. We present several convolution algorithms, which differ with respect to speed and quality trade-offs, the types of filter kernels they can handle, and whether or not they require the input texture to be pre-processed. We make use of state-of-the-art hardware features such as multi-texturing, pixel shaders, and vertex shaders. These capabilities are currently rapidly increasing in both power and flexibility due to the demand of real-time shading languages [18, 20], and will soon allow us to perform cubic texture filtering in a single rendering pass. The presented framework can be used to add *arbitrary programmable filter kernels* to any real-time renderer or shading language, especially for texture reconstruction. Second, we establish an already feasible high-quality substitute for the current de-facto standard of linear interpolation for hardware texture reconstruction, in the form of convolution with cubic high-resolution filter kernels. This approach is in line with the increased willingness and trend toward spending fill rate on rendering with higher quality [5]. We show that our framework is already able to perform bi-cubic texture filtering at up to 100 frames per second for full screen rendering.

Related work. Current graphics hardware has only very limited support for convolution. The OpenGL imaging subset [11] that has been introduced with OpenGL 1.2 can be used for image processing tasks, using 1D and 2D convolutions where the output and input sample grids coincide, and filter kernels are sampled at integer locations only. Building upon the imaging subset, Hopf and Ertl [6] have shown how to perform 3D convolutions for volume processing. Recent graphics hardware features like vertex and pixel shaders can be used for substituting the imaging subset with a faster approach [7], although this is more prone to precision artifacts. The framework we present in this paper can easily be combined with optional real-time image-processing filters. For these, we combine

* {Hadwiger,Hauser}@VRVis.at, http://www.VRVis.at/vis/
† theussl@cg.tuwien.ac.at, http://www.cg.tuwien.ac.at/home/

the standard approach [7] with hierarchical summation in order to reduce precision artifacts. The filter convolution sum is evaluated in reverse order than the one usually used in software-based convolution, which is also done by all splatting-based volume rendering techniques [24]. We build upon our earlier work on evaluating a high-resolution filter convolution sum in hardware [1]. However, we now propose a general framework realized with an entirely new implementation that is able to exploit more properties of the filter kernel, offers greater flexibility, leverages vertex and pixel shaders, and attains frame rates that are approximately ten times higher. Instead of being restricted to slice reconstruction, the algorithms we present can be used for texture mapping arbitrary polygonal objects in perspective, filtering static and animated textures, both pre-rendered and procedural, as well as both surface [2], and solid textures [17, 19]. The approach we present can be combined with MIP mapping [25], which is crucial to using it as full substitute for the usual linear interpolation. The two major factors contributing to the huge speed-up of the framework presented in this paper in comparison to our earlier results [1] are the exploitation of filter separability, and extensive use of pixel and vertex shaders. The combination of drastically improved performance and more flexibility and features now allows to consider using high-quality texture filtering in practice, where only linear interpolation has been used before.

High quality prefiltering techniques have been developed for both software [3], and hardware [10] renderers. Hardware prefiltering usually focuses on extending MIP mapping for anisotropic filtering in the case of minification, via footprint assembly [10], where several texture lookups at locations approximating the pixel footprint in texture space are combined. Although most of these methods require explicit hardware support, standard MIP mapping hardware can also be used for anisotropic filtering by accessing several MIP map levels, and compositing these samples via multitexturing or multiple rendering passes [14]. Our method also performs filtering by compositing several weighted samples. However, we are focusing on reconstruction for texture magnification instead of minification and handling anisotropy, and retrieve weights from a high-resolution filter kernel for each pixel.

Currently, interest in higher quality filtering of textures is resurging, especially in the field of point-based rendering [27]. Over the years, a lot of work in computer graphics has also been devoted to investigating high-quality reconstruction via convolution, although almost exclusively with software implementations. Keys [8] derived a family of cardinal splines for reconstruction and showed that among these the Catmull-Rom spline is numerically most accurate. Mitchell and Netravali [12] derived another family of cubic splines quite popular in computer graphics, the BC-splines. Marschner and Lobb [9] compared linear interpolation, cubic splines, and windowed sinc filters. They concluded that linear interpolation is the cheapest option and will likely remain the method of choice for time critical applications. Möller et al. provide a general framework for analyzing filters in the spatial domain, using it to analyze the cardinal splines, and the BC-splines [13]. They also show how to design accurate and smooth reconstruction filters. Theußl et al. [22] used the framework developed by Möller et al. to assess the quality of windowed reconstruction filters and to derive optimal values for the parameters of Kaiser and Gaussian windows.

2 High-res convolution in hardware

This section presents our framework for high-resolution convolution in graphics hardware. We begin the discussion by briefly summarizing key points of our method, before we describe the basic principle in section 2.1, and illustrate the algorithms constituting our framework in section 2.2. Section 2.3 covers issues related to range restrictions of current graphics hardware.

Our method achieves high-quality texture filtering by performing a convolution of an input texture with a "continuous" filter kernel, which in reality is stored in several texture maps of user-specified size. The filter kernel is sampled and stored into these textures in a pre-process. Each of the individual parts of the filter kernel that is stored in a separate texture map is called a *filter tile*, and the corresponding textures are referred to as *tile textures*. Since the filter representation is of relatively high resolution, i.e., contains a much higher number of samples than the width of the kernel, we call this representation a *tiled high-resolution filter*. The exact number of textures depends on properties of the filter and the algorithm used, but in the case of cubic filters it ranges from two to 64. At run time, arbitrary 1D, 2D, or 3D input textures can be convolved with the sampled filter kernel. In order to do so, the filter convolution sum is evaluated entirely in hardware, by exploiting multi-texturing, vertex shaders, pixel shaders, and multiple rendering passes. Basically, in each pass the input texture is point-sampled and multiplied with a single tile texture, which is replicated and scaled in such a way that an entire tile corresponds to a single texel of the input texture. Adding up the results of these multiplications performed in the corresponding passes yields the final result, which is equivalent to evaluating the filter convolution sum (equation 1).

An important property of the framework presented in this paper is that pre-processing of the input data to be filtered is not required for all but one variant of the presented approaches. However, we also propose algorithms operating on monochrome input data pre-interleaved into RGBA data, which can be faster than using unprocessed input data.

The filter convolution sum. Since we want to be able to deal with the completely general case of discrete convolution of an input texture with an arbitrary high-resolution filter kernel, we have to evaluate the well-known filter convolution sum [15], where $f[i]$ is the discrete input texture, $h(x)$ is the "continuous" (i.e., in practice still discrete, but sampled with high resolution) filter kernel, and m is half the filter width, which we show here for the one-dimensional case:

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

Note that, although not used in practice, we are able to deal with filter kernels of odd width, by simply extending them to the next even extent with zeros. The following two sections will show how the filter convolution sum can be evaluated in hardware extremely quickly.

2.1 Basics of high-res hardware convolution

This section illustrates the basic principle of evaluating equation 1 entirely in graphics hardware, by substituting $h(x)$ with a sampled high-resolution representation that is stored in several texture maps, and exploiting multi-texturing and multiple rendering passes for the actual computation. In this discussion, we are using an implementation-centric point of view, and review the basic idea only briefly. A more detailed discussion of the basic idea can be found in [1].

Filter kernel representation. The major property of high-resolution convolution is that any given continuous filter kernel is sampled at a user-specified resolution that is much higher than the width of the filter kernel. After sampling, we store the filter kernel in multiple texture maps, which we call tile textures. Specifically, each filter kernel subset of unit extent is sampled with the same resolution, and becomes exactly one texture map. That is, the sampled filter kernel is split up into filter tiles at integer locations, before it is converted to a collection of texture maps. In the case of a cubic kernel – which has width four – the kernel is therefore comprised of four individual 1D texture maps, i.e., for the ranges

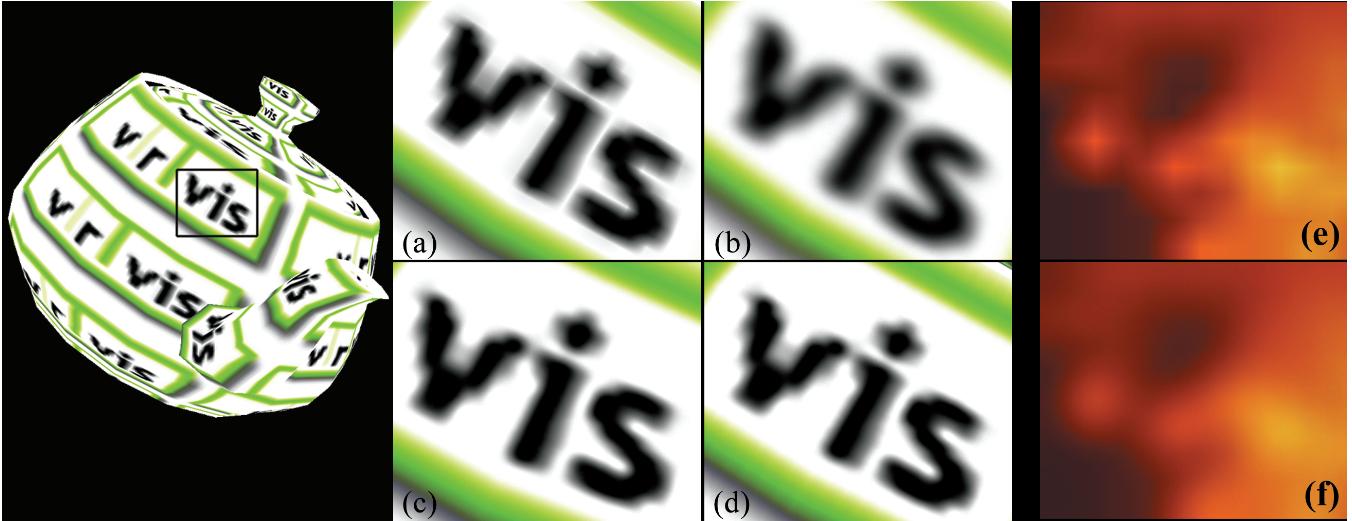

Figure 1: Left: filtering a 64x64 texture mapped several times onto a geometric object; although filtering and resampling is conceptually done in texture space, the algorithm resamples only at the locations actually corresponding to screen pixels, i.e., could be said to operate in image space; (a) bi-linear interpolation, (b) bi-cubic B-spline, (c) Catmull-Rom spline, (d) Kaiser-windowed sinc. Right: Procedural fire texture animation; (e) bi-linear interpolation, (f) bi-cubic B-spline.

$[-2,-1)$, $[-1,0)$, $[0,1)$, and $[1,2)$. Kernels of higher dimensionality than one are handled depending on whether they are separable or not. If a kernel is separable, lower-dimensional (usually 1D) components are stored, and multiplied on-the-fly at run time, preserving both texture memory, and texture fetch bandwidth. If it is not separable, it has to be sampled into several texture maps of according dimensionality, i.e., non-separable kernels are required to reside in either 2D or 3D texture maps. Thus, in the case of a bi-cubic kernel, sixteen 2D texture maps are needed, whereas a tri-cubic kernel requires 64 3D textures. However, if a filter kernel is symmetric, this property can also be exploited in order to reduce the number of kernel textures required. E.g., a symmetric bi-cubic kernel can be stored in three instead of sixteen textures, and a symmetric tri-cubic kernel can be stored in four instead of 64 textures. Fortunately, many important filter kernels – especially many filters used for function reconstruction – are both separable and symmetric. Thus, it is possible to attain both bi-cubic and tri-cubic reconstruction with only two 1D textures for storing the filter kernel itself. A sampling resolution of 64 or 128 samples per filter tile and dimension usually suffices.

Evaluation of the convolution sum. At run time, the filter convolution sum is evaluated over multiple rendering passes, in each pass simultaneously point-sampling the input texture, generating the needed filter weights, and multiplying the corresponding input data and weights. The number of rendering passes depends on the width of the filter kernel, the convolution algorithm employed (section 2.2), and the maximum number of texture units supported by the hardware. It may be as low as a single pass. Generation of filter weights ranges from simply sampling one of the tile textures, to compositing two or three values or vectors, retrieved from different filter tiles, in the pixel shader. Basically, both the input texture and from one to three tile textures need to be mapped to the output sample grid in screen space multiple times. This mapping is the same for the input texture and the filter tiles apart from scale, where an entire filter tile is mapped to a single input texel. Also, tile textures are automatically repeated, so that the same filter tile maps to every input texel. Perspective correction is not only used for input textures, but also for tile textures. Thus, all approaches presented in this paper are independent from the type of projection used. They work equally well for both orthogonal and perspective projections.

(a) FOR ALL output samples x_i DO
  FOR ALL contributing relative input locations r_j DO
    $g(x_i)$ += $f[\mathit{trunc}(x_i) + r_j] * h(\mathit{frac}(x_i) - r_j)$;

(b) FOR ALL contributing relative input locations r_j DO
  PAR ALL output samples x_i DO
    $g(x_i)$ += $\mathit{shift}_j(f)[\mathit{trunc}(x_i)] * h_j(\mathit{frac}(x_i))$;

Table 1: Evaluating Eq. 1 in the usual order (a), versus the one we are using (b). Each iteration of the outer loop in (b) basically corresponds to a single rendering pass; $r_j \in [-\lceil m \rceil + 1, \lceil m \rceil]$. *PAR* denotes a parallel *FOR* loop.

This basic principle amounts to evaluating the convolution sum in a different order than the one usually employed, i.e., in software convolution. Instead of calculating each output sample at a point x in its entirety, we instead distribute the contribution of a single relative input sample to all relevant output samples simultaneously. That is, equation 1 would usually be evaluated with code roughly equivalent to what is shown in table 1(a): looping over all output locations x_i one after the other, the corresponding output sample $g(x_i)$ is generated by adding up the contributions of the $2m$ contributing neighbor samples, whose locations are specified relative (r_j) to the output sample location. Instead of this, using high-resolution filter tiles and multiplying input samples by filter weights in the pixel shader amounts to what is shown in table 1(b): looping over all relative input sample locations contributing to each respective output sample location, a single rendering pass adds the corresponding contribution of each relative input sample to all output samples simultaneously. The shift operator shift_j denotes that the input texture is not actually indexed differently at each output sample location, but instead the entire input texture is shifted according to the current rendering pass, in order to retrieve a single *relative* input sample location for all output samples simultaneously. Sim-

ilarly, h_j denotes the filter tile corresponding to the current pass, instead of the entire filter kernel h.

2.2 Convolution algorithms

This section presents the actual algorithms for performing convolution with high-resolution filters in hardware. Each of the basic algorithms can be expanded to make use of more available texture units (and thus fewer passes), by simply adding the results of multiple basic blocks together in a single pass. Combining multiple logical rendering passes also helps to exploit higher internal calculation precision in order to reduce quantization artifacts. We illustrate the differences with pseudo code fragments showing the global setup (for all passes), setup that changes per pass, and the vertex and pixel shaders executed in each pass. For the actual implementations of the latter two, we use either the `NV_vertex_program` and `NV_register_combiners` (NVIDIA GeForce 3), or the `EXT_vertex_shader` and `ATI_fragment_shader` (ATI Radeon 8500) OpenGL extensions.

General filter kernels. In the general case, we assume that the filter kernel is used and stored in its entirety, i.e., potential separability or symmetry is not exploited. This implies that all tile textures have the dimensionality of the convolution itself (1D, 2D, or 3D). Naturally, this case has the highest demands on texture memory and texture fetch rate. Table 2 shows the pseudo code for the resulting algorithm, which we call *std-2x*. "std" meaning the standard algorithm (not separated, non-interleaved), and "2x" denoting the number of textures used in a single pass (in this case two). Several of these building blocks can be combined in a single rendering pass, depending on the number of texture units supported by the hardware, thus yielding the analogous extended algorithms *std-4x*, *std-6x*, and so on. The basic algorithm works identically for 1D, 2D, and 3D convolutions, respectively, apart from minor details such as the number of texture coordinates that need to be configured.

```
global setup:
    input texture:  as-is, all formats (mono, RGB, RGBA, etc.);
    filter tiles:  1D, or not-separated 2D or 3D;
per-pass setup:
    filter tiles:  select tile corresponding to pass;
    if kernel symmetric:  setup tile mirroring;
vertex shader:
    texcoord[ UNIT0 ].s{t{r}} = shift_j( texcoord[ UNIT0 ] );
    texcoord[ UNIT1 ].s{t{r}} = texcoord[ UNIT0 ] * tile_size;
pixel shader:
    reg0 = SampleTex( UNIT0 );
    reg1 = SampleTex( UNIT1 );
    out = Multiply( reg0, reg1 );
```

Table 2: Convolution with *std-2x* algorithm.

Symmetric filter kernels. Exploiting filter kernel symmetry can easily be done be reusing tiles with mirrored texture coordinates. Symmetry can be exploited both along axes and diagonals. Especially when the filter is not separable, it is important to exploit symmetry, in order to reduce texture memory consumption, which is especially high in the case of 3D convolutions. It is also important to note here that mirroring filter tiles is not necessarily as simple as swapping texture coordinates. The reason for this is that the mirroring is required to be pixel-exact. That is, if a specific pixel in screen space was covered by a specific weight in a filter tile in one pass, it must be covered by the exactly corresponding weight in the respective mirrored tile in another pass. While it might simply suffice to mirror texture coordinates by, e.g., negation in the vertex shader, on some hardware architectures it may be necessary to actually mirror tiles in the pixel shader. Although texture coordinate iteration can be exact enough for coordinates to be mirrored only at the vertices and still achieve consistent results in different passes, the only way to guarantee pixel-exactness is mirroring on a per-pixel basis. The algorithm outlined in table 2 contains an "if kernel symmetric" clause, which denotes where tile mirroring needs to be setup in order to exploit symmetric filter kernels.

Separable filter kernels. When the filter kernel is separable, the texture memory and texture fetch rate requirements can be reduced tremendously. Instead of using actual 2D or 3D tile textures for 2D or 3D convolutions, the higher-dimensional weights can be generated on-the-fly from either two or three one-dimensional tile textures in the pixel shader. This is easily possible by performing separable composition, i.e., multiplying corresponding weights retrieved from two – not necessarily different – lower-dimensional tiles. Alternatively, for 3D convolutions the needed filter weights can also be generated from one 1D texture and one 2D texture, e.g., in order to lower the number of texture units required. Table 3 shows the corresponding algorithms. The *sep-3x* algorithm can be used for 2D or 3D convolution. In the former, two-dimensional filter tiles are substituted by two one-dimensional tiles. In the latter, one 1D tile and one 2D tile are used instead of a single three-dimensional tile. Although in theory there would not be any difference between exploiting separability by generating filter weights on-the-fly, and not utilizing it by simply performing separable composition beforehand when building tile textures, in practice the results are slightly different. The reason for this is that in a pre-process the necessary multiplications can be carried out in floating-point precision, whereas in the pixel shader they have to be done at whatever precision the hardware offers there. Although rarely noticeable, exploiting separability or not is definitely a quality/performance trade-off. While we have also used numerical simulations for estimating the numerical difference, the best way to prefer one algorithm over the other is by visual comparison.

```
global setup:
    input texture:  as-is, all formats (mono, RGB, RGBA, etc.);
    filter tiles:  separable; only 1D (3x/4x), or 1D plus 2D (3x);
per-pass setup:
    filter tiles:  select tiles corresponding to pass;
    if kernel symmetric:  setup tiles mirroring;
vertex shader:
    texcoord[ UNIT0 ].s{t{r}} = shift_j( texcoord[ UNIT0 ] );
    texcoord[ UNIT1 ].s{t} = texcoord[ UNIT0 ] * tile_size;
    texcoord[ UNIT2 ].s = texcoord[ UNIT0 ] * tile_size;
    texcoord[ UNIT3 ].s = texcoord[ UNIT0 ] * tile_size;  // sep-4x
pixel shader:
    reg0 = SampleTex( UNIT0 );
    reg1 = SampleTex( UNIT1 );
    reg2 = SampleTex( UNIT2 );
    reg3 = SampleTex( UNIT3 );  // sep-4x
    reg2 = Multiply( reg2, reg3 );  // sep-4x
    reg1 = Multiply( reg1, reg2 );
    out = Multiply( reg0, reg1 );
```

Table 3: Convolution with *sep-3x* and *sep-4x* algorithms; only the latter contains the statements marked with *// sep-4x*.

Separable and symmetric filter kernels. The best possible combination of kernel properties is when a filter is both separable and symmetric, which fortunately many interesting filter kernels are – especially many of those one would like to use for function reconstruction purposes. The pseudo code in table 3 contains an "if kernel symmetric" clause, which denotes where kernel symmetry needs to be taken into account. Apart from this, the algorithm is

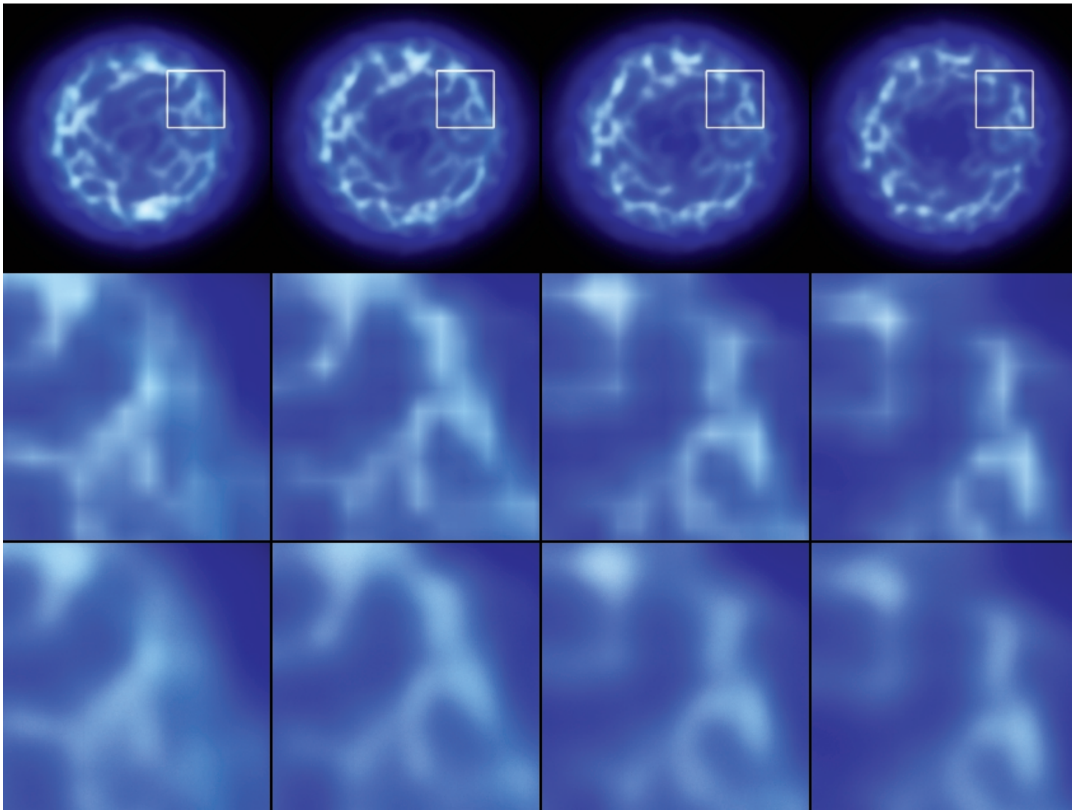

Figure 2: Four frames of a 2D particle animation from the space combat game Parsec [16], employing high-resolution convolution for high quality reconstruction; middle row: bi-linear interpolation; bottom row: bi-cubic B-spline.

identical to the separable-only case.

Pre-interleaved monochrome input. If the input data is single-valued and a certain limited amount of pre-processing is considered feasible, all of the algorithms outlined above can be combined with the following scheme that exploits the capability of graphics hardware to perform per-pixel dot products. The idea is to fold four passes into a single pass, by evaluating four terms of the convolution sum in a single operation. In order to do so, four input values have to be available simultaneously, which can be achieved by converting a monochrome input texture into an RGBA texture by interleaving it four times with itself, each time using a different texel offset. That is, at each position in the texture the value of three neighboring texels is available in addition to the original texel itself. If the tile textures are also interleaved accordingly, thus achieving correct correspondences of relative input samples and filter weights in all four channels, four terms of the convolution sum can then be evaluated concurrently by using a four-component dot product instead of a component-wise multiplication in the pixel shader. The *dot-2x* pseudo code in table 4 illustrates this as an extension of the *std-2x* algorithm of table 2. Although not shown here for brevity, the same approach can also be combined trivially with the separable algorithms outlined in table 3, thus yielding the *spd-3x*, *spd-4x*, *spd-6x*, etc. algorithms. Exploiting both separability and pre-interleaved input data is generally the fastest approach offered by our framework (see table 5). Note that the interleaved source texture is of exactly four times the size of the corresponding monochrome texture. Or, comparing with the size of an RGBA input, the size stays the same, but color information is lost.

Combination with MIP mapping. All of the algorithms presented up to now can be extended to cooperate with MIP mapping. The basic problem is that the actual resolution of the texture map has to be known in texels for two inherent parts of our framework. First, the input texture has to be shifted by one-texel offsets in order to distribute the corresponding input values to the convolution sum. Second, the tile textures have to be scaled in such a way that a single tile matches up with a single texel of the input texture. Theoretically, it would be possible to alleviate the latter problem by storing filter tiles repeatedly in the corresponding textures and building a MIP map pyramid for them, instead of letting the hardware automatically repeat tiles. However, this is entirely infeasible due to the texture resolutions that would be required. While both of these steps can be done in the vertex shader if the texture resolution is known, this is not possible anymore in the presence of MIP mapping, where the actual resolution of the input texture may vary from

```
global setup:
    input texture:  monochrome, but interleaved in RGBA;
    filter tiles:  1D, 2D, or 3D; analogously interleaved in RGBA;
per-pass setup:
    filter tiles:  select tile corresponding to pass;
    if kernel symmetric:  setup tiles mirroring;
vertex shader:
    texcoord[ UNIT0 ].s{t{r}} = shift_j( texcoord[ UNIT0 ] );
    texcoord[ UNIT1 ].s{t{r}} = texcoord[ UNIT0 ] * tile_size;
pixel shader:
    reg0 = SampleTex( UNIT0 );
    reg1 = SampleTex( UNIT1 );
    out = DotProduct4( reg0, reg1 );
```

Table 4: Convolution with *dot-2x* algorithm.

pixel to pixel. Thus, both steps have to be performed in the pixel shader instead. In order for this to work, the hardware is required to allow the output of per-pixel arithmetic operations to be used as texture coordinates in the pixel shader, which is already possible on the most recent generations of programmable graphics cards. Furthermore, the pixel shader must be able to determine which MIP map level the hardware is actually using for the pixel under consideration. Currently, this is only possible with a workaround, where in addition to the input texture a second MIP mapped texture is used, which only contains information about the MIP map levels themselves, instead of actual data used for rendering. In order to obviate this additional texture traffic, we propose adding an instruction to the instruction set of pixel shaders that allows determining the MIP map level corresponding to the current pixel.

The combination of MIP mapping and high-resolution convolution is currently implemented as follows. In addition to the textures needed by the basic algorithm, where the input texture now contains several MIP map levels, we also use a second MIP map containing information needed by the pixel shader in order to accommodate the changing texture resolution in the convolution algorithm. Depending on the resolution of the current MIP map level, two different kinds of information are needed. First, the texture coordinate offset from one input texel to the next, which usually is $1/$*texture_size*. Second, the scale factor required for mapping whole filter tiles to single input texels (*tile_size*). The first value is multiplied by a pass-specific offset, which is given in the unit of whole texels, and added to the texture coordinates of the input texture. The second value cannot easily be stored in a texture, since it is greater than one. Therefore, instead of storing it directly, we store a factor that determines how the texture coordinates for the largest MIP map level need to be scaled down in order to correspond to the current level: *tile_size*$/$*largest_level_size*. This value is then multiplied by the interpolated texture coordinates of the texture unit corresponding to the tile texture.

2.3 Filter kernel range considerations

Although signed texture formats have already become available, the fundamental problem that has to be overcome when using filter kernels that contain negative as well as positive values, is that the $[0, 1]$ range of current frame buffers must never be exceeded in either direction between rendering passes. Note that increased frame buffer range – at least to $[-1, 1]$ – would obviate many or all of the workarounds outlined in this section.

Using filter kernels containing negative weights. Many relevant filter kernels change sign only at the integers. For example, the zero-crossings of all useful interpolatory filters are integers. In this case, filter tiles are either entirely positive, or entirely negative, which can easily be handled by storing the absolute values of negative tiles, and subtracting from the frame buffer instead of adding to it. Second, even if single filter tiles contain weights of mixed sign, the problem can be solved by splitting up these tiles into two tiles. One tile for all positive values, where the others are set to zero, and one tile for the absolute values of all negative values, where once again the others are set to zero. Since the two split tiles have to be handled in separate passes, tiles that need to be split increase the overall number of rendering passes. In any case, care must be taken with respect to the order of passes, since subtracting from the frame buffer can only avoid clamping artifacts if sufficiently positive values have been written previously. Furthermore, depending on the actual function of the filter kernel, even entirely positive tiles may need to be split up in presence of other tiles containing negative values. This becomes necessary if the distribution of values is in such a way that intermediate results go above one when beginning with highly positive filter tiles, or below zero, when starting out with tiles where the function has already decreased to lower levels. The solution in such a case is to split up the highly positive tiles, and use one part of these before the negative ones, and the other part afterward. The actual splitting point that avoids clamping in all cases depends on the filter kernel and we determine it via a simple numerical simulation in a pre-process. For example, very high quality results with a bi-cubic Catmull-Rom spline can be achieved by splitting up the center tile, which increases the number of passes in the *std-2x* algorithm from sixteen to twenty.

Interpolation by pre-processing. Another possibility is to avoid negative values in the filter kernels at all. In order to do this, we modify the filter convolution sum (equation 1) slightly:

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - \lceil m \rceil + 1}^{\lfloor x \rfloor + \lceil m \rceil} c[i]h(x - i) \qquad (2)$$

The difference here is that we do not filter the actual data points but some coefficients $c[i]$, and require the resulting function to interpolate the original data. The coefficients depend on the actual filter used and can be calculated by matrix inversion [26]. We can now use a non-negative filter (e.g., cubic B-spline) and still get high-quality interpolation results [23]. The coefficients are calculated in a pre-processing step in software. As these coefficients usually exceed the range $[0, 1]$ we fit them into this range using simple scale and bias operations to avoid clamping artifacts. After filtering with the non-negative filter, where we render to a texture instead of the frame buffer, one additional rendering pass is required to restore the correct intensity values with an inverse scale and bias operation. Although the last pass is done in the pixel shader, the required render-to-texture operation is rather time-consuming.

3 Surface texture (2D) convolution

The framework presented in this paper can be used as a high quality, but still very fast, substitute for the most widely used hardware method for reconstruction of surface textures: using bi-cubic instead of bi-linear interpolation. We have used cubic Catmull-Rom splines, cubic B-splines, and windowed sinc filters with Kaiser and Blackman windows of width four for this purpose, although other filters can also be used. Table 5 shows frame rates we have been able to attain for 2D convolution.

Static textures. Figures 1 and 3 show a zoom-in of a 64x64 resolution texture mapped onto a 3D object, which illustrates that especially for textures of low resolution a higher order filter kernel can make a tremendous difference with respect to reconstruction quality. A case where textures of even much lower relative resolution are used frequently, are light maps for real-time display of radiosity lighting solutions, e.g., in computer games. Also, in the case of dynamic lighting via (possibly projective) texture maps [21], the resolution relative to the base texture is often very low, making linear interpolation artifacts strongly visible. Our framework filters in texture space and is therefore independent from the kind of projection used. Textures can be mapped to any underlying geometry. Furthermore, high-quality reconstruction of 2D textures can be used for high-quality volume rendering, where the indiviual slices constituting the volume are filtered via high-resolution convolution (figure 5). Since our framework employs multiple rendering passes, transparent polygons cannot be handled directly. As in all multi-pass algorithms, the computation split up into these passes must not interfere with the blending operation used for transparency. In our case, this means that transparent polygons must first be filtered in an off-screen buffer, and the already filtered result must be used when blending into the frame buffer.

Pre-rendered texture animations. In texture animations, such as the one shown in figure 2, the artifacts of linear interpolation are even more pronounced than in the case of static textures. The

underlying grid appears as "static" layer beneath the texture itself. These artifacts are successfully removed to a sufficient extent by bi-cubic interpolation, which can be seen very well in the accompanying video.

Procedural texture animations. When texture animations are generated procedurally, lower texture resolution speeds up the generation process, because fewer samples need to be generated. If the texture is generated using the CPU, this also decreases download time to the graphics hardware. Figure 1 also shows a comparison of two low-resolution procedural textures generated on the GPU itself, and subsequently filtered on-the-fly by the hardware.

4 Solid texture (3D) convolution

All the applications presented in the previous section also arise in three dimensions. Although in 3D an increased number of rendering passes and – in the case of a non-separable kernel – more texture memory is needed than in 2D, it is still possible to filter solid or volumetric textures at real-time frame rates (see table 5).

Static textures. Due to their highly increased memory consumption, solid textures are usually of rather low resolution. Figure 4 shows an example of an object mapped with a solid texture, where tri-cubic convolution significantly enhances image quality.

Animated textures. Although several frames of 3D textures usually consume too much texture memory for feasible use, such textures can be generated procedurally on-demand. Especially when the procedural texture is also generated in hardware, this is feasible for low resolutions. Using higher-order filtering again helps to overcome the low-resolution sampling.

5 Summary and conclusions

In this paper we present a framework for performing convolution with basically arbitrary high-resolution filter kernels in graphics hardware extremely quickly. In order to exploit different filter properties and speed/quality trade-offs, we have described several different algorithms, which have been implemented on both the NVIDIA GeForce 3 and the ATI Radeon 8500. They make use of state-of-the-art hardware features such as vertex shaders, pixel shaders, and multi-texturing, and are able to exploit almost any number of texture units. Although we are focusing on reconstruction filters, the presented approaches can be combined with per-pixel MIP mapping as approximation to prefiltering, which is crucial for high-quality reconstruction to become a real alternative to linear interpolation.

We have shown a number of applications for the especially interesting cases of cubic filters, and we propose cubic convolution as feasible real-time substitute for linear interpolation, which is able to avoid many of the artifacts associated with the latter. In general, high-quality reconstruction is especially suited to animations, where the integer lattice is often tremendously visible when using linear interpolation. The texture memory footprint of high-resolution filters can be very low. Cubic convolution from one to three dimensions can be done with as few as two 1D textures containing only 64 samples each. Also, the number of rendering passes required is suited to current hardware. Bi-cubic convolution is possible using from two to sixteen passes, tri-cubic convolution from eight to 64, depending on the algorithm employed.

Graphics hardware architectures are currently in the process of becoming much more programmable in order to serve as back-end for high-level real-time shading languages [18, 20], which can now be extended with *programmable convolution filters*. The migration towards real-time shading languages also necessitates higher-precision frame buffers, which will increase the variety of filter kernels that can be used by our framework in practice, especially with regard to filter width.

Acknowledgments

This work was done as part of the basic research on visualization at the VRVis Research Center in Vienna, which is funded by an Austrian research program called K plus.

References

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

[2] P. Haeberli and M. Segal. Texture mapping as a fundamental drawing primitive. In *Proc. of Fourth Eurographics Workshop on Rendering*, pages 259–266, 1993.

[3] P. Heckbert. Filtering by repeated integration. In *Proc. of SIGGRAPH '86*, pages 317–321, 1986.

[4] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley Department of Electrical Engineering and Computer Science, 1989.

[5] W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. In *Proc. of SIGGRAPH '99*, pages 171–178, 1999.

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

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

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

[9] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proc. of IEEE Visualization '94*, pages 100–107, 1994.

[10] J. McCormack, R. Perry, K. Farkas, and N. Jouppi. Feline: Fast elliptical lines for anisotropic texture mapping. In *Proc. of SIGGRAPH '99*, pages 243–250, 1999.

[11] T. McReynolds, D. Blythe, B. Grantham, and S. Nelson. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 2000 course notes*, 2000.

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

[13] T. Möller, R. Machiraju, K. Müller, 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.

[14] M. Olano, S. Mukherjee, and A. Dorbie. Vertex-based anisotropic texturing. In *Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop 2001*, 2001.

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

[fps]	# pixels	GF3	ATI	GF3	ATI	GF3	ATI	GF3	ATI	GF3	ATI	GF3	ATI	GF3	ATI	GF3	ATI
2D	60k	55	24	105	46	190	90	275	167	55	24	*n/a*	46	190	90	*n/a*	167
	180k	55	24	57	34	125	71	108	78	55	24	*n/a*	46	188	71	*n/a*	83
	260k	50	17	36	19	80	45	60	45	46	24	*n/a*	46	150	55	*n/a*	62
	900k	25	9	15	9	28	20	19	19	22	23	*n/a*	35	70	26	*n/a*	27
	1200k	20	8	18	9	28	16	13	15	15	16	*n/a*	24	55	55	*n/a*	32
	alg (#pass)	**std-2x** (16)		**std-4x** (8)		**dot-2x** (4)		**dot-4x** (2)		**sep-3x** (16)		**sep-6x** (8)		**spd-3x** (4)		**spd-6x** (2)	
3D	60k	19	23	21	26	64	71	66	71	21	59			76	90		
	180k	6.5	4.2	6.8	4.5	20	15	20	16	14	30			50	30		
	260k	4.2	2.8	4.5	2.3	11	8	11	9	8.7	18			34	40		
	alg (#pass)	**std-2x** (64)		**std-4x** (32)		**dot-2x** (16)		**dot-4x** (8)		**sep-4x** (64)		*sep-8x*		**spd-4x** (16)		*spd-8x*	

Table 5: Frame rates for convolution with hi-res filter of width four on NVIDIA GeForce 3 (GF3), and ATI Radeon 8500 (ATI), respectively; an object with about 500 visible triangles per view was texture-mapped; for 2D convolution, a 64x64 texture was used several times; for 3D convolution, an object-encompassing 128^3 solid texture was used; the approximate number of pixels drawn in each case is also shown. The corresponding objects and textures are shown in figures 3 and 4, respectively.

[16] Parsec - there is no safe distance. http://www.parsec.org/.

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

[18] M. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proc. of SIGGRAPH 2000*, pages 425–432, 2000.

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

[20] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proc. of SIGGRAPH 2001*, pages 159–170, 2001.

[21] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *Proc. of SIGGRAPH '92*, pages 249–252, 1992.

[22] T. Theußl, H. Hauser, and E. Gröller. Mastering windows: Improving reconstruction. In *Proc. of IEEE Symposium on Volume Visualization*, pages 101–108, 2000.

[23] P. Thévenaz, T. Blu, and M. Unser. Interpolation revisited. *IEEE Transactions on Medical imaging*, 19(7), 2000.

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

[25] L. Williams. Pyramidal parametrics. In *Proceedings of SIGGRAPH '83*, pages 1–11, 1983.

[26] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1990. IEEE Computer Society Press Monograph.

[27] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proc. of SIGGRAPH 2001*, pages 371–378, 2001.

Figure 3: Filtering a 64x64 texture mapped several times onto a geometric object; from left to right: bi-linear interpolation, Catmull-Rom spline, Kaiser-windowed sinc, cubic B-spline. The corresponding frame rates are shown in table 5.

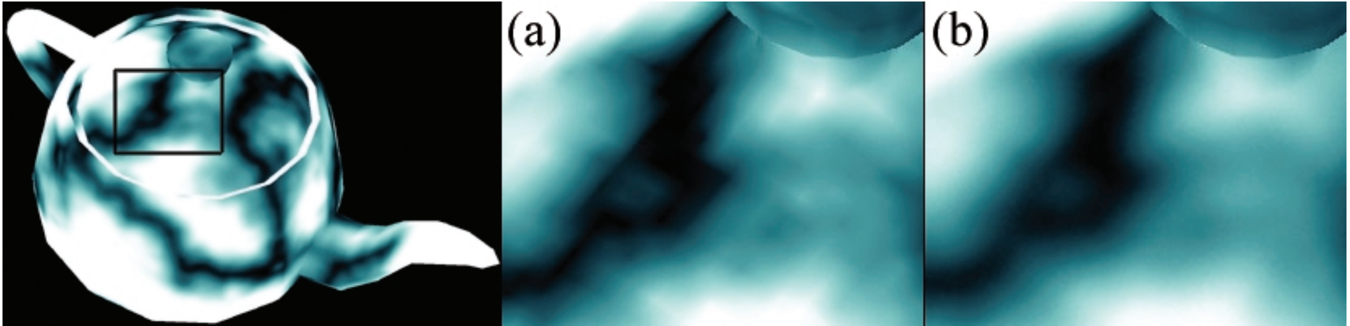

Figure 4: Filtering a 128^3 solid texture encompassing a geometric object; the two reconstruction filters used in these images are tri-linear interpolation (a), and a tri-cubic B-spline (b). The corresponding frame rates are shown in table 5.

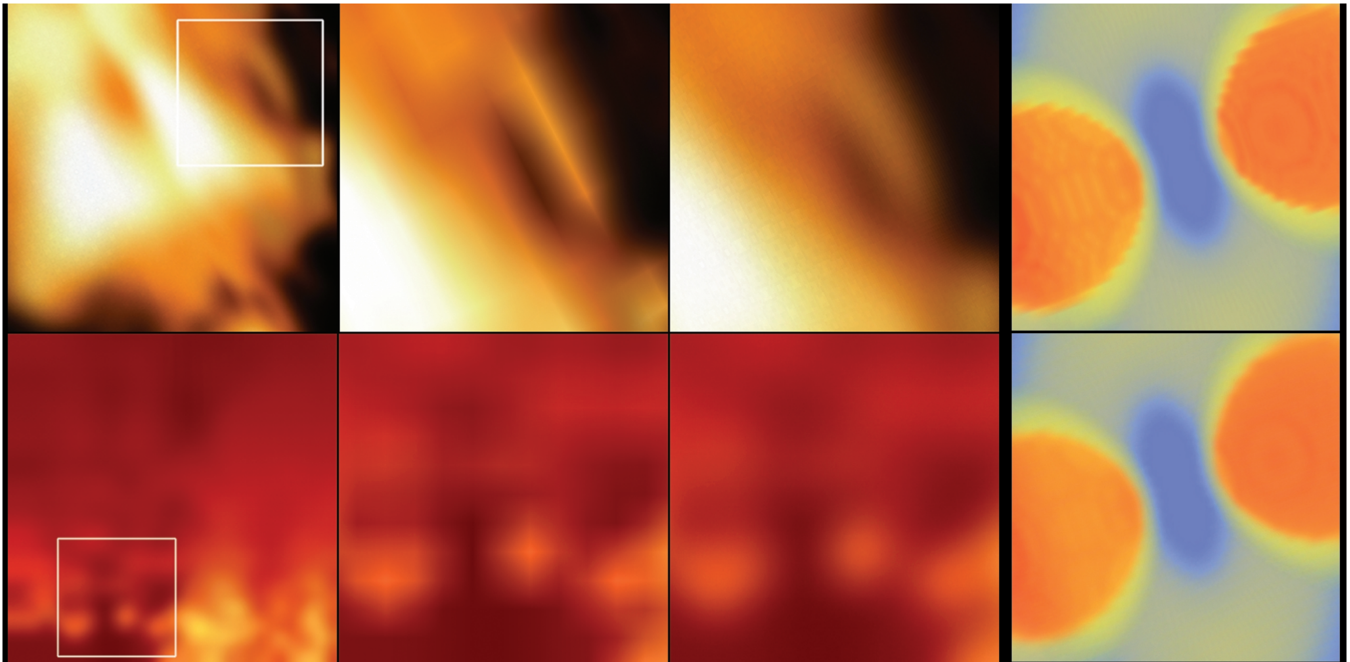

Figure 5: Different examples of using high-resolution convolution for filtering 2D textures with high quality. The three images on the upper left show a frame from a pre-rendered procedural fire animation with the magnified region filtered by the hardware-native bi-linear interpolation and a high-resolution bi-cubic B-spline, respectively. The three images on the lower left show a frame from a procedural fire animation that is generated on-the-fly on the GPU (the graphics hardware) itself. Once again, the magnified region has been filtered by using the bi-linear interpolation and a high-resolution bi-cubic B-spline, respectively. The two images on the right-hand side show a volume-rendered image of a hydrogen atom stored in a 64^3 volume and rendered by compositing 64 2D-texture mapped slices on top of each other. On the top, these slices have been individually filtered by bi-linear interpolation, whereas on the bottom a high-resolution bi-cubic B-spline has been used for this purpose. In this case, the artifacts removed by higher-order interpolation are especially visible.