Advance Access publication on 5 December 2010
doi:10.1093/comjnl/bxq086

GPU Prefilter for Accurate Cubic B-spline Interpolation

DANIEL RUIJTERS[1,*] AND PHILIPPE THÉVENAZ[2]

[1]*iXR Innovation, Philips Healthcare, Best, the Netherlands*
[2]*École polytechnique fédérale de Lausanne, Lausanne, Switzerland*
[*]*Corresponding author: danny.ruijters@philips.com*

Achieving accurate interpolation is an important requirement for many signal-processing applications. While nearest-neighbor and linear interpolation methods are popular due to their native GPU support, they unfortunately result in severe undesirable artifacts. Better interpolation methods are known but lack a native GPU support. Yet, a particularly attractive one is prefiltered cubic-spline interpolation. The signal it reconstructs from discrete samples has a much higher fidelity to the original data than what is achievable with nearest-neighbor and linear interpolation. At the same time, its computational load is moderate, provided a sequence of two operations is applied: first, prefilter the samples, and only then reconstruct the signal with the help of a B-spline basis. It has already been established in the literature that the reconstruction step can be implemented efficiently on a GPU. This article focuses on an efficient GPU implementation of the prefilter, on how to apply it to multidimensional samples (e.g. RGB color images), and on its performance aspects.

Keywords: cubic interpolation; B-spline; GPU acceleration

Received 12 May 2010; revised 5 October 2010
Handing editor: Hong Li

1. INTRODUCTION

Digital signals commonly consist of data samples on a discrete uniform (regular) grid. This is not only the case for images, but also for a broad range of other types of signals (e.g. audio [1]). The dimensionality of the grid typically depends on the type of the signal (e.g. 1D: audio, 2D: still images, 3D: video, volumetric data). The samples themselves can also be multidimensional, for instance when dealing with stereo audio or color images.

In many applications, such as signal processing [2], visualization [3, 4], image registration [5, 6] and scientific simulations, it is necessary to access signal values in between the sample locations, which calls for interpolation. This, in itself, is not a real hurdle since any bandwidth limited signal can be reconstructed perfectly by using the sinc function as the reconstruction basis. However, the slow decay of sinc makes this approach utterly impractical. Instead, nearest-neighbor and linear interpolation methods are favored because they are computationally much less expensive, and they are supported natively by the GPU.

Sigg and Hadwiger [7] have reported that using a cubic B-spline as the reconstruction basis can also be performed very efficiently by the GPU. Their method lacks some of the imperfections that are associated with nearest-neighbor (block artifacts) and linear interpolation (star-shaped artifacts), but also introduces a smoothing or blurring of the signal. The smoothing is caused by the fact that the cubic B-spline basis is a non-negative function. In addition to this smoothing, a different issue arises because the cubic B-spline is not itself interpolating, in the sense that the sequence $\{\ldots, 0, 0, \frac{1}{6}, \frac{2}{3}, \frac{1}{6}, 0, 0\ldots\}$ resulting from sampling a cubic B-spline at the integers differs from the unit sample sequence $\{\ldots, 0, 0, 0, 1, 0, 0, 0, \ldots\}$. Therefore, when used to directly reconstruct a signal, as was done in [7], the reconstructed function does not necessarily pass through the original sample points. This apparent drawback is shared by all B-splines of second and higher degree. This is particularly of importance when the filter is part of a processing chain.

An effective solution to the smoothing property has been formulated by Thévenaz *et al.* [8], and consists of applying the correct prefilter to the signal samples. In this paper, a GPU-accelerated version of this prefilter is described, implementation considerations are discussed and the resulting performance is evaluated. The acceleration is achieved by exploiting the massive parallelism available in modern graphics

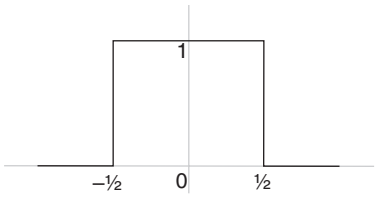

FIGURE 1. The box function (the B-spline basis of degree 0).

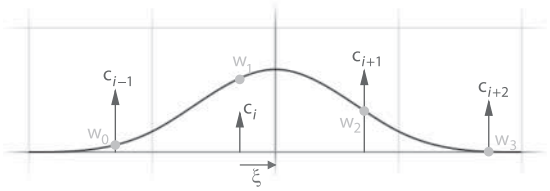

FIGURE 2. Cubic B-spline interpolation. The image coefficients c are multiplied by the weights $w_n(\xi)$. The weights are determined by the fractional amount $\xi = x - i$ of the present coordinate and by the B-spline basis function β^3. In this figure, the index i is the integer part of the coordinate.

hardware [9]. The corresponding source code is available for download [10].

This paper is organized as follows. Sections 2 and 3 provide the well-established theoretical background of B-spline interpolation that is needed to understand the GPU code. The implementation details and considerations are described in Section 4. The resulting performance and precision characteristics are discussed in Section 5.

2. B-SPLINE FILTERING

Uniform spline-based interpolation was introduced by Schoenberg [11] and has been described exhaustively by Thévenaz *et al.* and Unser [8, 12]. The foundation for B-spline functions of any non-negative integer degree is given by the B-spline basis of degree 0 (the box function; see Fig. 1) defined by

$$\beta^0(x) = \frac{1}{2}\left(\operatorname{sgn}\left(x + \frac{1}{2}\right) - \operatorname{sgn}\left(x - \frac{1}{2}\right)\right). \tag{1}$$

All other B-spline bases of higher integer degree n can be obtained by the recursive continuous convolution of the box function with the B-spline basis of degree $(n-1)$:

$$\beta^n(x) = \left(\beta^{n-1} * \beta^0\right)(x). \tag{2}$$

Nearest-neighbor and linear interpolation, which are popular because of their native GPU support, can be regarded as B-spline filtering of the 0th and 1st degree, respectively. It is straightforward to obtain explicit expressions of B-splines of any degree from (1) and (2); in particular, the cubic B-spline β^3 of $x \in \mathbb{R}$ can be written as

$$\beta^3(x) = \begin{cases} 0, & 2 \le |x|, \\ \frac{1}{6} \cdot (2 - |x|)^3, & 1 \le |x| < 2, \\ \frac{2}{3} - \frac{1}{2}|x|^2 \cdot (2 - |x|), & |x| < 1. \end{cases} \tag{3}$$

Given an appropriate sequence of coefficients c has been derived from the available sequence f of the samples of a signal, its spline-based reconstruction at a given position x can be written as

$$s(x) = \sum_{k' \in \mathbb{Z}} c[k']\, \beta^n(x - k'). \tag{4}$$

In other words, the value $s(x)$ reconstructed at a given position x is the sum of integer-shifted and weighted, centered B-spline bases β^n of degree n. The weights are provided by the coefficients c which are located on a uniform grid and reflect the contribution of the original samples f. We illustrate this process in Fig. 2 for the cubic B-spline. Since B-splines have a finite support, the number of coefficients $c[k']$ that play a role in the interpolation at position x is finite too. It turns out that, in the 1D case, this number is one more than the degree of the spline.

3. PREFILTER

A naive approach to reconstruct a continuous signal with (4) would be to enforce that the reconstruction coefficients c take the values of the original sequence f of samples. The corresponding reconstructed function, however, would not necessarily pass through the samples. Rather, it would present a smoothened version that would only approximate them, which is generally not desirable. Fortunately, as shown in [8], this can be overcome by assigning the appropriate prefiltered version of the samples to c. The objective of the prefilter is to obtain a sequence of coefficients that yield a truly interpolating function s that passes through the original samples f. Thus, the values for $c[k]$ should be collectively chosen such that the following equation is fulfilled for all integers $k \in \mathbb{Z}$:

$$f[k] = s(k) = \sum_{k' \in \mathbb{Z}} c[k']\, \beta^n(k - k'). \tag{5}$$

This version of (4) has the general flavor of a convolution, but is neither a discrete convolution nor a continuous one because it mixes the discrete sequence $c[\cdot]$ with the continuously defined function $\beta^n(\cdot)$. However, it must be realized that the argument of β^n is always discrete in the special case (5). Therefore, it is useful to define the sampled version of the continuously defined signal β^n as the discrete sequence b_n, with

$$\forall k \in \mathbb{Z}: \; b_n[k] = \beta^n(x)\big|_{x=k}. \tag{6}$$

Only then we can safely rewrite (5) as the discrete convolution:

$$f[k] = (c * b_n)[k]. \tag{7}$$

In the discrete Fourier space [13], with $z = e^{i\omega}$, this can be written as

$$F(z) = C(z)\, B_n(z). \tag{8}$$

This leads to the conclusion that c can be obtained by a convolution of the sequence f of samples with the inverse of the B-spline function, which can be expressed in the discrete Fourier space as

$$c[k] = \left(f * b_n^{-1}\right)[k] \;\overset{z}{\longleftrightarrow}\; C(z) = F(z)\, B_n^{-1}(z). \tag{9}$$

The interpolated continuous signal s does not only pass through the original discrete samples, but also delivers a function that is maximally differentiable. It is worth mentioning that this approach is equivalent to sinc interpolation for the infinite-degree B-spline [12]. In the case of the 3rd degree (cubic) B-spline, B_3 is given by

$$B_3(z) = \frac{1}{6}\left(z^{-1} + 4 + z\right), \tag{10}$$

while its inverse is

$$\begin{aligned} B_3^{-1}(z) &= \frac{6}{z^{-1} + 4 + z} \\ &= \frac{\lambda}{1 - z_p\, z^{-1}} \frac{-z_p}{1 - z_p\, z}, \end{aligned} \tag{11}$$

with $\lambda = 6$ and $z_p = \sqrt{3} - 2$. The sequence f of samples $f[k]$ needs to be convolved with this inverse filter.

If you are familiar with the z-transform, the first term $\lambda/(1 - z_p\, z^{-1})$ can be easily rewritten as a recursive filter. The second term, however, seems to be more cumbersome, since it contains an element z, which is looking into the future. Fortunately, we are dealing with images—as opposed to time signals. Thus, the whole sequence f is already known to us, and there is no problem in traversing the data backward starting from the end. The filters (illustrated in Fig. 3) can therefore be implemented as

$$\begin{aligned} &k \in \{1, 2, \ldots, N-1\}: \\ &c^+[k] = \lambda\, f[k] + z_p\, c^+[k-1], \end{aligned} \tag{12}$$

and

$$\begin{aligned} &k \in \{N-2, N-3, \ldots, 0\}: \\ &c^-[k] = z_p \left(c^-[k+1] - c^+[k]\right). \end{aligned} \tag{13}$$

Compute United Device Architecture (CUDA) texture lookups are clamped for non-normalized texture coordinates, meaning that $c[k] = c[0]\ \forall\ k < 0$ and $c[k] = c[N-1]\ \forall\ k \geq N$. We can use these boundary conditions to determine the starting coefficients $c^+[0]$ and $c^-[N-1]$:

$$\begin{aligned} c^+[0] &= \lambda \left(f[0] + \frac{1}{1 - z_p^{2N}} \sum_{k=0}^{N-1} \left(z_p^{k+1} + z_p^{2N-k}\right) f[k] \right), \\ c^-[N-1] &= -\frac{z_p}{1 - z_p} c^+[N-1]. \end{aligned} \tag{14}$$

The initialization of the c^+ series can be approximated by a summation with M terms.

$$c^+[0] = \lambda \left(f[0] + \sum_{k=0}^{M-1} z_p^{k+1}\, f[k] \right). \tag{15}$$

It turns out that $M = 12$ samples are sufficient to reach the 24 bits of accuracy found in the mantissa of 32-bit floating point numbers. However, when the length of a signal is smaller than M samples, it is advisable to use the exact initialization of $c^+[0]$. Meanwhile, the initial value of $c^-[N-1]$ for the backward recursion is exact and consistent with the clamping of the data at their boundaries.

After both filters have been applied to the samples, the cubic B-spline coefficients are available, with $c[k] = c^-[k]$. A careful analysis of (12) and (13) reveals that the whole process can be realized in-place, if so desired. This is the approach that we follow in Section 4.

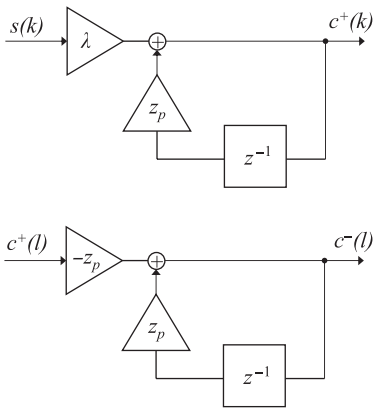

FIGURE 3. The causal and anti-causal filters can be implemented as recursive filters. The top drawing represents the causal filter and the bottom drawing is the anti-causal filter, whereby $l = N - 1 - k$.

4. CUDA IMPLEMENTATION

As mentioned in section 1, the samples often are multidimensional in nature (e.g. RGB colors). The CUDA implementation takes this into account by using templates [14], whereby the sample type is represented by *floatN*. The compiler automatically replaces *floatN* by *float, float2, float3*, depending on the calling code. The function for converting a 1D signal can be implemented in CUDA as

```
//pole for cubic b-spline
#define Pole (sqrt(3.0f)-2.0f)
```

```
template<class floatN>
__device__ void ConvertToBSplineCoefficients(
   floatN* c, uint DataLength)
{
   //compute the overall gain
   const float Lambda = 6.0f;

   //causal initialization
   c[0] = Lambda *
        InitCausalCoeff(c, DataLength);
   //causal recursion
   for (uint k = 1; k < DataLength; k++) {
      c[k] = Lambda * c[k] + Pole * c[k-1];
   }
   //anticausal initialization
   c[DataLength-1] =
      InitAntiCausalCoeff(c, DataLength);
   //anticausal recursion
   for (int k = DataLength-2; 0 <= k; k--) {
      c[k] = Pole * (c[k+1] - c[k]);
   }
}
```

where the causal and anti-causal coefficients are determined by

```
template<class floatN>
__device__ floatN InitCausalCoeff(
   floatN* c, uint DataLength)
{
   const uint Horizon = UMIN(12, DataLength);

   float zk = Pole;
   floatN Sum = c[0];
   for (uint k = 0; k < Horizon; k++) {
      Sum += zk * c[k];
      zk *= Pole;
   }
   return(Sum);
}

template<class floatN>
__device__ floatN InitAntiCausalCoeff(
   floatN* c, uint DataLength)
{

   return((Pole / (Pole - 1.0f)) *
          c[DataLength-1]);
}
```

The data are processed in place, which means that the samples $f[k]$ are passed as input argument and replaced by the coefficients $c[k]$. For 2D images, at first, all rows are processed by the 1D filter. Subsequently, all columns are passed to the filter. For 3D data, the same has to be done in the z-direction. The parallel processing units of the GPU are capable of handling multiple rows or columns simultaneously, which accounts for the acceleration reached by the GPU.

While the processing is trivial for the horizontal rows, where all consecutive data elements lie next to each other in memory, it needs some considerations for processing the data in the remaining directions. Two approaches were explored:

(i) Copying the string of data to a temporary array, and passing this to the functions above. While this seems a logical approach on the CPU, it is not so straightforward on the GPU since there is no way to dynamically allocate memory in a GPU program. To circumvent this, an array of fixed length was declared in-line in the CUDA routine: `float line[MAXSIZE];`. This, however, makes it impossible to process data that would be larger than `MAXSIZE` along any particular axis.

(ii) Changing the routines such that they can handle data that are not consecutive in memory. This is reached by passing an argument `step` to the function, which tells us how far two adjacent data elements are apart. So, `step` is 1 for the x-direction, $width$ for the y-direction and $width \times height$ for the z-direction. As a result, the loops in the routines are changed from

```
for (uint k = 1; k < DataLength; k++) {
   c[k] = Lambda * c[k] + Pole * c[k-1];
}
```

to

```
for (uint k = 1; k < DataLength; k++) {
   c += step;
   *c = Lambda * *c + Pole * c[-step];
}
```

5. RESULTS AND DISCUSSION

We show in Fig. 4 the result of zooming a frame of a video sequence using nearest-neighbor interpolation, linear interpolation, non-prefiltered cubic reconstruction and prefiltered cubic interpolation. Especially for real-time video, it is of great importance that the data can be processed on the fly. Computation times have to be modest in order to prevent latencies and frame-rate drops. Nearest-neighbor and linear interpolation clearly display the well-known blocking and star-shaped artifacts. Cubic B-spline filtering of the unprocessed texture data does not show any distinct image artifacts, but evidently has a smoothing effect on the image. While this may be acceptable for some visualization applications, it is undesirable when the operation is part of more generic signal- and image-processing steps. The prefiltered cubic interpolation does not suffer from the smoothing effect and delivers high-quality interpolation.

The effect of applying interpolation on multiple consecutive processing steps is shown in Fig. 5. It is demonstrated how an image is affected when it is repeatedly rotated (36 steps of 10 degrees each) using different types of filtering. The intermediate results were stored in 32-bit floating point precision and served as input for the next iteration. In the case of prefiltered cubic interpolation, the filter was applied in every iteration before

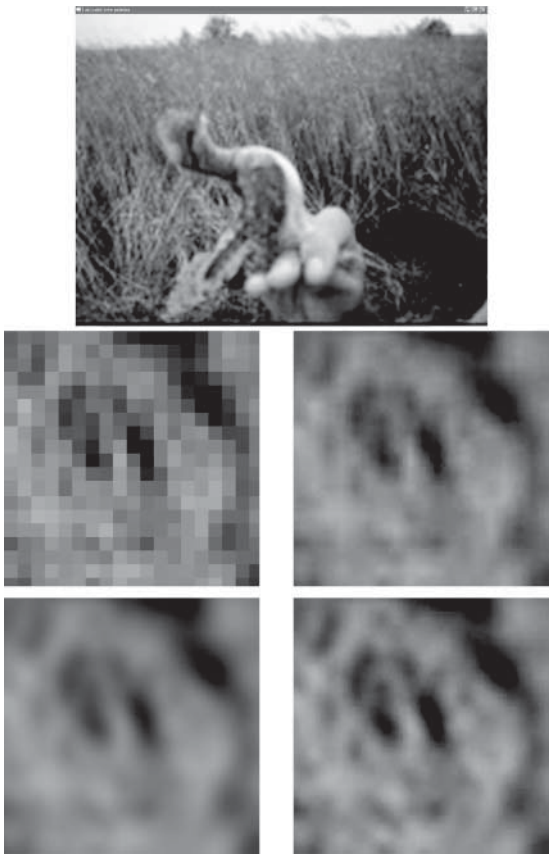

FIGURE 4. Upper image: a single frame from an AVI sequence of 160 × 120 pixels. Middle left: fragment using nearest-neighbor interpolation. Middle right: same fragment using linear interpolation. Bottom left: cubic reconstruction without prefilter. Bottom right: prefiltered cubic interpolation.

FIGURE 5. The Lena image is rotated repeatedly in 36 steps of 10 degrees each. For every iteration, the outcome of the previous step is interpolated on a rotated grid. Top left: nearest-neighbor interpolation. Top right: linear interpolation. Bottom left: cubic reconstruction without prefilter. Bottom right: prefiltered cubic interpolation.

TABLE 1. Prefiltering of 3D voxel data using the two GPU variants and a single-threaded CPU algorithm. The calculation times are decomposed for filtering in the x-, y- and z-direction. All measurements were taken on an Intel Xeon 3.6-GHz machine with an nVidia GeForce 9800 GTX with 512 MB on-board memory.

Algorithm	Data size (voxels)	x (ms)	y (ms)	z (ms)	Total (ms)
CUDA temp array	32^3	0.7	0.5	0.5	1.7
CUDA step	32^3	0.7	0.5	0.5	1.6
CPU	32^3	0.6	0.7	0.9	2.3
CUDA temp array	256^3	39	16	15	70
CUDA step	256^3	125	8	8	142
CPU	256^3	221	425	683	1329
CUDA temp array	$512^2 \times 300$	201	90	122	413
CUDA step	$512^2 \times 300$	550	40	54	644
CPU	$512^2 \times 300$	998	1992	43 204	46 194

interpolating on the rotated grid. In practice, it is not necessary to reapply the prefilter when the source data does not change. Here this is only done to demonstrate the effect of applying interpolation in multiple consecutive iterations.

Nearest-neighbor interpolation is clearly unsuited for repeated interpolation steps. It is strikingly apparent that the repeated unfiltered cubic reconstruction blurs the image even stronger than its linear counterpart. The prefiltered cubic interpolation, though, provides a nice crisp image even after 36 times of resampling. Also it is worth noting that since the templated code can easily deal with multidimensional samples (in this case, RGB), there is no problem to process full-color data in one go. More precisely, it is not necessary to split the data in to monochromatic red, green and blue images for the prefilter.

The time it takes to prefiler a 3D voxel volume, depending on its size, is given in Table 1 for the different proposed prefilter implementations. The CUDA implementation is least efficient when processing data in the x-direction. This is caused by the fact that spatially adjacent samples are consecutive in memory for this direction, which leads to a lot of reading and writing to the same memory banks. In the variant where the data are copied to a local array, this could be partially overcome by addressing the elements that are copied in a non-consecutive order. For the y- and z-direction, the *step* approach is faster. This approach also has the advantage that it is not limited by the fixed size of a temporary array. Table 2 shows the average processing times for prefiltering a frame in the AVI movie (the 160 × 120 frame is also depicted in Fig. 4). Clearly, the filter is fast enough to be used in real-time video rendering.

TABLE 2. Prefiltering of an RGB color frame in an AVI video using the two GPU variants. The calculation times were averaged from timing 100 frames and are decomposed for filtering in the x- and y-direction.

Algorithm	Data size (pixels)	x (ms)	y (ms)	Total (ms)
CUDA temp array	160×120	1.7	1.4	2.1
CUDA step	160×120	0.5	0.2	0.8
CUDA temp array	1024^2	21	8	29
CUDA step	1024^2	30	2	31

Performance measurements of cubic B-spline reconstruction using a data set of 256^3 voxels with the hardware specified in Table 1 delivered 356×10^6 reconstructions per second by applying the techniques described in [15]. Straightforward linear interpolation yielded 486×10^6 reconstructions per second. The small difference between those measurements can be explained by data caching; the tri-cubic weighting performs eight times more lookups than linear interpolation, but these are always located very close to each other, which means that very often the data are still in the fast local cache memory. This implies that the cubic reconstructions during visualization of 2D data and cross sections through 3D data are negligible, and the display process is rather limited by the refresh rate of the monitor (typically 60 or 75 Hz for LCD displays). The performance overhead of the cubic reconstruction may be noticeable only for volume rendering, where the entire volumetric data set needs to be sampled.

The source code that has been used to obtain all presented results and measurements is available for download [10]. The accuracy of the CUDA prefilter code is similar to a single-precision floating-point CPU implementation. The precision issues of cubic texture interpolation on the GPU by using the hardwired linear-interpolation capabilities of the graphics hardware are independent of the prefilter and are discussed in detail in [15].

REFERENCES

[1] Lagrange, M., Marchand, S. and Rault, J.-B. (2005) Long interpolation of audio signals using linear prediction in sinusoidal modeling. *J. Audio Eng. Soc.*, **53**, 891–905.

[2] Keys, R.G. (1981) Cubic convolution interpolation for digital image processing. *IEEE Trans. Acoust., Speech, Signal Process.*, **ASSP-29**, 1153–1160.

[3] Csébfalvi, B. (2008) An evaluation of prefiltered reconstruction schemes for volume rendering. *IEEE Trans. Vis. Comput. Graph.*, **14**, 289–301.

[4] Fedkiw, R., Stam, J. and Jensen, H.W. (2001) Visual Simulation of Smoke. *SIGGRAPH '01: Proc. Computer Graphics and Interactive Techniques*, Los Angeles, CA, USA, August 12–17, pp. 15–22. ACM, New York, NY, USA.

[5] Thévenaz, P., Ruttimann, U.E. and Unser, M. (1998) A pyramid approach to sub-pixel registration based on intensity. *IEEE Trans. Image Process.*, **7**, 27–41.

[6] Ruijters, D., ter Haar Romeny, B.M. and Suetens, P. (2008) Efficient GPU-accelerated Elastic Image Registration. *Proc. IASTED BioMed*, Innsbruck, Austria, February 13–15, pp. 419–424. ACTA Press, Anaheim, CA, USA.

[7] Sigg, C. and Hadwiger, M. (2005) Fast Third-Order Texture Filtering. In Pharr, M. (ed.), *GPU Gems 2: Programming Techniques for High-performance Graphics and General-Purpose Computation*, pp. 313—329. Addison-Wesley.

[8] Thévenaz, P., Blu, T. and Unser, M. (2000) Interpolation revisited. *IEEE Trans. Med. Imaging*, **19**, 739–758.

[9] Buck, I. (2007) GPU Computing: Programming a Massively Parallel Processor. *Code Generation and Optimization, CGO'07*, San Jose, CA, USA, March 11–14, p. 17. IEEE Press, Piscataway, NJ, USA.

[10] Ruijters, D. *CUDA cubic B-spline interpolation (CI)*. http://dannyruijters.nl/cubicinterpolation/.

[11] Schoenberg, I.J. (1946) Contributions to the problem of approximation of equidistant data by analytic functions. *Q. Appl. Math.*, **4**, 45–99 and 112–141.

[12] Unser, M. (1999) Splines: a perfect fit for signal and image processing. *IEEE Signal Process. Mag.*, **16**, 22–38.

[13] Lynn, P.A. (1972) Recursive digital filters with linear-phase characteristics. *Comput. J.*, **15**, 337–342.

[14] (2008) *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*. NVIDIA, Santa Clara, CA.

[15] Ruijters, D., ter Haar Romeny, B.M. and Suetens, P. (2008) Efficient GPU-based texture interpolation using uniform B-splines. *J. Graph. Tools*, **13**, 61–69.