Vol. 13, No. 4: 61–69

Efficient GPU-Based Texture Interpolation using Uniform B-Splines

Daniel Ruijters
Philips Healthcare

Bart M. ter Haar Romeny
Eindhoven University of Technology

Paul Suetens
Katholieke Universiteit Leuven

Abstract. This article presents uniform B-spline interpolation, completely contained on the graphics processing unit (GPU). This implies that the CPU does not need to compute any lookup tables or B-spline basis functions. The cubic interpolation can be decomposed into several linear interpolations [Sigg and Hadwiger 05], which are hard-wired on the GPU and therefore very fast. Here it is demonstrated that the cubic B-spline basis function can be evaluated in a short piece of GPU code without any conditional statements. Source code is available online.

1. Introduction

Cubic B-spline interpolation produces results that are noticeably closer to ideal sinc interpolation than nearest neighbor or linear interpolation (see Figure 1). Many image processing (e.g., resampling and elastic registration) and visualization (e.g., volume rendering) applications benefit from the superior interpolation quality, as is illustrated in Figure 2. With the increasing interest in applying the GPU in signal and image processing tasks [Krüger and Westermann 05, Owens et al. 07, Strzodka et al. 04], there is a consid-

1086-7651/08 $0.50 per page

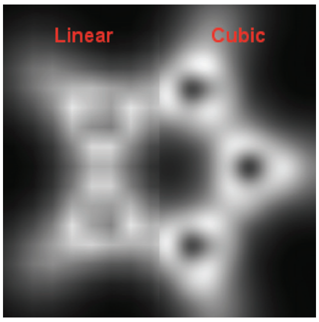


Figure 1. The left part of this figure was generated using trilinear interpolation on a cross-section plane through a voxel data set. The right part is the continuation of the same plane but using 3D cubic B-spline interpolation.

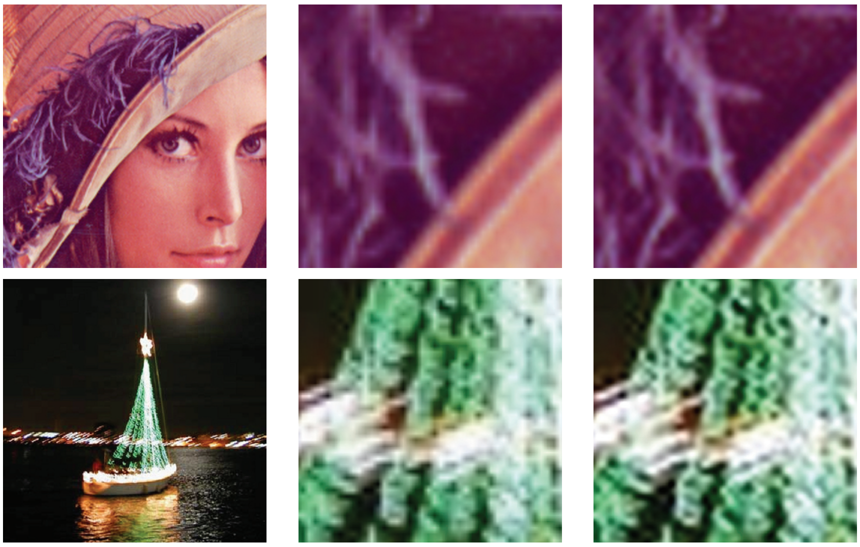


Figure 2. Left: source image. Middle: bilinearly zoomed fragment. Right: zoomed fragment using bicubic interpolation.

erable benefit in having easy and fast cubic interpolation available on the GPU.

2. Uniform B-Spline Interpolation

Uniform spline-based interpolation was introduced by Schoenberg [Schoenberg 46] and has been described exhaustively by Unser [Unser 99]. 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. We use the variant of the B-spline function that is centered around the origin, which is chosen because its symmetry can be exploited within the GPU program:

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

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 $n-1$:

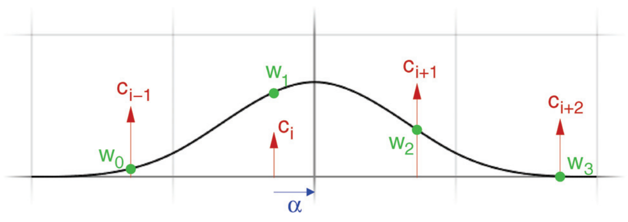
$$\beta_n(x) = \beta_0(x) * \beta_{n-1}(x), \qquad n \geq 1.$$

Spline-based interpolation at a given position $x \in \mathbb{R}$ is the summation of the shifted central B-spline β_n, weighted by the B-spline coefficients $c(k)$:

$$s_n(x) = \sum_{k \in \mathbb{Z}} c(k) \beta_n(x-k). \tag{1}$$

Since B-splines have limited support, the number of coefficients $c(k)$ that play a role in the interpolation at position x is quite moderate. Evaluating cubic B-spline interpolation for any given position involves the weighted addition of the four adjacent coefficients (see Figure 3), which allows Equation (1) to be rewritten as

$$\begin{aligned} s_3(i+\alpha) \quad = \quad & w_0(\alpha) \cdot c(i-1) + w_1(\alpha) \cdot c(i) \\ & + w_2(\alpha) \cdot c(i+1) + w_3(\alpha) \cdot c(i+2), \end{aligned} \tag{2}$$



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

Figure 3. 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 by the B-spline basis function β_3, which consists of a single equation per quadrant. Index i is the integer part of the coordinate.

whereby the weights w depend on the fractional amount α of the present coordinate and on the cubic B-spline basis function. More specifically,

$$\begin{aligned} w_0(\alpha) &= \beta_3(-\alpha-1), \\ w_1(\alpha) &= \beta_3(-\alpha), \\ w_2(\alpha) &= \beta_3(1-\alpha), \\ w_3(\alpha) &= \beta_3(2-\alpha). \end{aligned} \tag{3}$$

It should be pointed out that $c(k) = s(k)$ is only the case for the zeroth- and first-order B-splines (corresponding to nearest neighbor and linear interpolation, respectively). The texture with the coefficients for cubic B-spline interpolation can be readily obtained, using a causal and anti-causal filter [Unser 99].

3. GPU Cubic B-Spline Evaluation

Sigg and Hadwiger have described how cubic B-spline interpolation can be performed efficiently by the GPU [Sigg and Hadwiger 05]. Their method is based on decomposing the cubic interpolation into 2^N weighted linear interpolations, instead of 4^N weighted nearest neighbor interpolations, where N is the dimensionality. Since linear interpolations are hard-wired on the graphics hardware, they can be performed much faster than addressing the corresponding set of nearest neighbor look-ups.

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

$$s_1(i+\alpha) = (1-\alpha)\cdot s_0(i) + \alpha \cdot s_0(i+1),$$

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

$$a \cdot s_0(i) + b \cdot s_0(i+1) = (a+b)\cdot s_1(i + \tfrac{b}{a+b}). \tag{4}$$

Using Equation (4), Equation (2) can be decomposed into two weighted linearly interpolated look-ups:

$$\begin{aligned} s_3(i+\alpha) = g_0 \cdot c_1(i+h_0) &+ g_1 \cdot c_1(i+h_1); \\ g_0 &= w_0 + w_1, \\ g_1 &= w_2 + w_3, \\ h_0 &= (w_1/g_0) - 1, \\ h_1 &= (w_3/g_1) + 1, \end{aligned} \tag{5}$$

where c_1 expresses linear interpolation between the cubic B-spline coefficients.

This scheme can easily be extrapolated to the N-dimensional case, which for 3D cubic interpolation means that 64 nearest neighbor lookups can be replaced by eight linear interpolations. On modern GPUs, that leads to a considerable performance gain.

Sigg and Hadwiger put g_0, h_0, and h_1 as a function of α in a 1D look-up texture (g_1 is redundant) and use this texture to obtain the variables g and h in the GPU program. They suggest using an RGB texture, consisting of 128 samples of 16-bit accuracy, and using linear filtering between the samples. For 3D interpolation, this approach involves three look-ups in this texture, and from the resulting parameters the eight coordinates for the linear interpolations are calculated.

4. Avoiding the Look-up Table

The look-up table distributes the cubic interpolation into two parts in your code: the GPU part that performs the actual interpolation, and the CPU part that creates the look-up table. Further, the look-up table is one of the sources of imprecision, since for any value between its entries linear interpolation is used. Therefore, we explore the on-the-fly calculation of the weights on the GPU, reducing source code complexity and improving the precision.

Equation (5) shows that the variables g and h are a function of the B-spline weights w obtained in Equation (3). Since the B-spline is composed of piecewise polynomials, it would appear that a GPU implementation would involve a number of undesirable conditional statements, leading to a considerable slowdown of the GPU 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) consists of a single equation per quadrant (see Figure 3), the following equations for the set of weights can be established:

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

After the weights have been established, the variables g and h can be calculated using Equation (5). The GPU source code in Section 6 illustrates this process for the 2D case.

Method	**RMS**	**Time** (ms)
Look-up table	$9.39 \cdot 10^{-5}$	0.96
On-the-fly	$8.58 \cdot 10^{-5}$	0.74

Table 1. Accuracy and timing of cubic interpolation with and without using a look-up table. All measurements were obtained on an NVIDIA GeForce 9800 GTX.

In Table 1, the deviation from the expected interpolated value is given for both cubic interpolation methods. The error is defined as the pixel intensity calculated by the GPU program minus the intensity calculated by the CPU using double floating-point precision. The root mean square of the errors was calculated for 512^2 pixels. The on-the-fly method is both more accurate and faster. However, on older graphics hardware (before 2007), the on-the-fly approach is slightly slower than the look-up table method, while still being more accurate.

5. Discussion

It should be noted that there are some precision issues associated with the hard-wired linear texture interpolation. When, e.g., an eight-bit texture is filtered, most people would expect that first the neighboring texture knots are queried, casted to floating point, and then weighted and added. This is, however, not the case; the texture knots are first weighted and added, and then casted to floating point, which limits the precision to the least significant bit of the texture data format [Ruijters et al. 08], as is illustrated in Figure 4. As a consequence, higher accuracies can only be obtained by using larger texture words, and thus at the cost of texture memory consumption.

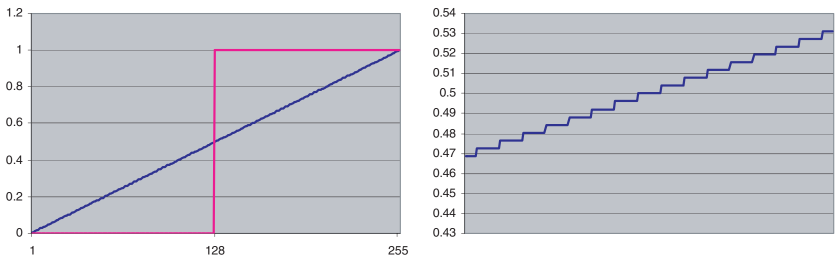


Figure 4. The left graph shows linear interpolation between 0 and 1/65535 using a 16-bit integer texture (purple) and a 16-bit floating-point texture (blue). The right graph zooms in on the blue line, showing the limited precision of the fixed-point texture coordinates.

A further precision issue of the linear texture interpolation is caused by the fact that the accuracy of the texture coordinates is limited to a fixed-point format with eight bits of fractional value [NVIDIA 08]. This means that there are only 254 discrete coordinate positions between two texture knots, as shown in the zoomed graph in Figure 4, which is especially of interest when the knots are far apart (e.g., in a B-spline deformation field for elastic registration). The mentioned texture interpolation accuracy effects are the main cause for the deviations of the on-the-fly method in Table 1.

Performance measurements of the 3D cubic B-spline interpolation, using a CUDA implementation of the on-the-fly method on an NVIDIA GeForce 9800 GTX, reached $356 \cdot 10^6$ cubic interpolations per second. As a reference, a straightforward CUDA implementation using 64 nearest neighbor look-ups delivered $93.6 \cdot 10^6$ cubic interpolations per second, and simple trilinear interpolation delivered $486 \cdot 10^6$ linear interpolations per second. Cubic interpolation was also implemented to run on the CPU. On an Intel Xeon 5140 2.33 GHz, a straightforward implementation delivered $0.45 \cdot 10^6$ cubic interpolations per second, and a multi-threaded SSE implementation managed $10.3 \cdot 10^6$ cubic interpolations per second.

Since the tricubic approach uses eight trilinear interpolations per cubic interpolation, a slowdown of factor eight could be expected. The cubic interpolation scores much better than this, which can be explained by the fact that the mentioned eight linear interpolations are spatially very close to each other, and the data, therefore, is still locally present in the texture cache. This favorable performance aspect, together with the compact code, makes the cubic B-spline interpolation an attractive solution for fast and high-quality interpolation on the GPU.

6. Source Code

The CUDA code [Buck 07] below illustrates the cubic B-spline interpolation, with inline evaluation of the variables g and h. It should be noted that the code can be ported very easily to, e.g., Cg [Mark et al. 03], the OpenGL Shading Language, or DirectX HLSL.

```
__device__ float interpolate_bicubic(texture tex, float x, float y)
{
   // transform the coordinate from [0,extent] to [-0.5, extent-0.5]
   float2 coord_grid = make_float2(x - 0.5, y - 0.5);
   float2 index = floor(coord_grid);
   float2 fraction = coord_grid - index;

   float2 one_frac = 1.0 - fraction;
   float2 one_frac2 = one_frac * one_frac;
   float2 fraction2 = fraction * fraction;
   float2 w0 = 1.0/6.0 * one_frac2 * one_frac;
   float2 w1 = 2.0/3.0 - 0.5 * fraction2 * (2.0-fraction);
   float2 w2 = 2.0/3.0 - 0.5 * one_frac2 * (2.0-one_frac);
   float2 w3 = 1.0/6.0 * fraction2 * fraction;

   float2 g0 = w0 + w1;
   float2 g1 = w2 + w3;
   // h0 = w1/g0 - 1, move from [-0.5, extent-0.5] to [0, extent]
   float2 h0 = (w1 / g0) - 0.5 + index;
   float2 h1 = (w3 / g1) + 1.5 + index;

   // fetch the four linear interpolations
   float tex00 = tex2D(tex, h0.x, h0.y);
   float tex10 = tex2D(tex, h1.x, h0.y);
   float tex01 = tex2D(tex, h0.x, h1.y);
   float tex11 = tex2D(tex, h1.x, h1.y);

   // weigh along the y-direction
   tex00 = lerp(tex01, tex00, g0.y);
   tex10 = lerp(tex11, tex10, g0.y);

   // weigh along the x-direction
   return lerp(tex10, tex00, g0.x);
}
```

Acknowledgments. We would like to thank Javier Olivan Bescos for helping to correct some bugs in the cubic interpolation code.

References

[Buck 07] Ian Buck. "GPU Computing: Programming a Massively Parallel Processor." In *Code Generation and Optimization, CGO'07*, p. 17. Los Alamitos, CA: IEEE Press, 2007.

[Krüger and Westermann 05] Jens Krüger and Rüdiger Westermann. "Linear Algebra Operators for GPU Implementation of Numerical Algorithms." In *ACM SIGGRAPH 2005 Courses*, Article No. 234. New York: ACM Press, 2005.

[Mark et al. 03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. "Cg: A System for Programming Graphics Hardware in a C-like Language." *ACM Trans. Graphics* 22:3 (2003), 896–907.

[NVIDIA 08] NVIDIA Corporation. "Linear Filetering." In *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*, Appendix D.2, 2008.

[Owens et al. 07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." *Computer Graphics Forum* 26:1 (2007), 80–113.

[Ruijters et al. 08] Daniel Ruijters, Bart M. ter Haar Romeny, and Paul Suetens. "Accuracy of GPU-based B-Spline Evaluation." In *Computer Graphics and Imaging (CGIM)*, pp. 117–122. Calgary, AB, Canada: ACTA Press, 2008.

[Schoenberg 46] I. J. Schoenberg. "Contributions to the Problem of Approximation of Equidistant Data by Analytic Functions." *Quarterly of Applied Mathematics* 4:1 (1946), 45–99 and 112–141.

[Sigg and Hadwiger 05] Christian Sigg and Markus Hadwiger. "Fast Third-Order Texture Filtering." In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, edited by Matt Pharr, pp. 313–329. Boston, MA: Addison-Wesley Professional, 2005.

[Strzodka et al. 04] R. Strzodka, M. Droske, and M. Rumpf. "Image Registration by a Regularized Gradient Flow: A Streaming Implementation in DX9 Graphics Hardware." *Computing* 73:4 (2004), 373–389.

[Unser 99] Michael Unser. "Splines: A Perfect Fit for Signal and Image Processing." *IEEE Signal Processing Magazine* 16:6 (1999), 22–38.

Web Information:

The CUDA source code regarding 2D and 3D cubic B-spline interpolation, as presented in this work, is available at http://jgt.akpeters.com/papers/RuijtersEtAl08/.

Daniel Ruijters, Cardio/Vascular Innovation, Philips Healthcare, Veenpluis 6, 5680 DA Best, The Netherlands (danny.ruijters@philips.com)

Bart M. ter Haar Romeny, Eindhoven University of Technology, Department of Biomedical Engineering, Biomedical Image Analysis & Interpretation, Den Dolech 2 – WH 2.108, PO Box 513, 5600 MB Eindhoven, The Netherlands (B.M.terHaarRomeny@tue.nl)

Paul Suetens, Katholieke Universiteit Leuven, Medical Imaging Research Center, UZ Gasthuisberg, Herestraat 49 – Bus 07003, 3000 Leuven, Belgium (paul.suetens@esat.kuleuven.be)

Received June 23, 2008; accepted in revised form February 3, 2009.