ACCURACY OF GPU-BASED B-SPLINE EVALUATION

Daniel Ruijters
X-Ray Predevelopment
Philips Medical Systems
Best, the Netherlands
email: danny.ruijters@philips.com

Bart M. ter Haar Romeny
Biomedical Engineering,
Image Analysis and Interpretation
Technische Universiteit Eindhoven
Eindhoven, the Netherlands

Paul Suetens
Medical Image Computing,
ESAT/Radiologie
Katholieke Universiteit Leuven
Leuven, Belgium

ABSTRACT
In this article we discuss the accuracy issues that arise, when implementing cubic B-spline interpolation on the Graphics Processing Unit (GPU). Imprecision is inherent to the discrete nature of digital computing, especially when using floating point numbers. However, there are special accuracy issues to deal with, when employing the GPU. The GPU is more and more being regarded as a general purpose parallel co-processor, and currently it is also finding its way in environments, where the algorithmic outcome has great impact, such as in biomedical image analysis. For such applications it is eminent that the accuracy of the results is documented, and properly taken into account. Next to analyzing the accuracy of the cubic B-spline interpolation, we propose a modification to the GPU algorithm, in order to increase its precision, without sacrificing any significant performance.

KEY WORDS
Image Quality, B-spline, Cubic Interpolation, GPU Acceleration

1 Introduction

The arithmetical power of modern GPUs has surpassed the power of mainstream CPUs. Though the GPU performance does not scale equally well for any kind of algorithm, the architecture of the GPU has proven to be very suited for many signal processing tasks in general, and image processing in particular. Therefore, there has been an increasing interest to employ the GPU for such applications [1, 2]. An interest which has motivated nVidia to release the CUDA language [3], which is especially targeted at the implementation of non-visualization algorithms on the GPU (though it can be integrated with visualization oriented APIs, like OpenGL and DirectX).

Also in the medical image processing community there has been a growing interest in the application of the GPU in non-visualization tasks [4, 5, 6, 7]. With this new application domain, the accuracy of the GPU arithmetic has become immensely more important. It might be annoying, when an imprecision leads to an artefact in a game. However, when medical decisions depend on it, the stakes are incomparable. It is not so much that the medical domain demands infinite precision. After all, inaccuracies are inherent to discrete arithmetics. Primarily, it is most important that the accuracy is known and documented, so that it can be taken into account, when implementing an algorithm, and interpreting its results.

This work was originally motivated by the confrontation with unexpected discontinuities (see figure 1), when implementing a GPU version of a B-spline driven deformation field, which was to be used in elastic registration [8] of medical image data. The discontinuities motivated us to investigate their source, and finding improvements and solutions to avoid, or at least diminish them.

2 Uniform B-spline interpolation

Uniform B-spline interpolation has been described exhaustively by Unser [9]. The starting point for any degree of the B-spline function forms the B-spline basis of degree 0, also known as the box function:

$$\beta_0(x) = \begin{cases} 1, & |x| < \frac{1}{2} \\ \frac{1}{2}, & |x| = \frac{1}{2} \\ 0, & |x| > \frac{1}{2} \end{cases} \tag{1}$$

Any subsequent B-spline basis of degree n can be obtained by the recursive convolution of the box function with the B-spline basis of degree $n-1$:

$$\beta_n(x) = \beta_0(x) * \beta_{n-1}(x), \qquad n \geq 1 \tag{2}$$

The derivative of the B-spline basis function can easily be obtained by:

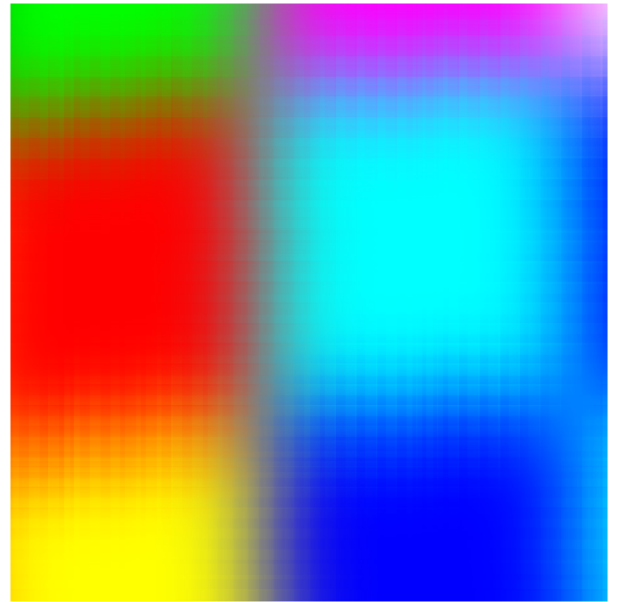
$$\frac{\delta\beta_n(x)}{\delta x} = \beta_{n-1}\left(x + \tfrac{1}{2}\right) - \beta_{n-1}\left(x - \tfrac{1}{2}\right) \tag{3}$$

Which means that the derivative of a B-spline function of degree n, is a B-spline function of degree $n-1$. Further it can be concluded that the B-spline function of degree n has a non-zero derivative up to the n-th order, which is a indicator for the ‘smoothness’ of the function.

The integral of the B-spline basis function of degree n can be expressed as:

$$\int_{-\infty}^{x} \beta_n(x)\,dx = \sum_{k=0}^{+\infty} \beta_{n+1}\left(x - \frac{1}{2} - k\right) \tag{4}$$

(a)



(b)

Figure 1. (a) A GPU-based B-spline deformed image. (b) A zoomed part of the left image. The artifacts are clearly visible: the transition between the blocks should be smooth, whereas it is jerky.

Spline-based interpolation at a given position $x \in \mathbb{R}$ can be written as:

$$s(x) = \sum_{k \in \mathbb{Z}} c(k) \beta_n (x - k) \quad (5)$$

Or in words: the interpolated value s at a given position x, is the summation of the shifted central B-spline β_n, weighted by the B-spline coefficients $c(k)$.

Since B-splines have limited support, the amount of coefficients $c(k)$ that play a role in the interpolation at position x are quite moderate. It should be pointed out that $c(k) = s(k)$ is only the case for the 0th and 1st order B-spline (corresponding to nearest neighbour and linear interpolation). The coefficients for the cubic B-spline can be efficiently obtained, using a causal and anti-causal filter (see [9]).

The 0th (nearest-neighbour), 1st (linear) and 3rd (cubic) order B-spline are most popular. The 0th and 1st order B-spline can be evaluated very rapidly, and do not need any change of sampled values. However often they do not produce a result that is sufficiently close to natural signals. The cubic B-spline is sufficiently smooth, while its support is still quite local (its width is 4), which is favourable for the cost of the interpolation. Since the deformation of organs and other anatomical structures is typically rather smooth, we chose the cubic B-spline to model our deformation field.

3 GPU cubic B-spline evaluation

Sigg and Hadwiger [10] have described how cubic B-spline interpolation can be performed efficiently by the GPU. Their method was primarily targeted at interpolation used in visualization, such as image interpolation and volume rendering applications. Even though in the case of high dynamic rendering (hdr), the result of the interpolation has to surpass the 8-bit accuracy that is common in visualization tasks, the accuracy requirements are typically far less than when B-spline functions are used in biomedical signal and image processing and modeling algorithms.

The method of Sigg and Hadwiger is based on decomposing the cubic interpolation in 2^N weighted linear interpolations, instead of 4^N weighted nearest neighbour interpolations, whereby N denotes the dimensionality. Since linear interpolations are hardwired on the graphics hardware, they can be performed much faster than addressing the set of nearest neighbour lookups, they are composed of.

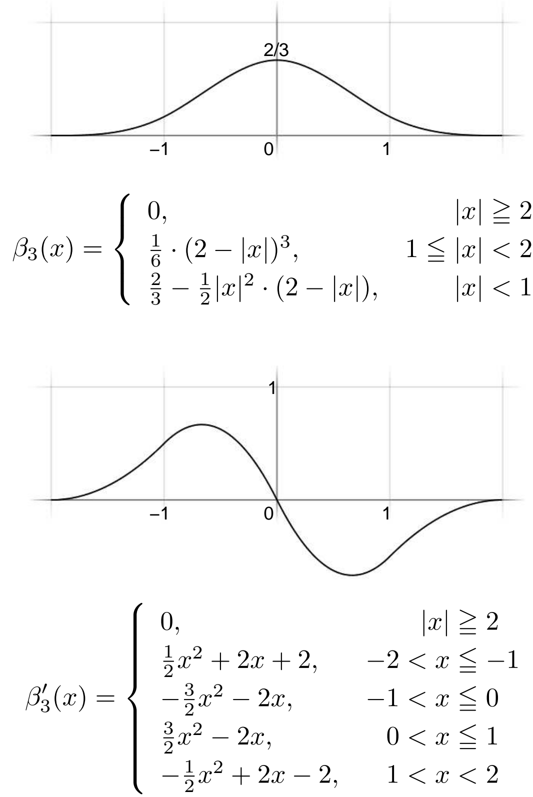
The basic idea can be understood by considering 1D linear interpolation, which can be expressed as follows:

$$f_{i+\alpha} = (1 - \alpha) \cdot f_i + \alpha \cdot f_{i+1} \quad (6)$$

with $i \in \mathbb{N}$ and $\alpha \in [0, 1]$. Building on this equation, the weighted addition of two neighbouring samples can be rewritten to be expressed as a weighted linear interpolation:

$$a \cdot f_i + b \cdot f_{i+1} = (a + b) \cdot f_{i+(b/(a+b))} \quad (7)$$

Evaluating the deformation for any given position, using a cubic B-spline, means the weighted addition of 4^N adjacent coefficients, whereby the weights are determined by the cubic B-spline function (see equation 5 and figure 2).



$$\beta_3(x) = \begin{cases} 0, & |x| \geqq 2 \\ \frac{1}{6} \cdot (2 - |x|)^3, & 1 \leqq |x| < 2 \\ \frac{2}{3} - \frac{1}{2}|x|^2 \cdot (2 - |x|), & |x| < 1 \end{cases}$$

$$\beta_3'(x) = \begin{cases} 0, & |x| \geqq 2 \\ \frac{1}{2}x^2 + 2x + 2, & -2 < x \leqq -1 \\ -\frac{3}{2}x^2 - 2x, & -1 < x \leqq 0 \\ \frac{3}{2}x^2 - 2x, & 0 < x \leqq 1 \\ -\frac{1}{2}x^2 + 2x - 2, & 1 < x < 2 \end{cases}$$

Figure 2. The cubic B-spline, and its 1st order derivative. Note that for all, there is a single equation per quadrant. We use the variant of the B-spline function that is centered around the origin, since this allows us to exploit its symmetry in the GPU programs.

In the 1D case this looks like:

$$\begin{aligned} \tilde{f}_{i+\alpha} &= w_0(\alpha) \cdot c_{i-1} + w_1(\alpha) \cdot c_i + \\ & \quad w_2(\alpha) \cdot c_{i+1} + w_3(\alpha) \cdot c_{i+2} \end{aligned} \tag{8}$$

Using equation 7, we can decompose equation 8 into two weighted linear interpolated lookups.

$$\tilde{f}_{i+\alpha} = g_0 \cdot c_{i+h_0} + g_1 \cdot c_{i+h_1}$$

$$\begin{aligned} g_0 &= w_0 + w_1 \\ g_1 &= w_2 + w_3 \\ h_0 &= (w_1/g_0) - 1 \\ h_1 &= (w_3/g_1) + 1 \end{aligned} \tag{9}$$

Of course this scheme can easily be extrapolated to the N-dimensional case, whereby $g_{\vec{j}} = \prod g_{j_k}$, and $\vec{h}_{\vec{j}} = \sum \vec{e}_k \cdot h_{j_k}$, with k denoting the axis and $\vec{e}_k$ the basis vector. In the 3D case this means that 64 nearest neighbour interpolations can be replaced by 8 linear interpolations. On modern GPUs that means a considerable performance gain.

Sigg and Hadwiger put g_0, h_0 and h_1 as a function of α in a 1D lookup texture (g_1 is redundant), and use this texture to obtain the variables g and h in the fragment program. They suggest using an RGB texture, consisting of 128 samples of 16-bit accuracy, and using linear filtering between the samples. This means that for 3D interpolation 3 lookups in this texture are performed, and from the result the eight coordinates for the linear interpolations are calculated.

4 Texture filtering

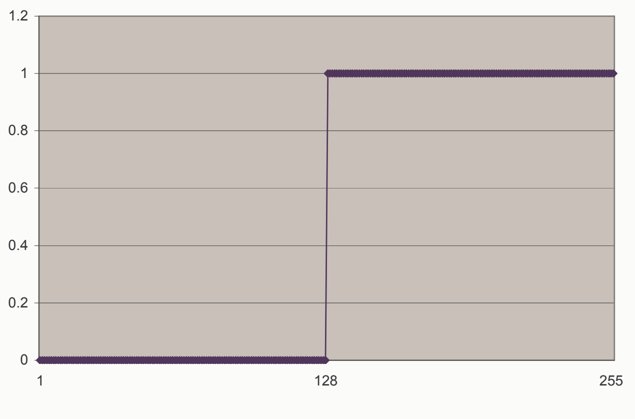
A number of publications have discussed the accuracy of the floating point arithmetic of the GPU [11, 12, 1, 13]. The floating point precision certainly has to be considered, when employing the GPU in *e.g.* medical environments. However, in our case it was not the main source of imprecision.

The method for efficient B-spline interpolation, described in section 3, is based on using the hardware linear interpolation capabilities of the GPU, in order to speed up the evaluation of the B-spline. Exactly in this fast hardwired linear interpolation lies the root cause of the inaccuracies and discontinuities, which we observed.

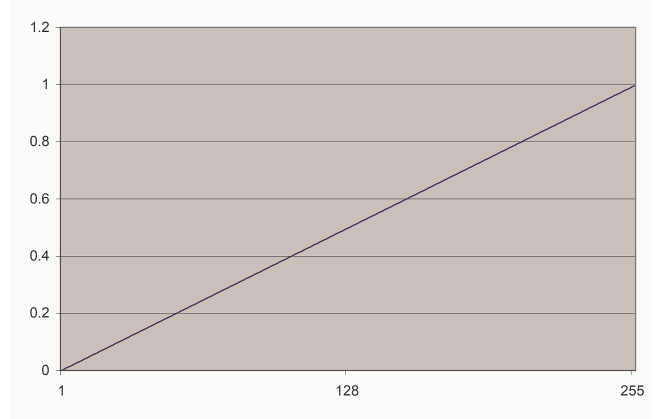
In order to characterise the precision of the linear interpolation capabilities, we created a texture, using a 16-bit gray scale image, whereby half the image was filled with zeros, and the other half with ones. It should be noted that OpenGL normalizes this image, when it is used as a texture, meaning that all pixel values are divided by the maximum capacity of the pixel word (which is 65535 for 16-bit words). Now we zoomed in on the border between the zeros and the ones in the image. This is done by mapping a border pixel with zero on the left edge of an offscreen buffer of 32-bit floating point precision, and a border pixel with one to the right edge. The expected result of this procedure, when discretization and accuracy issues are ignored, is a set of linear increasing samples in the range from 0 to $1/65535$.

If we take a look at figure 3, we can notice that for textures in the 16-bit floating point format, the graph looks exactly as predicted. The graph for the 16-bit integer format, however, shows a ramp function, instead of a linear increasing function. The reason for this behaviour, is the fact that the result of the linear filtered texture fetch, possesses the same accuracy as the texture data format. Since the texture values we are interpolating only differ at the least significant bit, the linear interpolation in integer format is only able to express either 0 or 1.

The 16-bit floating point format on the GPU consists of one sign bit, a five bit exponent and ten bit mantissa. This means that it is impossible to encapsulate data, which consist natively of 12- or 16-bit integer words (popular formats in the medical world), without losing precision. Such data can be casted to 24- or 32-bit floating point format, without loss of precision, but that would mean that more memory is needed to hold the data, which might be a se-



(a)



(b)

Figure 3. Linear interpolation between 0 and $1/65535$. The vertical axis represents the interpolated value times 65535, and the horizontal axis the 256 sample coordinates. (a) shows the results for a 16-bit integer texture, and (b) for a 16-bit floating point texture.

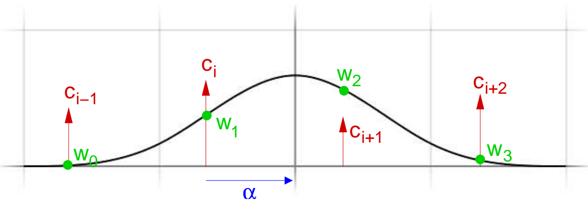


Figure 4. Cubic B-spline interpolation. The image coefficients c are multiplied by the weights $w_n(\alpha)$. The weights are determined by the fractional amount α of the present coordinate, and the B-spline basis function. Index i is the integer part of the coordinate.

rious issue for large datasets (> 1 GB). Further it should be taken into account that hardware linear interpolation of 24- or 32-bit floating point textures is only supported on the very latest generation of GPUs.

Conclusively, we can state that the texture format always poses a tradeoff between 1) interpolation accuracy, 2) lossy data encoding, 3) memory efficiency, or redundant encoding.

5 Improving the accuracy

Next to the linear interpolation, a second cause for imprecision is the usage of a lookup table to determine the weights g_0, g_1, h_0 and h_1. Of course it is possible to increase the number of samples in the lookup table, and to change the data format to 16- or 32-bit floating point, if the hardware supports linear interpolations of these formats. We, however, explore a more radical approach: calculating the weights on the fly, on the GPU.

Equation 8 shows that the variables g and h are a function of the B-spline weights w. The weights w, on their turn, depend on the fractional amount α of the present coordinate, and on the interpolating basis function, which is the cubic B-spline in our case (see figure 4). More specifically:

$$\begin{aligned} w_0(\alpha) &= \beta_3(-\alpha-1) \\ w_1(\alpha) &= \beta_3(-\alpha) \\ w_2(\alpha) &= \beta_3(1-\alpha) \\ w_3(\alpha) &= \beta_3(2-\alpha) \end{aligned} \tag{10}$$

Implementing equation 10 on the GPU, seems to suggest that a number of conditional statements have to be evaluated, see figure 2. Which would be undesirable, since that would lead to a considerable slowdown of the fragment program. However, the conditional statements can be avoided, since the determination of the weights is facilitated by the fact that w_0 is always located in the first quadrant of the cubic B-spline, w_1 always in the second, *etc.* Since the cubic B-spline (as well as its derivatives) consist of a single equation per quadrant, the following equations for the set of weights can be established:

$$\begin{aligned} w_0(\alpha) &= \tfrac{1}{6}\cdot(1-\alpha)^3 \\ w_1(\alpha) &= \tfrac{2}{3}-\tfrac{1}{2}\alpha^2\cdot(2-\alpha) \\ w_2(\alpha) &= \tfrac{2}{3}-\tfrac{1}{2}(1-\alpha)^2\cdot(1+\alpha) \\ w_3(\alpha) &= \tfrac{1}{6}\cdot(\alpha)^3 \end{aligned} \tag{11}$$

After the weights have been established, the variables g and h can be calculated, using equation 9. The pseudo Cg code [14] in appendix 1 illustrates this process for the 2D case.

In table 1 the deviation from the expected interpolated value is given for both cubic interpolation methods. The error is defined as the normalized pixel intensity calculated

Table 1. Accuracy and timing of the two presented cubic interpolation flavours.

Method	$\sum error^2$	Time (ms)
Lookup table	$7.68 \cdot 10^{-4}$	15.0
On-the-fly	$5.83 \cdot 10^{-4}$	15.6

by the fragment program, minus the normalized intensity calculated by the CPU, using double floating point precision. The errors were squared, and summed, for 256^2 pixels. The on-the-fly method is clearly more accurate than the lookup table method, while the duration of the interpolation per frame is only slightly longer. Our figures were measured, using an nVidia QuadroFX 3500 with 256 MB on board memory.

The division $h = w/g$ in equation 9 can be accounted for the fact that the on-the-fly method is somewhat slower. On the same time, it is the division that is the biggest cause for imprecisions, since on the GPU it is only an approximation, instead of an exact division. It is possible to avoid any explicit divisions by using projective texture lookups. In such lookups, the x-, y-, and z- coefficients of the homogeneous texture coordinates are divided by the w-coefficient. All dividers can be united in one w-coefficient, by making use of the following identity:

$$\frac{a}{b} + \frac{c}{d} = \frac{1}{bd} \cdot (ad + cb) \qquad (12)$$

However, the projective texture lookup scheme did not provide any improvements, neither in terms of timing, nor in terms of accuracy.

6 Conclusions

The GPU is more and more being considered as a potent, massively parallel co-processor, and therefore being more often employed in general purpose computing. Especially when the GPU is being used in a situation where the outcome of the algorithm has major impact, such as is the case in biomedical image processing, the accuracy of the GPU output is highly important. In order to assess whether the precision is sufficient, it is necessary that it is documented. In this context, we intended to analyze and discuss the accuracy issues that are encountered when performing cubic B-spline interpolation on the GPU.

We have demonstrated that the hardwired linear interpolation capabilities, though very time efficient, are the main concern when accurate results are required. Further, we have proposed a solution that circumvents the usage of lookup tables in GPU cubic B-spline interpolation. Our method rather performs all calculations inline. Our tests show that this yields more accurate results, while the performance (in terms of speed) hardly suffers.

References

[1] J. Krüger and R. Westermann, “Linear algebra operators for GPU implementation of numerical algorithms,” in *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH’05*, p. 234, 2005.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” *Computer Grphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] I. Buck, “GPU computing: Programming a massively parallel processor,” in *Code Generation and Optimization, CGO’07*, p. 17, Mar 2007.

[4] K. Mueller and F. Xu, “Practical considerations for GPU-accelerated CT,” in *Biomedical Imaging: Nano to Macro, ISBI’2006*, pp. 1184–1187, Apr 2006.

[5] D. Ruijters, D. Babic, R. Homan, P. Mielekamp, B. M. ter Haar Romeny, and P. Suetens, “3D multimodality roadmapping in neuroangiography,” in *SPIE Medical Imaging’07*, vol. 6509, Feb 2007.

[6] R. Strzodka, M. Droske, and M. Rumpf, “Image registration by a regularized gradient flow. A streaming implementation in DX9 graphics hardware,” *Computing*, vol. 73, no. 4, pp. 373–389, 2004.

[7] C. Vetter, C. Guetter, C. Xu, and R. Westermann, “Nonrigid multi-modal registration on the GPU,” in *SPIE Medical Imaging’07*, vol. 6512, Feb 2007.

[8] J. Kybic and M. Unser, “Fast parametric elastic image registration,” *IEEE Trans. Image Processing*, vol. 12, pp. 1427–1442, Nov 2003.

[9] M. Unser, “Splines: A perfect fit for signal and image processing,” *IEEE Signal Processing Magazine*, vol. 16, pp. 22–38, Nov 1999.

[10] C. Sigg and M. Hadwiger, “Fast third-order texture filtering,” in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (M. Pharr, ed.), pp. 313–329, 2005.

[11] G. Da Graça and D. Defour, “Implementation of float-float operators on graphics hardware,” in *7th conference on Real Numbers and Computers, RNC7*, pp. 23–32, Mar 2006.

[12] K. E. Hillesland and A. Lastra, “GPU floating-point paranoia,” in *In ACM Workshop on General Purpose Computing on Graphics Processors*, p. C8, Aug 2004.

[13] A. Thall, “Extended-precision floating-point numbers for GPU computation,” in *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH’06*, p. 52, 2006.

[14] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, “Cg: A system for programming graphics hardware in a C-like language,” *ACM Trans. Graphics*, vol. 22, no. 3, pp. 896–907, 2003.

Appendix 1

The pseudo Cg code [14] below illustrates the cubic B-spline interpolation, with inline evaluation of the variables g and h:

```
float2 CubicInterpolate(
   float2 coordSource : TEXCOORD0,
   uniform sampler2D tex_cp,        // 2D texture with coefficients
   uniform float2 nrCP,             // size of texture tex_cp:
   uniform float2 rec_nrCP          // 1 / nrCP:
) : COLOR
{
   // transform the coordinate from [0,1] to [-0.5, nrCP-0.5]
   float2 coord_hg = coordSource * nrCP - 0.5;
   float2 index = int2(coord_hg);
   float2 fraction = coord_hg - index;
   float2 one_frac = 1.0 - fraction;
   float2 w0 = 1.0/6.0 * one_frac*one_frac*one_frac;
   float2 w1 = 2.0/3.0 - 0.5 * fraction*fraction*(2.0-fraction);
   float2 w2 = 2.0/3.0 - 0.5 * one_frac*one_frac*(2.0-one_frac);
   float2 w3 = 1.0/6.0 * fraction*fraction*fraction;

   float2 g0 = w0 + w1;
   float2 g1 = w2 + w3;
   float2 h0 = (w1 / g0) - 1;
   float2 h1 = (w3 / g1) + 1;

   // determine the coordinates for linear interpolation
   float2 coord00 = index + h0;
   float2 coord10 = index + float2(h1.x, h0.y);
   float2 coord01 = index + float2(h0.x, h1.y);
   float2 coord11 = index + h1;

   // transform the coordinate from [-0.5, nrCP-0.5] to [0,1]
   coord00 = (coord00 + 0.5) * rec_nrCP;
   coord10 = (coord10 + 0.5) * rec_nrCP;
   coord01 = (coord01 + 0.5) * rec_nrCP;
   coord11 = (coord11 + 0.5) * rec_nrCP;

   // fetch the four linear interpolations
   float2 tex_cp00 = tex2D(tex_cp, coord00).xy;
   float2 tex_cp10 = tex2D(tex_cp, coord10).xy;
   float2 tex_cp01 = tex2D(tex_cp, coord01).xy;
   float2 tex_cp11 = tex2D(tex_cp, coord11).xy;

   // weigh along the y-direction
   tex_cp00 = lerp(tex_cp01, tex_cp00, g0.y);
   tex_cp10 = lerp(tex_cp11, tex_cp10, g0.y);

   // weigh along the x-direction
   tex_cp00 = lerp(tex_cp10, tex_cp00, g0.x);
   return tex_cp00;
}
```