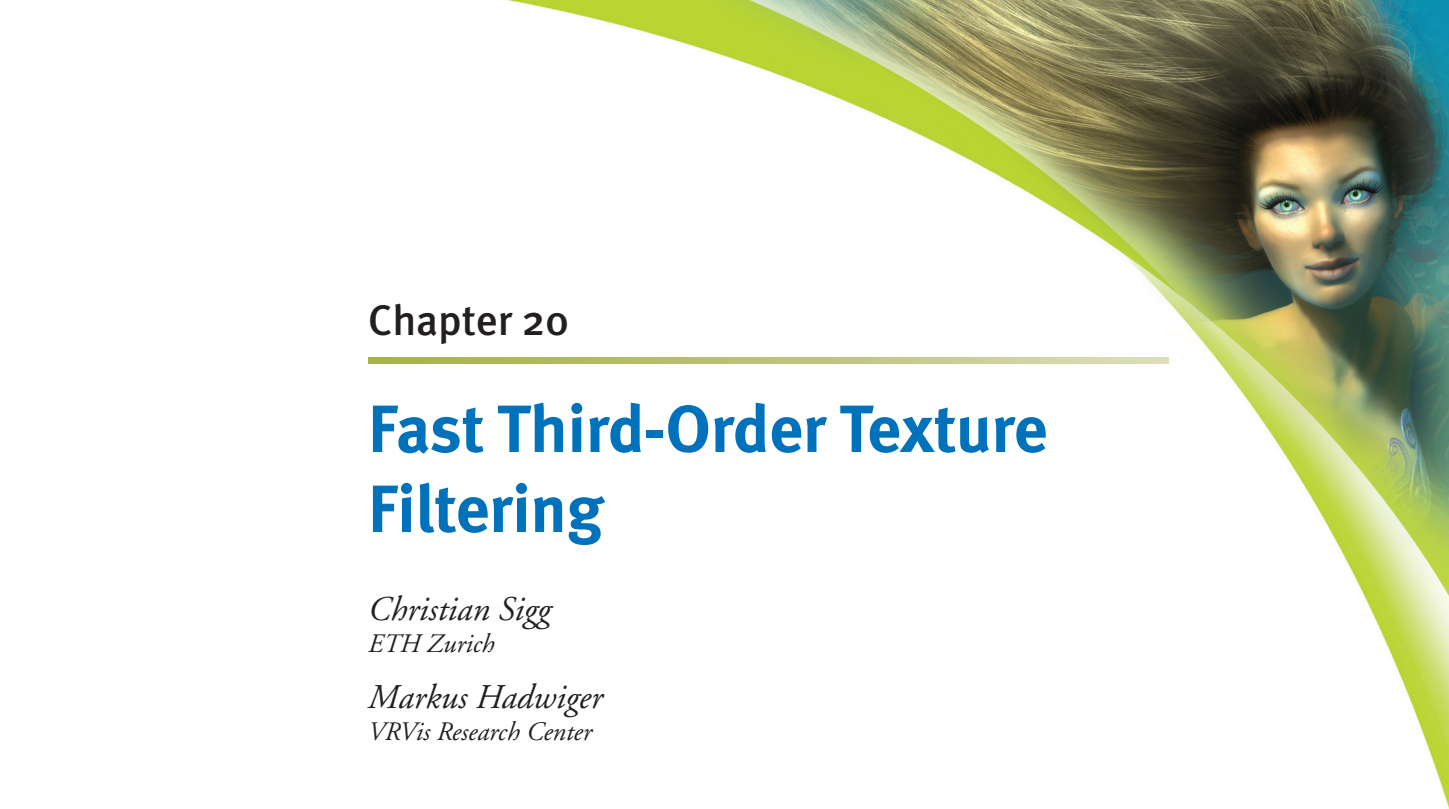


Chapter 20

Fast Third-Order Texture Filtering

Christian Sigg
ETH Zurich

Markus Hadwiger
VRVis Research Center

Current programmable graphics hardware makes it possible to implement general convolution filters in fragment shaders for high-quality texture filtering, such as cubic filters (Bjorke 2004). However, several shortcomings are usually associated with these approaches: the need to perform many texture lookups and the inability to antialias lookups with mipmaps, in particular. We address these issues in this chapter for filtering with a cubic B-spline kernel and its first and second derivatives in one, two, and three dimensions.

The major performance bottleneck of higher-order filters is the large number of input texture samples that are required, which are usually obtained via repeated nearest-neighbor sampling of the input texture. To reduce the number of input samples, we perform cubic filtering building on linear texture fetches, which reduces the number of texture accesses considerably, especially for 2D and 3D filtering. Specifically, we are able to evaluate a tricubic filter with 64 summands using just eight trilinear texture fetches.

Approaches that perform custom filtering in the fragment shader depend on knowledge of the input texture resolution, which usually prevents correct filtering of mipmapped textures. We describe a general approach for adapting a higher-order filtering scheme to mipmapped textures.

Often, high-quality derivative reconstruction is required in addition to value reconstruction, for example, in volume rendering. We extend our basic filtering method to

reconstruction of continuous first-order and second-order derivatives. A powerful application of these filters is on-the-fly computation of implicit surface curvature with tricubic B-splines, which have been applied to offline high-quality volume rendering, including nonphotorealistic styles (Kindlmann et al. 2003).

20.1 Higher-Order Filtering

Both OpenGL and Direct3D provide two different types of texture filtering: nearest-neighbor sampling and linear filtering, corresponding to zeroth and first-order filter schemes. Both types are natively supported by all GPUs. However, higher-order filtering modes often lead to superior image quality. Moreover, higher-order schemes are necessary to compute continuous derivatives of texture data.

We show how to implement efficient third-order texture filtering on current programmable graphics hardware. The following discussion primarily considers the one-dimensional case, but it extends directly to higher dimensions.

To reconstruct a texture with a cubic filter at a texture coordinate x, as shown in Figure 20-1a, we have to evaluate the convolution sum

$$w_0(x) \times f_{i-1} + w_1(x) \times f_i + w_2(x) \times f_{i+1} + w_3(x) \times f_{i+2} \quad (1)$$

of four weighted neighboring texels f_i. The weights $w_i(x)$ depend on the filter kernel used. Although there are many types of filters, we restrict ourselves to B-spline filters in this chapter. If the corresponding smoothing of the data is not desired, the method can also be adapted to interpolating filters such as Catmull-Rom splines.

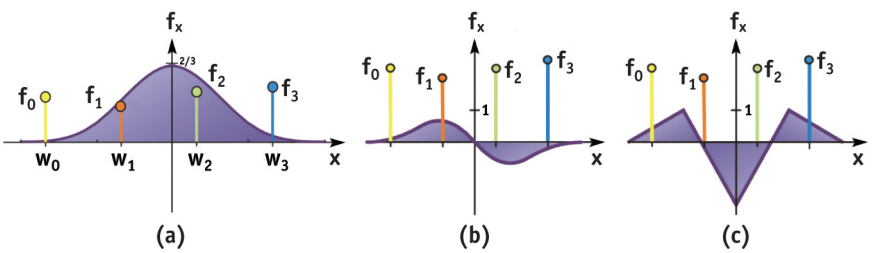


Figure 20-1. The Cubic B-Spline and Its Derivatives
(a) Convolution of input samples f_i with filter weights $w_i(x)$. First-order (b) and second-order (c) derivatives of the cubic B-spline filter for direct reconstruction of derivatives via convolution. (For purposes of illustration, the scale of the vertical axes varies.)

Note that cubic B-spline filtering is a natural extension of standard nearest-neighbor sampling and linear filtering, which are zeroth and first-degree B-spline filters. The degree of the filter is directly connected to the smoothness of the filtered data. Smooth data becomes especially important when we want to compute derivatives. For volume rendering, where derivatives are needed for shading, it has become common practice to store precomputed gradients along with the data. Although this leads to a continuous approximation of first-order derivatives, it uses four times more texture memory, which is often constrained in volume-rendering applications. Moreover, this approach becomes impractical for second-order derivatives because of the large storage overhead. On the other hand, on-the-fly cubic B-spline filtering yields continuous first-order and second-order derivatives without any storage overhead.

20.2 Fast Recursive Cubic Convolution

We now present an optimized evaluation of the convolution sum that has been tuned for the fundamental performance characteristics of graphics hardware, where linear texture filtering is evaluated using fast special-purpose units. Hence, a single linear texture fetch is much faster than two nearest-neighbor fetches, although both operations access the same number of texel values. When evaluating the convolution sum, we would like to benefit from this extra performance.

The key idea is to rewrite Equation 1 as a sum of weighted linear interpolations between every other pair of function samples. These linear interpolations can then be carried out using linear texture filtering, which computes convex combinations denoted in the following as

$$f_x = (1 - \alpha) \times f_i + \alpha \times f_{i+1}, \tag{2}$$

where $i = \lfloor x \rfloor$ is the integer part and $\alpha = x - i$ is the fractional part of x. Building on such a convex combination, we can rewrite a general linear combination $a \times f_i + b \times f_{i+1}$ with general a and b as

$$(a + b) \times f_{(i+b)/(a+b)} \tag{3}$$

as long as the convex combination property $0 \leq b/(a + b) \leq 1$ is fulfilled. Thus, rather than perform two texture lookups at f_i and f_{i+1} and a linear interpolation, we can do a single lookup at $i + b/(a + b)$ and just multiply by $(a + b)$.

The combination property is exactly the case when a and b have the same sign and are not both zero. The weights of Equation 1 with a cubic B-spline do meet this property, and therefore we can rewrite the entire convolution sum:

$$w_0(x) \times f_{i-1} + w_1(x) \times f_i + w_2(x) \times f_{i+1} + w_3(x) \times f_{i+2} = g_0(x) \times f_{x-h_0(x)} + g_1(x) \times f_{x+h_1(x)}, \tag{4}$$

introducing new functions $g_0(x)$, $g_1(x)$, $h_0(x)$, and $h_1(x)$ as follows:

$$\begin{aligned} g_0(x) &= w_0(x) + w_1(x) \quad h_0(x) = 1 - \frac{w_1(x)}{w_0(x) + w_1(x)} + x \\ g_1(x) &= w_2(x) + w_3(x) \quad h_1(x) = 1 + \frac{w_3(x)}{w_2(x) + w_3(x)} - x \end{aligned} \tag{5}$$

Using this scheme, the 1D convolution sum can be evaluated using two linear texture fetches plus one linear interpolation in the fragment program, which is faster than a straightforward implementation using four nearest-neighbor fetches. But most important, this scheme works especially well in higher dimensions; and for filtering in two and three dimensions, the number of texture fetches is reduced considerably, leading to much higher performance.

The filter weights $w_i(x)$ for cubic B-splines are periodic in the interval $x \in [0, 1]$: $w_i(x) = w_i(\alpha)$, where $\alpha = x - \lfloor x \rfloor$ is the fractional part of x. Specifically,

$$\begin{aligned} w_0(\alpha) &= \frac{1}{6}\left(-\alpha^3 + 3\alpha^2 - 3\alpha + 1\right) \quad w_1(\alpha) = \frac{1}{6}\left(3\alpha^3 - 6\alpha^2 + 4\right) \\ w_2(\alpha) &= \frac{1}{6}\left(-3\alpha^3 + 3\alpha^2 + 3\alpha + 1\right) \quad w_3(\alpha) = \frac{1}{6}\alpha^3 \end{aligned} \tag{6}$$

As a result, the functions $g_i(x)$ and $h_i(x)$ are also periodic in the interval $x \in [0, 1]$ and can therefore be stored in a 1D lookup texture.

We now discuss some implementation details, which include (1) transforming texture coordinates between lookup and color texture and (2) computing the weighted sum of the texture fetch results. The Cg code of the fragment program for one-dimensional cubic filtering is shown in Listing 20-1. The schematic is shown in Figure 20-2.

Listing 20-1. Cubic B-Spline Filtering of a One-Dimensional Texture

```
float4 bspline_1d_fp( float coord_source : TEXCOORD0,
  uniform sampler1D tex_source,  // source texture
  uniform sampler1D tex_hg,      // filter offsets and weights
  uniform float e_x,             // source texel size
  uniform float size_source      // source texture size
) : COLOR
{
  // calc filter texture coordinates where [0,1] is a single texel
  // (can be done in vertex program instead)
  float coord_hg = coord_source * size_source - 0.5f;

  // fetch offsets and weights from filter texture
  float3 hg_x = tex1D( tex_hg, coord_hg ).xyz;

  // determine linear sampling coordinates
  float coord_source1 = coord_source + hg_x.x * e_x;
  float coord_source0 = coord_source - hg_x.y * e_x;

  // fetch two linearly interpolated inputs
  float4 tex_source0 = tex1D( tex_source, coord_source0 );
  float4 tex_source1 = tex1D( tex_source, coord_source1 );
```

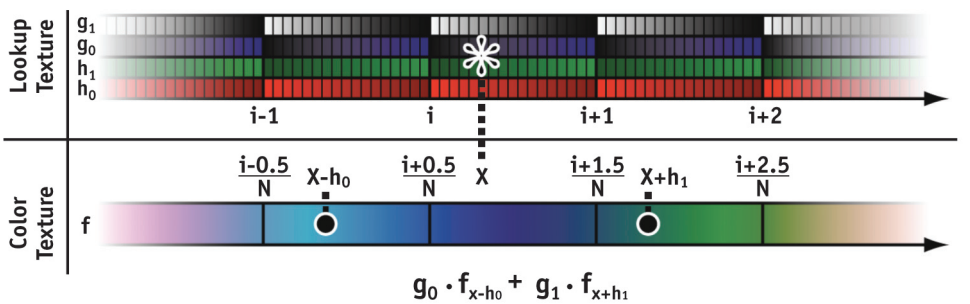


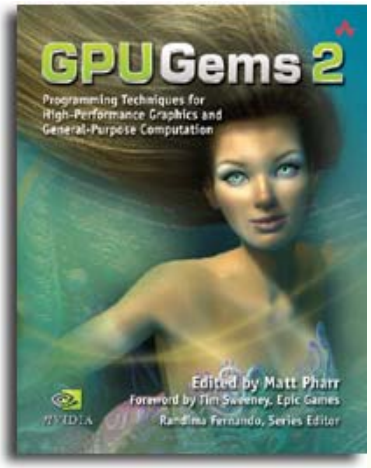
Figure 20-2. Cubic Filtering of a One-Dimensional Texture

To reconstruct a color texture of size N, we first perform a linear transformation of the reconstruction position x (∗). This gives us the texture coordinates for reading offsets $h_i(x)$ and weights $g_i(x)$ from a lookup texture. Second, two linear texture fetches of the color texture are carried out at the offset positions (●). Finally, the output color is computed by a linear combination of the fetched colors using the weights $g_i(x)$.

The full chapter appears in *GPU Gems 2*:

GPU Gems 2

Programming Techniques for High-Performance Graphics and General-Purpose Computation



- **880 full-color pages, 330 figures**
- **Hard cover**
- **$59.99**
- **Available at GDC 2005 (March 7, 2005)**
- **Experts from universities and industry**

Graphics Programming

- **Geometric Complexity**
- **Shading, Lighting, and Shadows**
- **High-Quality Rendering**

GPGPU Programming

- **General Purpose Computation on GPUs: A Primer**
- **Image-Oriented Computing**
- **Simulation and Numerical Algorithms**

For more information, please visit:

http://developer.nvidia.com/object/gpu_gems_2_home.html