DOI: 10.1111/cgf.14377
High-Performance Graphics 2021
N. Binder and T. Ritschel
(Guest Editors)

Volume 40 (2021), Number 8

Hardware Adaptive High-Order Interpolation for Real-Time Graphics

D. Lin[1] and L. Seiler[2] and C. Yuksel[1]

[1] University of Utah
[2]Facebook Reality Labs

Abstract

Interpolation is a core operation that has widespread use in computer graphics. Though higher-order interpolation provides better quality, linear interpolation is often preferred due to its simplicity, performance, and hardware support.
We present a unified refactoring of quadratic and cubic interpolations as standard linear interpolation plus linear interpolations of higher-order terms and show how they can be applied to regular grids and (triangular/tetrahedral) simplexes Our formulations can provide significant reduction in computation cost, as compared to typical higher-order interpolations and prior approaches that utilize existing hardware linear interpolation support to achieve higher-order interpolation. In addition, our formulation allows approximating the results by dynamically skipping some higher order terms with low weights for further savings in both computation and storage. Thus, higher-order interpolation can be performed adaptively, as needed.
We also describe how relatively minor modifications to existing GPU hardware could provide hardware support for quadratic and cubic interpolations using our approach for both texture filtering operations and barycentric interpolation.
We present a variety of examples using triangular, rectangular, tetrahedral, and cuboidal interpolations, showing the effectiveness of our higher-order interpolations in different applications.

CCS Concepts
• ***Computing methodologies*** → ***Graphics processors; Texturing;***

1. Introduction

Parameter interpolation is widely used in computer graphics. Most commonly, it is performed linearly (i.e. bilinearly in 2D and trilinearly in 3D). For example, 2D texture sampling on the GPU uses bilinear interpolation to blend the color of the nearest four pixels, and shading normal (or any other attribute on a triangle) is computed using a linear combination of the three triangle vertex normals (or attributes).

However, linear interpolation is prone to visual artifacts like Mach bands. Such problems can be resolved with high-order interpolations, such as quadratic or cubic, which are known to provide superior quality.

Yet, lack of hardware support for high-order interpolation makes it undesirable for real-time graphics applications with limited computation budgets. This can be attributed to the computation cost of high-order interpolation and the significant hardware changes needed for supporting them directly.

In this paper, we present a unified mathematical formulation that covers quadratic and cubic interpolation, expressing them as linear interpolation plus some high-order difference terms. This provides a simpler form than common high-order interpolation formulations in 2D and 3D domains. We also explain how this approach can be extended to interpolation in a simplex (triangles and tetrahedrons).

Our formulations require less computation than standard high-order interpolation approaches and the state-of-art high-order interpolation methods performed on existing hardware [Csé18]. In addition, it is suitable for an efficient hardware implementation that requires relatively minor changes to existing linear interpolation pipeline on today's GPUs, as we describe.

Moreover, it allows clamping high-order difference terms when they are below a threshold, saving a sizeable amount of computation when high frequency details are sparse. This leads to an adaptive high-order interpolation solution, which incurs additional computation over linear interpolation only when needed.

In applications not suitable for a hardware implementation, our formulation allows skipping additional storage of high-order data, saving substantial amount of storage and computation cost.

We show examples in a wide range of real-time graphics rendering domains to show that our adaptive high-order interpolation with our proposed hardware can significantly improve visual quality, using only 1× to 2× more computation than linear interpolation in typical cases. Note that this is significantly cheaper than 5× to

7× more computation required by the state-of-art high-order interpolation on existing hardware [Csé18; Csé19].

We begin by providing the background and related prior work in Section 2. In Section 3 we describe our high-order interpolation formulations in grids and explain how they can be used in practical applications, such as texture filtering. Section 4 presents the details of how the existing hardware texture filtering pipeline can be modified to provide support for our high-order interpolation formulations. Then, in Section 5 we describe our high-order interpolations for simplexes, such as triangles and tetrahedra. Possible hardware acceleration techniques for simplex meshes are described in Section 6. We present our evaluation and results in Section 7 and conclude in Section 8.

2. Related Work

Before we discuss the details of our approach, we summarize the related prior work in this section.

2.1. Interpolation for Grids of Data

Many graphics applications require reconstructing smooth signals from (1D, 2D, or 3D) grids of data, usually stored as images or textures. For that, reconstruction filters are required. Bilinear or trilinear interpolation provides a cheap way to generate continuous signal out of discrete samples and they are supported by most graphics hardware. Yet, cubic interpolation is known to significantly improve the quality of texture filtering [SH05], volume rendering [ML94], and temporal anti-aliasing [YLS20].

Keys [Key81] introduced a family of cubic cardinal splines that interpolates the sampling data. Mitchell and Netravali [MN88] derived BC-splines to describe a more general family of cubic reconstruction filters that may or may not interpolate the data. The family of splines is parameterized by B and C. All cardinal splines have B=0. A separable bicubic filter using the BC-spline family has been implemented in shader code [Bjo04] to provide high quality image magnification filtering. Sigg and Hadwiger [SH05] proposed refactoring a bicubic/tricubic B-Spline filter (B=1,C=0), into a linear combination of four/eight hardware bilinear/trilinear taps.

By modulating the source image with a checkerboard pattern, Csébfalvi [Csé18] solves the problem of negative bilinear/trilinear weights, allowing Catmull-Rom spline filter to be partially accelerated by hardware in a similar way. Since Catmull-Rom spline (B=0,C=1/2) interpolates the original data, it does not have the over-blurring problem of B-Spline filters. In a survey by Moller at al. [MMMY97], Catmull-Rom splines are verified to achieve the lowest reconstruction error in the entire family of BC-Spline filters.

More recently, Csébfalvi [Csé19] proposed a method that uses hardware trilinear interpolation results for gradient estimation to do tricubic density filtering for volumes. This method closely approximates the result of Catmull-Rom spline interpolation but uses fewer taps. However, even with partial hardware acceleration proposed by these methods, bicubic and tricubic interpolation remain significantly more expensive than bilinear and trilinear interpolation.

Numerous works have proposed FPGA implementation of cubic interpolation. Due to the high computational complexity of bicubic interpolation, a direct FPGA implementation of bicubic interpolation requires a lot of hardware resources [NA05]. To reduce the computational complexity, many FPGA implementations [LSC*08; WDLY11; GNSS14] limit the scope to handle specific image operation like scaling, where the bicubic weight pattern is repeated across the whole image and only needs to be computed once. Orthogonal methods like quantizing the interpolation weights [ZLZ*10], approximating the cubic kernel with multiple piecewise linear function [LSC*10; GNSS14], and using a mixture of cubic and linear function [BBGB20] have been applied in FPGA implementations. Sanaullah et al. [SKH16] presented an FPGA implementation of tricubic interpolation for molecular dynamics simulations. In comparison to these FPGA implementations, our method adaptively reduces the computational cost, and only requires slight modification to the existing GPU. Thus, our method can easily utilize the power of existing texture units to provide high order interpolation for a wide range of graphics applications.

A graphics workstation system [MBDM97] has been made to support hardware bicubic interpolation at half of the rate of trilinear interpolation [Map06]. However, modern GPUs do not provide extra hardware to support higher-order filtering. Hardware implementations of higher-order filtering into standard GPU texture units cannot be justified if they require a large amount of dedicated logic that could instead be devoted to performing more bilinear interpolations per clock. Proposals to reuse existing texture logic require at least four bilinear texture reads per sample, plus shader execution time to select the bilinear sample positions [SH05; Csé18].

There is a category of adaptive interpolation techniques [MH15] for image resizing that derives interpolation weights from local spatial features (e.g. edge orientation statistics) of the images to provide better visual quality than bicubic interpolation. However, these methods generally involve expensive computation and are highly specified for the task of image resizing. In comparison, our approach is similar to the hierarchical form of high-order FEM [ZTZ05], where the difference between the higher-order element node values and lower-order element interpolation results are used as part of the high-order element. We adaptively discard small high-order terms purely based on the mathematical formulation of bicubic (and other high-order) interpolation. Our method is targeted at improving the performance of high order interpolation, and our method handles a wide range of applications in real-time rendering.

2.2. Interpolation for Simplexes

Triangles and tetrahedrons are common building blocks of computer graphics. Shading a triangular mesh relies on interpolating vertex attributes, such asposition, normal, and texture coordinates. Linear interpolation of triangle vertex attributes are widely supported by graphics hardware.

Higher order simplex interpolation has not been supported by graphics hardware, but research work has revealed problems that could benefit from higher order interpolation in triangles. Brown [Bro99] proposed using quadratic Bézier triangles [Far93] to interpolate an cosine highlight function over a triangle to approximate Phong shading [Pho75], avoiding the cost of renormalization of normal vectors. Research work has proposed hardware that directly uses quadratic interpolation in screen space to interpolate a variety of vertex attributes without the need for perspective division [Sei98; ASS*01]. PN Triangles [VPBM01] constructs cubic and quadratic Bézier patches on the fly from local triangle attributes to achieve smooth visual appearance using low-poly meshes. With a similar goal, Phong Tessellation [BA08] introduces a computationally simple way to turn a triangle into a quadratic patch.

Tetrahedral interpolation is widely used in Finite Element Methods [ZTZ05] for various kinds of simulation. Bargteil and Cohen [BC14] proposes using quadratic elements to reduce the simulation error and artifacts of deformable bodies. To reduce computation, they adaptively choose between linear and quadratic tetrahedral elements based on the difference of the predicted values of edge midpoints interpolated by each method. Phong deformation [Jam20] blends per-tet average gradients and per-vertex deformation gradients to achieve a quadratic tetrahedral interpolator to achieve higher order of accuracy for embedded deformation.

Different from these methods, we propose a unified mathematical formulation for quadratic and cubic interpolation of simplexes of different dimensions. Our adaptive high-order triangular interpolation can benefit from hardware acceleration by slightly modifying the existing hardware used for rasterization. If the the triangles or the tetrahedrons are structured data, our method can use modified texture units to accelerate interpolation.

3. High-Order Interpolation in Grids

Interpolation in 1D, 2D, and 3D grids are commonplace in computer graphics for applications like texture filtering. Though high-order is known to produce better quality, linear interpolation is more popular in practice, because it has direct hardware support on GPUs.

In this section, we discuss high-order interpolation in grids and present how we can reorder the terms in quadratic and cubic interpolations to represent them as linear interpolation plus *high-order difference terms*. This includes simplified forms using fewer data points. We also describe how we can take advantage of our reordering to provide adaptive high-order interpolation, such that linear interpolation is used wherever high-order interpolation would not produce visible improvement. Finally, we present how our approach can be used in typical applications. Most importantly, our reordering provides a convenient mechanism for modifying the existing hardware texture filtering system on GPUs to support high-order filtering, as we describe in Section 4.

Notation: We use $\mathbf{P}_i$, $\mathbf{P}_{ij}$, and $\mathbf{P}_{ijk}$ to represent the data points (i.e. grid vertices) to be interpolated in 1D, 2D, and 3D grids, respectively, where $i,j,k \in \mathbb{Z}$. The evaluation position within the interpolation domain is represented using localized parameters $s,t,q \in [0,1]$. The data points at the corners of the interpolation domain correspond to $i,j,k \in \{0,1\}$. For representing values at element/edge centers, we use $i,j,k = ½$. The interpolation functions are represented as ${}_{\blacksquare}\mathbf{C}_m^{n\mathrm{D}}$, where $n \in \{1,2,3\}$ is the dimension and $m \in \mathbb{N}$ is the number of data points and high-order difference terms used in the interpolation, using $\mathbf{L}$ for linear, $\mathbf{Q}$ for quadratic, and $\mathbf{C}$ for cubic interpolations. We also present simpler interpolation functions that omit one or more higher-order terms and are represented as ${}_{\square}\mathbf{C}_m^{n\mathrm{D}}$, as opposed to standard interpolation functions ${}_{\blacksquare}\mathbf{C}_m^{n\mathrm{D}}$ that include all terms.

3.1. 1D Interpolation

Let $\mathcal{L}^1$ represent the linear interpolation operator, such that

$$\mathcal{L}_s^1(\mathbf{P}_0, \mathbf{P}_1) = (1-s)\mathbf{P}_0 + s\mathbf{P}_1 \,.$$

Obviously, linear interpolation along an edge in 1D simply uses this operator

$$ {}_{\blacksquare}\mathbf{L}_2^{1\mathrm{D}}(s) = \mathcal{L}_s^1(\mathbf{P}_0, \mathbf{P}_1) \,. \tag{1}$$

For defining quadratic interpolation along this edge, we can specify the desired value $\mathbf{P}_{½}$ at the center of the edge. The resulting quadratic interpolation can be written in Bézier form as

$$ {}_{\blacksquare}\mathbf{Q}_3^{1\mathrm{D}}(s) = (1-s)^2\mathbf{P}_0 + 2(1-s)s\mathbf{P}^* + s^2\mathbf{P}_1 \,, \tag{2}$$

where $\mathbf{P}^* = 2\mathbf{P}_{½} - (\mathbf{P}_0 + \mathbf{P}_1)/2$. By expanding and rearranging the terms, we can write

$$ {}_{\blacksquare}\mathbf{Q}_3^{1\mathrm{D}}(s) = (1-s)\mathbf{P}_0 + s\mathbf{P}_1 + 4(1-s)s\left(\mathbf{P}_{½} - \frac{\mathbf{P}_0+\mathbf{P}_1}{2}\right) \,. \tag{3}$$

Here, the first two terms are the linear interpolation between $\mathbf{P}_0$ and $\mathbf{P}_1$, and the last term includes the difference between the desired center value $\mathbf{P}_{½}$ and the linear interpolation at the center. Therefore, by defining this difference as

$$\mathbf{D}_{½} = \mathbf{P}_{½} - \frac{\mathbf{P}_0+\mathbf{P}_1}{2} \,, \tag{4}$$

we can write the quadratic interpolation as

$$ {}_{\blacksquare}\mathbf{Q}_3^{1\mathrm{D}}(s) = {}_{\blacksquare}\mathbf{L}_2^{1\mathrm{D}}(s) + 4(1-s)s\,\mathbf{D}_{½} \,. \tag{5}$$

Thus, quadratic interpolation becomes linear interpolation plus a second-order difference term.

For cubic interpolation, we consider the derivatives at the vertices. Let $\mathbf{P}_{\vec{0}}$ and $-\mathbf{P}_{\vec{1}}$ represent the desired derivatives of the interpolated value at the two vertices. Along with the values at the vertices, they uniquely define a cubic function. Similar to the quadratic case, we define the difference values $\mathbf{D}_{\vec{0}}$ and $\mathbf{D}_{\vec{1}}$ between the desired derivatives and the derivative of linear interpolation, such that

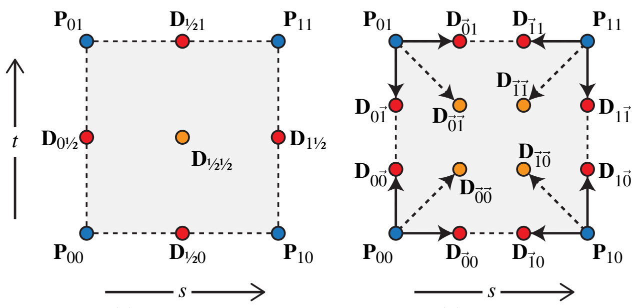
$$\mathbf{D}_{\vec{0}} = \mathbf{P}_{\vec{0}} - (\mathbf{P}_1 - \mathbf{P}_0) \qquad \text{and} \qquad \mathbf{D}_{\vec{1}} = \mathbf{P}_{\vec{1}} - (\mathbf{P}_0 - \mathbf{P}_1) \,.$$

Then, the *cubic interpolation along an edge* can be written as

$$ {}_{\blacksquare}\mathbf{C}_4^{1\mathrm{D}}(s) = {}_{\blacksquare}\mathbf{L}_2^{1\mathrm{D}}(s) + (1-s)s\,\mathcal{L}_s^1(\mathbf{D}_{\vec{0}}, \mathbf{D}_{\vec{1}}) \,, \tag{6}$$

where the first term is, again, linear interpolation and the second term includes the third-order components with a linear interpolation of the difference values.

These quadratic and cubic formulations in Equations 5 and 6 are mathematically identical to standard second-degree and third-degree interpolations. Their advantage is purely in computation, as they allow us to begin with linear interpolation and factor out all high-order terms. This formulation will be particularly helpful for modifying existing linear filtering hardware on GPUs to perform higher-order interpolation, as we explain in Section 4.



(a) *Biquadratic* **(b)** *Bicubic*

Figure 1: *The difference terms of biquadratic and bicubic interpolations in 2D.*

3.2. 2D Interpolation

In 2D, we rely on the bilinear interpolation operator

$$\begin{aligned}\mathcal{L}^2_{st}(\mathbf{P}_{00},\mathbf{P}_{10},\mathbf{P}_{01},\mathbf{P}_{11}) &= (1-t)\mathcal{L}^1_s(\mathbf{P}_{00},\mathbf{P}_{10}) + t\mathcal{L}^1_s(\mathbf{P}_{01},\mathbf{P}_{11})\\ &= (1-s)(1-t)\mathbf{P}_{00} + s(1-t)\mathbf{P}_{10}\\ &\quad + (1-s)t\mathbf{P}_{01} + st\mathbf{P}_{11}\end{aligned}$$

Bilinear interpolation simply uses this operator, such that

$$_{\blacksquare}\mathbf{L}^{2D}_4(s,t) = \mathcal{L}^2_{st}(\mathbf{P}_{00},\mathbf{P}_{10},\mathbf{P}_{01},\mathbf{P}_{11})\,. \tag{7}$$

Biquadratic interpolation involves 9 control points: 4 at the vertices, 4 at the edge centers, and one at the middle of the rectangle they form (Figure 1a). Similar to the 1D case, we can write biquadratic interpolation using difference values at the edge centers $\mathbf{D}_{\frac12 0}$, $\mathbf{D}_{\frac12 1}$, $\mathbf{D}_{0\frac12}$, $\mathbf{D}_{1\frac12}$, and the difference value at the middle position $\mathbf{D}_{\frac12\frac12}$. If we omit this middle difference value by taking $\mathbf{D}_{\frac12\frac12} = \mathbf{0}$, the resulting quadratic interpolation can be written as

$$\begin{aligned}_{\square}\mathbf{Q}^{2D}_8(s,t) = {}_{\blacksquare}\mathbf{L}^{2D}_4(s,t) &+ 4(1-s)s\,\mathcal{L}^1_t(\mathbf{D}_{\frac12 0},\mathbf{D}_{\frac12 1})\\ &+ 4(1-t)t\,\mathcal{L}^1_s(\mathbf{D}_{0\frac12},\mathbf{D}_{1\frac12})\,.\end{aligned} \tag{8}$$

If the middle difference term $\mathbf{D}_{\frac12\frac12}$ is non-zero, biquadratic interpolation becomes

$$_{\blacksquare}\mathbf{Q}^{2D}_9(s,t) = {}_{\square}\mathbf{Q}^{2D}_8(s,t) + 16(1-s)s(1-t)t\,\mathbf{D}_{\frac12\frac12} \tag{9}$$

and $\mathbf{D}_{\frac12\frac12}$ can be written using the desired middle value $\mathbf{P}_{\frac12\frac12}$ as

$$\mathbf{D}_{\frac12\frac12} = \mathbf{P}_{\frac12\frac12} - {}_{\square}\mathbf{Q}^{2D}_8(\tfrac12,\tfrac12)\,. \tag{10}$$

The standard bicubic interpolation involves 16 control points. Similar to the quadratic case, if we omit the four difference terms in the interior of the rectangle and only consider the edges (Figure 1b), we get

$$\begin{aligned}_{\square}\mathbf{C}^{2D}_{12}(s,t) = {}_{\blacksquare}\mathbf{L}^{2D}_4(s,t)&\\ &+ (1-s)s\,\mathcal{L}^2_{st}(\mathbf{D}_{\vec{0}0},\mathbf{D}_{\vec{1}0},\mathbf{D}_{\vec{0}1},\mathbf{D}_{\vec{1}1})\\ &+ (1-t)t\,\mathcal{L}^2_{st}(\mathbf{D}_{0\vec{0}},\mathbf{D}_{1\vec{0}},\mathbf{D}_{0\vec{1}},\mathbf{D}_{1\vec{1}})\,.\end{aligned} \tag{11}$$

Note that, with this formulation, bicubic interpolation turns into three linear interpolations: the first one interpolates the four vertex values and the other two interpolate the difference in the derivatives.

The standard bicubic interpolation with 16 control points $_{\blacksquare}\mathbf{C}^{2D}_{16}$ can be written in a similar form by using the second derivatives, such that the desired second derivatives

$$\mathbf{P}_{\vec{i}\vec{j}} = (-1)^{i+j}\frac{\partial^2}{\partial s\,\partial t}\,{}_{\blacksquare}\mathbf{C}^{2D}_{16}(i,j) \tag{12}$$

are achieved using difference terms

$$\begin{aligned}\mathbf{D}_{\vec{i}\vec{j}} &= \mathbf{P}_{\vec{i}\vec{j}} - (-1)^{i+j}\frac{\partial^2}{\partial s\,\partial t}\,{}_{\square}\mathbf{C}^{2D}_{12}(i,j)\\ &= \mathbf{P}_{\vec{i}\vec{j}} - (\mathbf{P}_{ij} - \mathbf{P}_{(1-i)j} - \mathbf{P}_{i(1-j)} + \mathbf{P}_{(1-i)(1-j)})\\ &\qquad - (\mathbf{D}_{\vec{i}(1-j)} - \mathbf{D}_{\vec{i}j} + \mathbf{D}_{(1-i)\vec{j}} - \mathbf{D}_{i\vec{j}})\,.\end{aligned} \tag{13}$$

for $i,j \in \{0,1\}$. With these internal difference terms, standard bicubic interpolation can be written as

$$\begin{aligned}_{\blacksquare}\mathbf{C}^{2D}_{16}(s,t) &= {}_{\square}\mathbf{C}^{2D}_{12}(s,t)\\ &\quad + (1-s)s(1-t)t\,\mathcal{L}^2_{st}(\mathbf{D}_{\vec{0}\vec{0}},\mathbf{D}_{\vec{1}\vec{0}},\mathbf{D}_{\vec{0}\vec{1}},\mathbf{D}_{\vec{1}\vec{1}})\,.\end{aligned} \tag{14}$$

In this form, standard bicubic interpolation involves an additional bilinear interpolation over $_{\square}\mathbf{C}^{2D}_{12}$.

3.3. 3D Interpolation

This concept of representing higher-order interpolation as a sum of linear interpolation and higher-order terms can be extended to 3D as well. In 3D, we can use the trilinear interpolation operator

$$\begin{aligned}\mathcal{L}^3_{stq}(\mathbf{P}_{000},\mathbf{P}_{100},\mathbf{P}_{010},\mathbf{P}_{110},\mathbf{P}_{001},\mathbf{P}_{101},\mathbf{P}_{011},\mathbf{P}_{111}) &=\\ (1-q)\,\mathcal{L}^2_{st}(\mathbf{P}_{000},\mathbf{P}_{100},\mathbf{P}_{010},\mathbf{P}_{110})&\\ +\,q\,\mathcal{L}^2_{st}(\mathbf{P}_{001},\mathbf{P}_{101},\mathbf{P}_{011},\mathbf{P}_{111})&\end{aligned}$$

This operator linearly blends two bilinear operators and trilinear interpolation simply uses it, such that

$$\begin{aligned}_{\blacksquare}\mathbf{L}^{3D}_8(s,t,q) = \mathcal{L}^3_{stq}(\,&\mathbf{P}_{000},\mathbf{P}_{100},\mathbf{P}_{010},\mathbf{P}_{110},\\ &\mathbf{P}_{001},\mathbf{P}_{101},\mathbf{P}_{011},\mathbf{P}_{111}\,)\,,\end{aligned} \tag{15}$$

where $\mathbf{P}_{ijk}$ with $i,j,k \in \{0,1\}$ are the data values at the eight vertices of a cube.

Again, for our quadratic and cubic interpolation functions, we omit the higher-order difference terms inside the cube and on the face centers of the cube, resulting

$$\begin{aligned}_{\square}\mathbf{Q}^{3D}_{20}(s,t,q) = {}_{\blacksquare}\mathbf{L}^{3D}_8(s,t,q)&\\ &+ 4(1-s)s\;\mathcal{L}^2_{tq}(\,\mathbf{D}_{\frac12 00},\mathbf{D}_{\frac12 10},\mathbf{D}_{\frac12 01},\mathbf{D}_{\frac12 11}\,)\\ &+ 4(1-t)t\;\mathcal{L}^2_{sq}(\,\mathbf{D}_{0\frac12 0},\mathbf{D}_{1\frac12 0},\mathbf{D}_{0\frac12 1},\mathbf{D}_{1\frac12 1}\,)\\ &+ 4(1-q)q\,\mathcal{L}^2_{st}(\,\mathbf{D}_{00\frac12},\mathbf{D}_{10\frac12},\mathbf{D}_{01\frac12},\mathbf{D}_{11\frac12}\,)\end{aligned} \tag{16}$$

$$\begin{aligned}_{\square}\mathbf{C}^{3D}_{32}(s,t,q) = {}_{\blacksquare}\mathbf{L}^{3D}_8(s,t,q)&\\ &+ (1-s)s\;\mathcal{L}^3_{stq}(\,\mathbf{D}_{\vec{0}00},\mathbf{D}_{\vec{1}00},\mathbf{D}_{\vec{0}10},\mathbf{D}_{\vec{1}10},\\ &\qquad\qquad\quad \mathbf{D}_{\vec{0}01},\mathbf{D}_{\vec{1}01},\mathbf{D}_{\vec{0}11},\mathbf{D}_{\vec{1}11}\,)\\ &+ (1-t)t\;\mathcal{L}^3_{tqs}(\,\mathbf{D}_{0\vec{0}0},\mathbf{D}_{0\vec{1}0},\mathbf{D}_{0\vec{0}1},\mathbf{D}_{0\vec{1}1},\\ &\qquad\qquad\quad \mathbf{D}_{1\vec{0}0},\mathbf{D}_{1\vec{1}0},\mathbf{D}_{1\vec{0}1},\mathbf{D}_{1\vec{1}1}\,)\\ &+ (1-q)q\;\mathcal{L}^3_{qst}(\,\mathbf{D}_{00\vec{0}},\mathbf{D}_{00\vec{1}},\mathbf{D}_{10\vec{0}},\mathbf{D}_{10\vec{1}},\\ &\qquad\qquad\quad \mathbf{D}_{01\vec{0}},\mathbf{D}_{01\vec{1}},\mathbf{D}_{11\vec{0}},\mathbf{D}_{11\vec{1}}\,)\,.\end{aligned} \tag{17}$$

Note that standard triquadratic and tricubic interpolations ${}_{\blacksquare}\mathbf{Q}^{3D}_{27}$ and ${}_{\blacksquare}\mathbf{C}^{3D}_{64}$ use 27 and 64 control points, respectively. Therefore, the versions above that skip the interior difference terms save 7 and 32 control points for quadratic and cubic interpolation, respectively.

3.4. Adaptive High-Order Interpolation

Notice that all our high-order interpolation formulations contain high-order difference terms, i.e. **D**-*terms*, defined as the difference between a quantity approximated by lower-order interpolation and the desired value. These **D**-terms are indicators of how well linear interpolation approximates the desire values.

When the **D**-terms are close to zero, high-order interpolation produces results with relatively small difference from linear interpolation. In such cases, simply using linear interpolation instead may be an acceptable approximation. This opens up the possibility of *adaptive high-order interpolation* that skips the high-order difference terms when they are close to zero, determined by a user-defined threshold $\mathbf{D}_{\min}$.

At first glance, this may appear as a minor simplification, particularly considering software interpolation. However, as we explain in Section 4, adaptive interpolation can be used for more than doubling the throughput of a hardware implementation.

3.5. High-Order Texture Filtering

Bicubic image filtering is known to produce superior image quality, as compared to bilinear, and it is often used for enlarging raster images. Using a Catmull-Rom spline, interpolation along 1D can be written as

$$\begin{aligned}\mathcal{S}^1_s(\mathbf{P}_{-1},\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2) = & -s(1-s)^2/2\,\mathbf{P}_{-1}\\ & +\left((1-s)^3+3s(1-s)^2+s^2(1-s)/2\right)\mathbf{P}_0\\ & +\left(s^3+3s^2(1-s)+s(1-s)^2/2\right)\mathbf{P}_1\\ & -s^2(1-s)/2\,\mathbf{P}_2\end{aligned}$$

1D cubic interpolation can simply use this function

$$ {}_{\blacksquare}\mathbf{C}^{1D}_4(s) = \mathcal{S}^1_s(\mathbf{P}_{-1},\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2)\,. \tag{18}$$

For our cubic interpolation, however, we must first compute the **D**-terms. Using a Catmull-Rom spline (with uniform parameterization) that interpolates the data points $\mathbf{P}_{-1}$, $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$, the **D**-terms can be written as

$$\mathbf{D}_{\vec{i}} = \mathbf{P}_i - \frac{\mathbf{P}_{i-1}+\mathbf{P}_{i+1}}{2} \tag{19}$$

for $i \in \{0,1\}$. Then, we can use our ${}_{\blacksquare}\mathbf{C}^{1D}_4$ formulation in Equation 6.

Similarly, bicubic interpolation using a 4×4 block of texel samples shown in Figure 2 can be defined using Catmull-Rom splines, such that

$$\begin{aligned}{}_{\blacksquare}\mathbf{C}^{2D}_{16}(s,t) = \mathcal{S}^1_s(\, & \mathcal{S}^1_t(\,\mathbf{P}_{-1-1},\mathbf{P}_{-10},\mathbf{P}_{-11},\mathbf{P}_{-12}),\\ & \mathcal{S}^1_t(\,\mathbf{P}_{0-1},\,\mathbf{P}_{00},\,\mathbf{P}_{01},\,\mathbf{P}_{02}\,),\\ & \mathcal{S}^1_t(\,\mathbf{P}_{1-1},\,\mathbf{P}_{10},\,\mathbf{P}_{11},\,\mathbf{P}_{12}\,),\\ & \mathcal{S}^1_t(\,\mathbf{P}_{2-1},\,\mathbf{P}_{20},\,\mathbf{P}_{21},\,\mathbf{P}_{22}\,)\,)\,.\end{aligned} \tag{20}$$

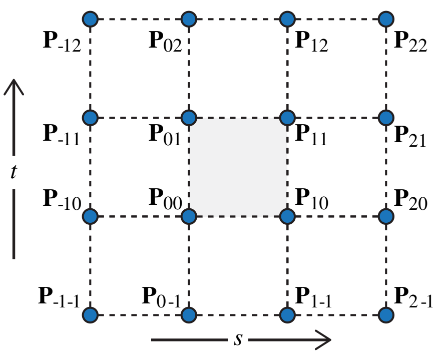


Figure 2: *The texel data used for high-order filtering in 2D.*

In our formulation, the $\mathbf{D}_{\vec{i}j}$ and $\mathbf{D}_{i\vec{j}}$ terms can be computed using Equation 19. These are sufficient for evaluating ${}_{\square}\mathbf{C}^{2D}_{12}$ in Equation 14. For ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$ we also need $\mathbf{D}_{\vec{i}\vec{j}}$ with $i, j \in \{0,1\}$. They can be computed using the desired second derivatives

$$\begin{aligned}\mathbf{D}_{\vec{i}\vec{j}} = \mathbf{P}_{ij} & - \frac{\mathbf{P}_{(1-i)j}+\mathbf{P}_{i(1-j)}+\mathbf{P}_{i(3j-1)}+\mathbf{P}_{(3i-1)j}}{2}\\ & + \frac{\mathbf{P}_{(1-i)(1-j)}+\mathbf{P}_{(3i-1)(1-j)}+\mathbf{P}_{(1-i)(3j-1)}+\mathbf{P}_{(3i-1)(3j-1)}}{4}\end{aligned} \tag{21}$$

Note that computing these last four **D**-terms for ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$ involves combining 9 data points $\mathbf{P}_{ij}$ within a 3×3 block.

Quadratic interpolation for texture filtering is not as popular. This is because it involves accessing the same amount of texture data as cubic interpolation and it cannot deliver the same quality. Nonetheless, it is still superior to linear filtering and requires less computation than cubic. Therefore, it might be preferable for some applications.

We define quadratic interpolation similarly, using Catmull-Rom splines. In this case, the **D**-terms ensure that the interpolation matches the Catmull-Rom spline at the middle points. Thus, in 1D we can write

$$\mathbf{D}_{½} = \frac{-\mathbf{P}_{-1}+\mathbf{P}_0+\mathbf{P}_1-\mathbf{P}_2}{16}\,. \tag{22}$$

In 2D, we can compute the **D**-terms for ${}_{\square}\mathbf{Q}^{2D}_8$ similarly. As for ${}_{\blacksquare}\mathbf{Q}^{2D}_9$, the computation of the middle difference term involves all 16 data points, using

$$\mathbf{D}_{½½} = {}_{\blacksquare}\mathbf{C}^{2D}_{16}(½,½) - {}_{\square}\mathbf{Q}^{2D}_8(½,½)\,. \tag{23}$$

Note that in both biquadratic and bicubic interpolations, evaluating the **D**-terms for ${}_{\square}\mathbf{Q}^{2D}_8$ and ${}_{\square}\mathbf{C}^{2D}_{12}$ are much simpler than the additional **D**-terms needed for ${}_{\blacksquare}\mathbf{Q}^{2D}_9$ and ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$. Also, ${}_{\square}\mathbf{Q}^{2D}_8$ and ${}_{\square}\mathbf{C}^{2D}_{12}$ do not need to access the corner texels $\mathbf{P}_{-1-1}$, $\mathbf{P}_{2-1}$, $\mathbf{P}_{-12}$, and $\mathbf{P}_{22}$. These corner texels are only needed for computing the internal **D**-terms used by ${}_{\blacksquare}\mathbf{Q}^{2D}_9$ and ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$.

The equations for triquadratic and tricubic cases are similar. Again, the **D**-terms for ${}_{\square}\mathbf{Q}^{3D}_{20}$ and ${}_{\square}\mathbf{C}^{3D}_{32}$ are much simpler to compute than ${}_{\blacksquare}\mathbf{Q}^{3D}_{27}$ and ${}_{\blacksquare}\mathbf{C}^{3D}_{64}$.

4. High-Order Hardware Texture Filtering

The versions for quadratic and cubic interpolations we present in Section 3 provide convenient mechanisms for hardware implementation. In this section, we discuss the details of existing GPU texture filtering hardware and how it can be modified to support high-order interpolation using our formulations.

4.1. Texture Filtering on Current GPUs

Bilinear interpolation is a fundamental texture filtering operation on the GPU. Current GPUs implement it in one of two ways [MSY19]. The first way is to linearly interpolate pairs of texels along one dimension, then linearly interpolate the results along the other dimension, using 3 linear interpolations. The second way is to compute a weight for each of the four texels, then multiply them and add the four results. This requires an extra multiplier but allows more parallelism. It also requires only one renormalization for floating point textures instead of three. Our adaptive higher order interpolation method works with both of these implementation methods.

Texture units in GPUs perform certain filtering operations in multiple steps. For example, trilinear filtering uses two bilinear operations. The result of each bilinear operation, or *BOP* for short, is scaled and accumulated to produce the trilinear result. This is illustrated in Figure 3. Alternately, the GPU could perform the two BOPs in parallel, linearly interpolating the pair of results. The area involved is similar and the former method allows pure bilinear filtering operations to go twice as fast, while supporting two two-step trilinear operations in parallel, so the raw throughput of trilinear operations is the same. Therefore, we will assume that the GPU implements BOPs, although our method can also be adapted to work with a GPU designed with trilinear filtering as the basic operation.

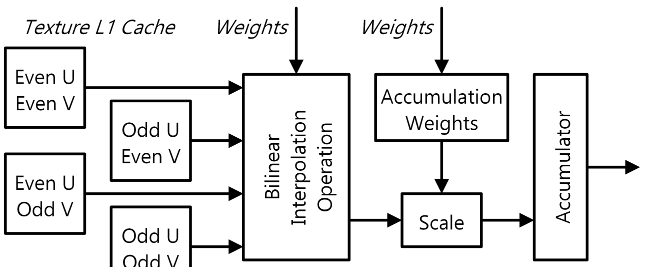


Figure 3: *Texture filter unit on current GPUs. Bilinear operations (BOPs) can be scaled and accumulted to perform more complex filtering. The L1 texture cache is divided into four interleaved banks to allow parallel access to an unaligned 2×2 of texels.*

Another multi-step texture filtering operation is anisotropic filtering. In this case, up to 16 individual bilinear or trilinear filter results are blended to approximate an elongated filter region. Therefore, even if a texture unit implements trilinear filtering as its basic operation, it still needs to scale and accumulate the results of multiple BOPs to support anisotropic filtering. Since the number of BOPs is the bottleneck for computation, we use that to represent the computational cost of our texture interpolation techniques.

4.2. Current Texture Cache Access Methods

Texture units in GPUs are fed by deep queues of pending filter operations. This provides time to compute filter weights, determine which texels will be accessed, and load the texels into caches. Typically there are multiple cache levels, e.g. a large cache shared over the whole GPU that feeds L2 and specialized L1 caches that are dedicated to individual texture units. The operation queue is typically sized to cover the latency of accessing off-chip memory, so that BOPs can usually occur without off-chip memory read delays.

The L1 caches in texture units are specialized to allow a BOP per cycle without cache read delays. Bilinear texture filtering requires accessing an unaligned 2×2 of texels. For a standard cache or memory this could require 1, 2, or 4 memory accesses, depending on how the unaligned 2×2 block maps onto the aligned blocks that are stored in cache or memory. This is not a problem when reading texels from memory or from the L2 cache, since nearby texel values are likely to be used in later texture filter operations. But requiring multiple read cycles from the L1 cache has obvious problems for maintaining the desired texture processing rate of one BOP per clock.

The standard solution is to divide the L1 texture cache into four banks, based on the low order bits of the U and V indices of the stored texture coordinates. The left side of Figure 3 illustrates how this works: texels are stored in banks based on whether their U and V indices are even or odd. Texture dimensions are padded to some tile size, typically at least 4 texels, so for each texture, one quarter of its texels fall into each of the four buffers. As a result, a bilinear interpolation unit can receive an unaligned 2×2 of texels on each clock by computing the appropriate memory addresses for each of the four banks of texture L1 cache.

Finally, the number of L1 cache banks used depends on the number of texels that must be accessed in parallel. For example, GPUs that perform a trilinear filter in a single clock cycle require eight banks. Four of the banks provide an unaligned 2×2 access for even mip levels or even slice numbers. The other four banks provide an unaligned 2×2 access for odd mip levels or odd slice numbers.

4.3. High-Order Texture Filtering on Hardware

High-order filtering with our formulations begins with linear interpolation (using $_\blacksquare\mathbf{L}_4^{2\text{D}}$ in 2D and $_\blacksquare\mathbf{L}_8^{3\text{D}}$ in 3D). This is exactly what the current hardware is designed to do. Then, we add the $\mathbf{D}$-terms. Notice that with all our quadratic and cubic formulations, except for $_\blacksquare\mathbf{Q}_9^{2\text{D}}$, the $\mathbf{D}$-terms can be processed in groups of 4. The computation of each group is the same as a BOP multiplied by a scale factor.

Computing the scale factors (e.g. $(1-s)s$, $(1-t)t$, etc.) needed for the steps involving the $\mathbf{D}$-terms can be pipelined, so they do not require extra clock cycles. The amount of additional logic required is relatively small as well. For example, with 8-bit precision $(1-s)s$ can be computed with an 8×8 multiplier, though we used 32-bit floating point numbers for tests in Section 7.

Computing the $\mathbf{D}$-terms is also straightforward, given access to the necessary texels. Figure 4 shows that the $\mathbf{D}$-term calculation is

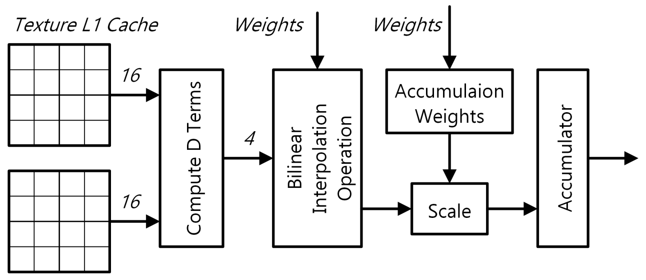


Figure 4: *Proposed texture filter unit for higher order interpolation. The L1 texture cache is divided into 4, 8, 16, or 32 interleaved banks to allow parallel access.*

pipelined between the texel L1 cache read and the bilinear operation. Therefore, utilizing the existing BOP logic unit, we can include the **D**-terms for high-order filtering by simply using the same unit multiple times. This way, high-order filtering simply involves additional steps, similar to computing trilinear filtering with two bilinear filtering steps or anisotropic filtering in multiple steps on current hardware.

The cost of computing the **D**-terms depends on the filtering operation being performed. Computing the **D**-terms along the edges needed for quadratic and cubic interpolations $_{\square}\mathbf{Q}^{2D}_{8}$, $_{\square}\mathbf{C}^{2D}_{12}$, $_{\square}\mathbf{Q}^{3D}_{20}$, and $_{\square}\mathbf{C}^{3D}_{32}$, only require adders and bit-shifters (for division by a power of 2), which do not require much extra area or power. The internal **D**-terms for quadratic and cubic interpolations $_{\blacksquare}\mathbf{Q}^{2D}_{9}$, $_{\blacksquare}\mathbf{C}^{2D}_{16}$, $_{\blacksquare}\mathbf{Q}^{3D}_{27}$, and $_{\blacksquare}\mathbf{C}^{3D}_{64}$ involve more expensive operations to evaluate from the data points **P**, so they require a greater area cost when modifying existing texture filtering hardware. Based on the number of **D**-terms, we can process $_{\square}\mathbf{Q}^{2D}_{8}$ with 2 BOPs, $_{\square}\mathbf{C}^{2D}_{12}$ with 3 BOPs, $_{\square}\mathbf{Q}^{3D}_{20}$ with 5 BOPs, and $_{\square}\mathbf{C}^{3D}_{32}$ with 8 BOPs in total. The interpolations that involve the internal **D**-terms, such as $_{\blacksquare}\mathbf{Q}^{2D}_{9}$, $_{\blacksquare}\mathbf{C}^{2D}_{16}$, $_{\blacksquare}\mathbf{Q}^{3D}_{27}$, and $_{\blacksquare}\mathbf{C}^{3D}_{64}$, would require 3, 4, 6, and 16 BOPs, respectively.

These numbers imply a dramatic reduction of performance compared to bilinear interpolation, but we can improve performance by using adaptive high-order interpolation, as we discussed in Section 3.4. Computation of the **D**-terms is pipelined in advance of performing BOPs, so adaptive filtering can be implemented by checking the four **D**-terms to be used for the next step. If they all are below the given threshold $\mathbf{D}_{\min}$, the bilinear step can be skipped. If all of the **D**-terms are below the threshold, higher order interpolation requires just 1 BOP and so has the same performance as linear interpolation.

For high-order interpolation involving more than 2 steps, instead of using pre-defined groups of **D**-terms, it is possible to group the **D**-terms that pass the threshold for minimizing the number of steps. However, this would require more complex logic, so we assume pre-defined groups of **D**-terms in our evaluation (Section 7). Still, it is possible to save power by simply turning off the multiplier for any **D**-term that is below the threshold.

4.4. Texture Cache Access for Higher Order Filtering

High-order texture interpolation involves accessing more data, but this does not necessarily inflate the off-chip memory bandwidth, since the filter kernels near neighboring texels largely overlap. Therefore, we can expect the higher level caches to efficiently handle the data flow. However, the L1 cache must be changed to allow accessing more texels in parallel.

As illustrated in Figure 4, using more banks in the L1 cache allows computing the **D**-terms in parallel to achieve peak performance. Then, groups of 4 **D**-terms can be passed to the existing bilinear interpolation logic. The L1 cache can be implemented using 4, 8, 16, or 32 banks with different levels of performance, as we discuss in Appendix A.

5. Extensions to Interpolation in Simplexes

In Section 3 we described our difference-based formulations of quadratic and cubic interpolations for structured sample data in a grid. In this section, we extend this concept to arbitrary simplexes, such as line segments (1D), triangles (2D), and tetrahedra (3D). Similar to our notation for grids, (cubic) interpolations are represented as $_{\triangle}\mathbf{C}^{nD}_{m}$, if they omit some terms, and $_{\blacktriangle}\mathbf{C}^{nD}_{m}$, if they include all terms, where n is the dimension and m is the total number of interpolated data values, including the simplex vertices.

5.1. Interpolation in a Simplex

A simplex in n-dimensions is defined by $n+1$ vertices. Let $\mathbf{P}_i$ where $i \in \{0,\ldots,n\}$ represent the data values at the vertices and $\mathbf{w} = [w_0,\ldots,w_n]^T$ is the barycentric coordinates of the interpolation point, forming a partition of unity, such that

$$\sum_{i=0}^{n} w_i = 1\,. \tag{24}$$

Linear interpolation is defined as a simple weighted average, using

$$_{\blacktriangle}\mathbf{L}^{nD}_{n+1}(\mathbf{w}) = \sum_{i=0}^{n} w_i \mathbf{P}_i\,. \tag{25}$$

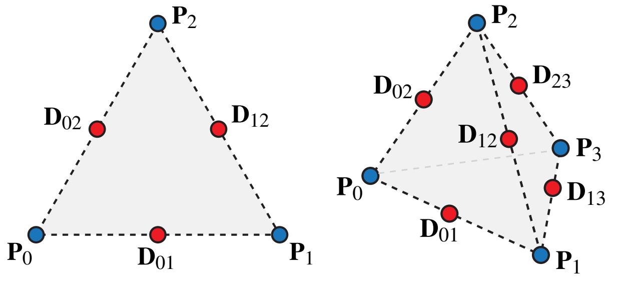


Figure 5: *The high-order difference terms for quadratic interpolation on a triangle and in a tetrahedron.*

For quadratic interpolation, we define high-order difference terms $\mathbf{D}_{ij}$ between each pair of vertices i and j with $i<j$, as shown in Figure 5. Then, quadratic interpolation can be written as

$$_{\blacktriangle}\mathbf{Q}^{nD}_{(n+1)(n+2)/2}(\mathbf{w}) = {}_{\blacktriangle}\mathbf{L}^{nD}_{n+1}(\mathbf{w}) + \sum_{i=0}^{n-1}\sum_{j=i+1}^{n} 4 w_i w_j \mathbf{D}_{ij}\,. \tag{26}$$

There are $n(n+1)/2$ quadratic **D**-terms: a line has 1, a triangle

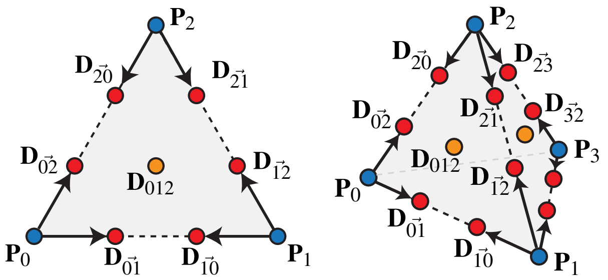


Figure 6: *The high-order difference terms for cubic interpolation on a triangle and in a tetrahedron.*

has 3, and a tetrahedron has 6. Thus, we get ${}_{\blacktriangle}\mathbf{Q}^{2D}_{6}$ and ${}_{\blacktriangle}\mathbf{Q}^{3D}_{10}$ for triangles and tetrahedra, respectively.

For defining cubic interpolation, we use $\mathbf{D}_{\vec{ij}}$ to specify the desired derivatives along each edge, as shown in Figure 6. Then, we can write our cubic interpolation formulation as

$$
\begin{aligned}
{}_{\triangle}\mathbf{C}^{nD}_{(n+1)^2}(\mathbf{w}) = {}_{\blacktriangle}\mathbf{L}^{nD}_{n+1}(\mathbf{w}) \\
+ \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} w_i w_j \left(w_i \mathbf{D}_{\vec{ij}} + w_j \mathbf{D}_{\vec{ji}} \right) .
\end{aligned}
\tag{27}
$$

This formulation defines ${}_{\triangle}\mathbf{C}^{2D}_{9}$ and ${}_{\triangle}\mathbf{C}^{3D}_{16}$ for triangles and tetrahedra, respectively, only considering the derivatives along the edges of the simplex. For triangles, however, cubic interpolation can also specify a desired value $\mathbf{P}_{012}$ at the center using an interior difference term $\mathbf{D}_{012} = \mathbf{P}_{012} - {}_{\triangle}\mathbf{C}^{2D}_{9}(\mathbf{w}_{center})$, where $\mathbf{w}_{center} = \mathbf{1}/3$ is the barycentric coordinates of the center of the triangle. Including higher-dimensional simplexes, we can write

$$
\begin{aligned}
{}_{\blacktriangle}\mathbf{C}^{nD}_{(n+1)(n^2+1)}(\mathbf{w}) = {}_{\triangle}\mathbf{C}^{nD}_{(n+1)^2}(\mathbf{w}) \\
+ \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} 27 w_i w_j w_k \mathbf{D}_{ijk} .
\end{aligned}
\tag{28}
$$

Here, $\mathbf{D}_{ijk} = \mathbf{P}_{ijk} - {}_{\triangle}\mathbf{C}^{nD}_{(n+1)^2}(\mathbf{w}^{ijk}_{center})$, where $\mathbf{w}^{ijk}_{center}$ is the barycentric coordinates for the center of the triangle formed by verices i, j, k of the simplex (e.g. for a tetrahedron, $\mathbf{w}^{012}_{center} = [{}^1\!/_3 \; {}^1\!/_3 \; {}^1\!/_3 \; 0]^T$).

Note that this formulation provides $n(n-1)(n+1)/6$ interior $\mathbf{D}$-terms: a triangle has 1 and a tetrahedron has 4. The resulting interpolations for triangles and tetrahedra are ${}_{\blacktriangle}\mathbf{C}^{2D}_{10}$ and ${}_{\blacktriangle}\mathbf{C}^{3D}_{20}$, respectively.

5.2. Practical Interpolation Applications Using Simplexes

High-order interpolations within a grid described in Section 3 allow computing the **D**-terms from the data points on-the-fly. In the case of arbitrary simplex meshes, however, a typical application needs to pre-compute the **D**-terms. This is because determining the desired derivatives or the edge/triangle center values typically requires traversing the simplex topology, using discrete differential geometry operators [MDSB03].

There are exception, however. For example, reconstructing smooth normals for shading a triangle, the **D**-terms can be computed on-the-fly from the triangle's vertex positions and normals [VPBM01]. Also, the **D**-terms can be computed on-the-fly for barycentric filtering using mesh color textures [Yuk17] or patch textures [MSY19] for providing hardware texture filtering support for mesh colors [YKH10], as they use structured triangular texel distributions.

Another example is a regular simplex mesh in a grid, such as a triangular mesh or a tetrahedral mesh with vertices on a regular grid. A 2D grid cell can be represented using two triangles and a 3D grid cell can be formed by 5 or 6 tetrahedra. Indeed, the vertex data for such meshes can be stored in 2D or 3D textures. This offers a cheaper alternative to texture filtering using fewer data points, and it can be used for applications like color space conversion [KNPH95] that can benefit from high-order filtering.

When the **D**-terms cannot be computed during interpolation and must be pre-computed, adaptively skipping some **D**-terms can provide storage, memory bandwidth, and computation savings at run time.

6. Hardware Interpolation for Simplex Meshes

For providing hardware-accelerated interpolation, there are two cases to consider: regular simplex meshes on a grid and arbitrary simplex meshes.

The data for regular meshes of triangles or tetrahedra can be stored in 2D or 3D grids. In that case, hardware interpolation can be supported in a similar way as described in Section 4. The only differences are the interpolation functions and the subsets of the texel data blocks used in the interpolation (Section 6.1).

Arbitrary simplex meshes, however, cannot be handled similarly and require a different treatment (Section 6.2).

6.1. Hardware Interpolation for Regular Simplexes

For regular triangles on a grid, the existing bilinear interpolation unit can be modified to support barycentric linear interpolation [MSY19]. One of the four texels is weighted as zero and the other three use weights that produce the same result as barycentric interpolation. This can be done for both methods of designing the bilinear operation block, that is using three linear interpolation units or using four parallel multipliers.

Our method for supporting nonlinear triangular interpolation extends this method. Reading an unaligned 4×4 array of texels allows **D**-terms to be computed. These are then fed into the bilinear operation block, along with appropriate weights. In general, up to three **D**-terms can be retired per cycle through the bilinear operation logic.

A similar method may be used for regular tetrahedra in a 3D grid. Linear interpolation can be achieved with an unaligned $2 \times 2 \times 2$ array of texels. Four of the texel values are used and the rest are ignored. The four chosen vertices are gathered into a single BOP. If the BOP is implemented using four multipliers, these get the four barycentric parameters. If the BOP is implemented using three linear interpolations, the weights for the first pair are $w_0/(w_0 + w_1)$

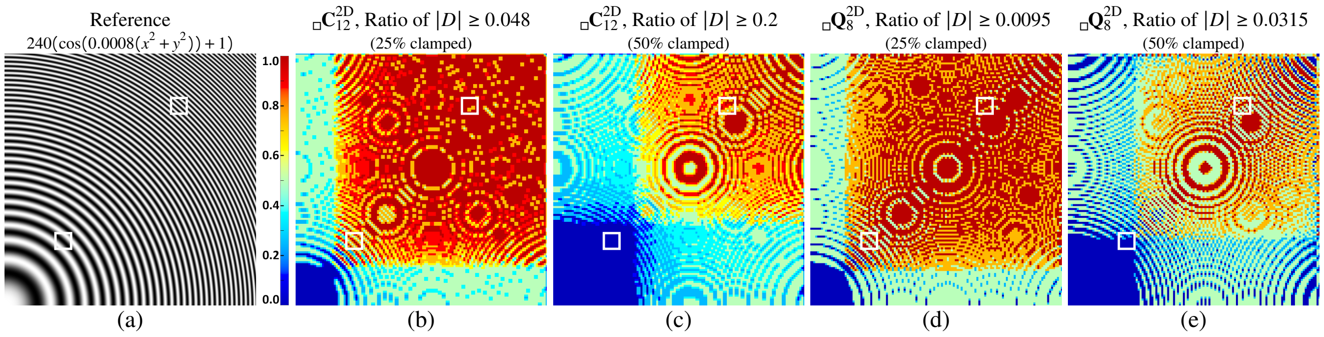


Figure 7: *We test with a sinusoidal function (the image spans a parameter range from* $(0,0)$ *to* $(1,1)$*). The reference image (left) is sampled using* 1024×1024 *resolution (the compared insets are highlighted), while our tested methods upsample a* 128×128 *resolution input to* 1024×1024 *resolution. We visualize the ratio of* **D***-terms in bicubic interpolation that are above a threshold (middle and right). Dark red stands for 1 (all 8* **D***-terms for the pixel are above the threshold), and dark blue stands for 0.*

and $w_2/(w_2+w_3)$. The weight for the linear interpolation that combines them is $(w_0+w_1)/(w_0+w_1+w_2+w_3)$.

Our method for supporting nonlinear tetrahedral interpolation extends this method. All **D**-terms to be computed using an unaligned $4 \times 4 \times 4$ array of texels. They are then fed into the bilinear operation block as barycentric operations, along with appropriate weights. In general, up to four **D**-terms can be retired per cycle through the bilinear operation logic.

6.2. Hardware Interpolation for Arbitrary Simplexes

Linear barycentric interpolation for triangles can be performed either in hardware or using code attached to the start of the fragment shader. Either way, it typically supports multiplying three barycentric coefficients times three parameter values.

D-terms for quadratic and cubic triangular interpolation can be generated e.g. in a geometry shader and then passed to the fragment shader. The interpolation can be performed using either logic or shader code that computes products of the barycentric terms and multiplies them in turn by the **D**-terms. In hardware this could be performed using multiple passes through the existing logic, eliminating the extra hardware multiplies where the **D**-terms are zero. Typically, GPU shader instructions support testing conditionals in parallel with ALU operations, so testing to see if **D**-terms can be eliminated does not need to reduce the performance of a shader code implementation, either.

Tetrahedral interpolation is performed in the same way, except with four barycentric terms instead of three. This is not needed in pixel shaders, but can be useful in vertex shaders (as illustrated in Section 7.5) as well as geometry shaders. As for the software implementation of unstructured triangles, shader code computes **D**-terms for unstructured tetrahedra and then performs the necessary multiplies and adds for non-zero sets of **D**-terms.

7. Evaluation

We demonstrate the effectiveness of our hardware adaptive high-order interpolation method in 5 different real-time rendering applications to cover all of rectangular, triangular, tetrahedral, and cuboidal cases. In each application, we compare the quality of our result to linear interpolation. We show how our result achieves similar quality as non-adaptive high-order interpolation by only computing high-order interpolation when necessary. By exploiting the sparseness of high-frequency information, our method can discard most high-order terms in many applications, making it run at a comparative performance or use similar amount of storage as linear interpolation. We also compare with the state-of-art high-order interpolation method for the application, if one exists and show how our method improves the performance while delivering similar quality.

For interpolations in grids, we use the number of BOPs computed as the performance metric for comparing different methods. For simplex mesh examples targetting software implementation, we report shader execution times on current hardware.

7.1. 2D Texture Filtering

First, we present results using a synthetic texture that contains a variety of frequencies, shown in Figure 7. Given a threshold $\mathbf{D}_{\min} = 0.048$ for bicubic interpolation $_{\square}\mathbf{C}_{12}^{2D}$, 25% of the **D**-terms are eliminated (Figure 7b-c). The higher frequency region shows higher ratio of **D**-terms with magnitude greater than the threshold. Below a certain frequency, all **D**-terms are below the threshold. Using $\mathbf{D}_{\min} = 0.2$, 50% of the **D**-terms are clamped. The observation is similar for biquadratic interpolation $_{\square}\mathbf{Q}_{8}^{2D}$ (Figure 7d-e). With $\mathbf{D}_{\min} = 0$ we turn off clamping and all **D**-terms are used.

A comparison between different interpolation methods using parts of the same synthetic texture can be found in Figure 8. Our bicubic interpolation $_{\square}\mathbf{C}_{12}^{2D}$ with $\mathbf{D}_{\min} = 0$ generates almost identical result as standard bicubic interpolation $_{\blacksquare}\mathbf{C}_{16}^{2D}$. This shows that the impact of omitting the 5th and 6th order terms is relatively minor in this example. When increasing $\mathbf{D}_{\min}$ to 0.048 to clamp 25% of the **D**-terms, the difference is hard to notice. With $\mathbf{D}_{\min} = 0.2$, the low-frequency inset begins to show patterns related to bilinear interpolation, because most **D**-terms are clamped. However, the high-frequency inset still looks unchanged, because higher-frequency regions contain more pixels with larger **D**-terms. Our $_{\square}\mathbf{Q}_{8}^{2D}$ biquadratic interpolation also produces almost identical result as the more expensive $_{\blacksquare}\mathbf{Q}_{9}^{2D}$ version. $\mathbf{D}_{\min}$ affects the results similarly to the bicubic case.

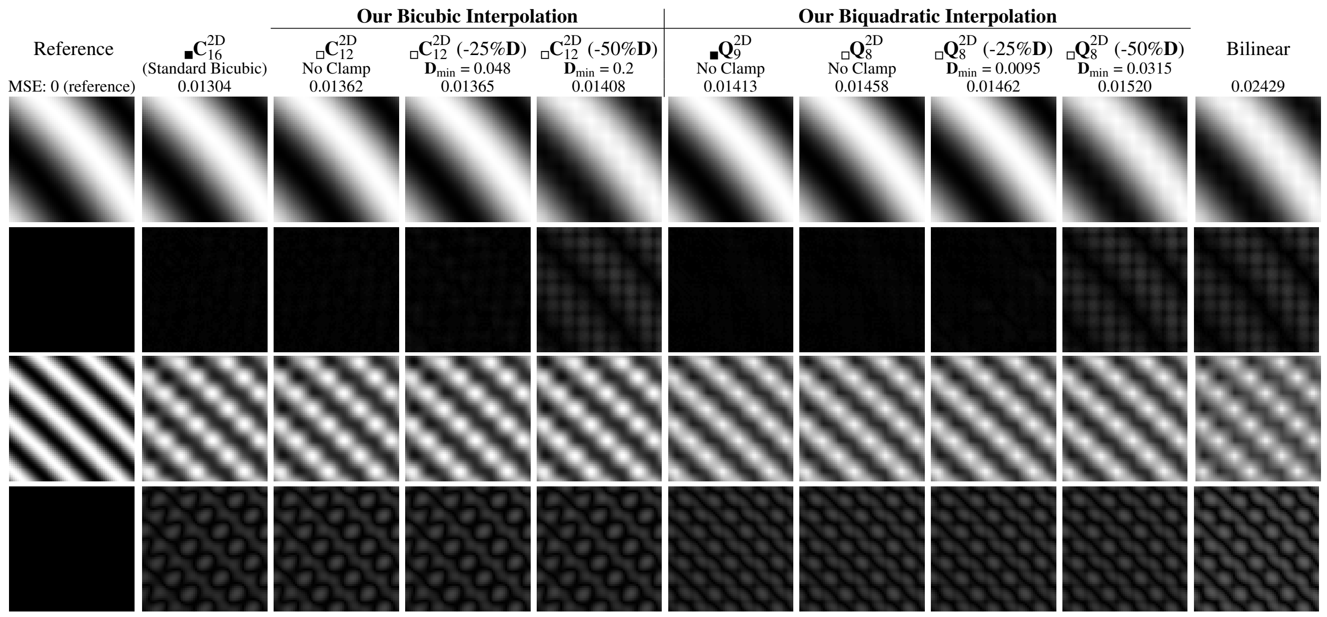

Our Bicubic Interpolation
Our Biquadratic Interpolation
Reference
MSE: 0 (reference)
${}_{\blacksquare}\mathbf{C}^{2D}_{16}$
(Standard Bicubic)
0.01304
${}_{\square}\mathbf{C}^{2D}_{12}$
No Clamp
0.01362
${}_{\square}\mathbf{C}^{2D}_{12}$ (-25%$\mathbf{D}$)
$\mathbf{D}_{min} = 0.048$
0.01365
${}_{\square}\mathbf{C}^{2D}_{12}$ (-50%$\mathbf{D}$)
$\mathbf{D}_{min} = 0.2$
0.01408
${}_{\blacksquare}\mathbf{Q}^{2D}_{9}$
No Clamp
0.01413
${}_{\square}\mathbf{Q}^{2D}_{8}$
No Clamp
0.01458
${}_{\square}\mathbf{Q}^{2D}_{8}$ (-25%$\mathbf{D}$)
$\mathbf{D}_{min} = 0.0095$
0.01462
${}_{\square}\mathbf{Q}^{2D}_{8}$ (-50%$\mathbf{D}$)
$\mathbf{D}_{min} = 0.0315$
0.01520
Bilinear
0.02429

Figure 8: *A comparison of different interpolation method for upsampling the sinusoidal function from* 128×128 *to* 1024×1024*. The first and third row: insets. The second and fourth row: 4 × and 1 × difference images with respect to the reference.*

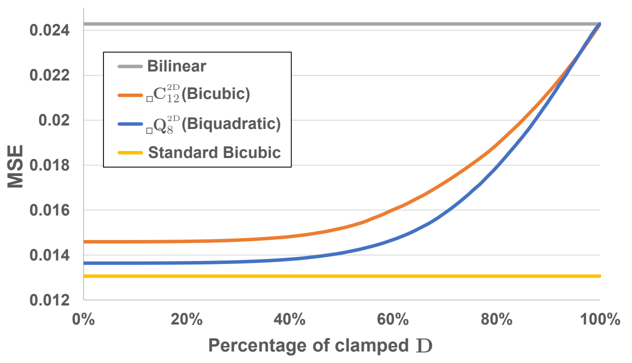


Figure 9: *Comparing MSE of different interpolation method using different* $\mathbf{D}_{min}$ *(only our bicubic and biquadratic interpolation is affected) for the sinusoidal function upsampling.*

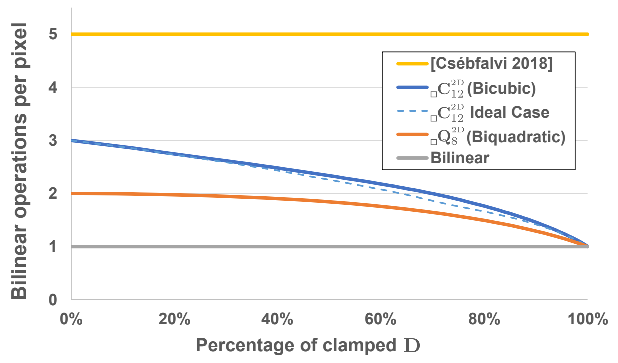


Figure 10: *Comparing average bilinear operations per pixel of different methods for the sinusoidal function upsampling.*

In Figure 9, we compare mean square error (MSE) of different interpolation methods. With $\mathbf{D}_{min} = 0$, our bicubic interpolation ${}_{\square}\mathbf{C}^{2D}_{12}$ produces a slightly higher MSE than the standard bicubic interpolation ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$, due to the omission of 5th and 6th order terms. Our biquadratic interpolation ${}_{\square}\mathbf{Q}^{2D}_{8}$ produces a slightly higher MSE. Nonetheless, MSE for all high-order interpolations is much lower than bilinear. As expected, MSE grows with increasing $\mathbf{D}_{min}$.

We compare the performance of our approach to Csébfalvi's method [Csé18], the most efficient implementation of standard bicubic interpolation on current GPU hardware. This method uses 5 bilinear operations (4 bilinear texture access on the GPU plus 1 bilinear operation for combining 4 terms with weights) to produce the same result as standard Catmull-Rom bicubic interpolation. The texture is modified by modulating the input texture with a checkboard pattern of 1 and −1 values.

In Figure 10, we visualize the average BOPs per pixel for each method. We see that even using $\mathbf{D}_{min} = 0$, our bicubic interpolation (3 per pixel) has fewer number of BOPs than Csébfalvi's method (5 per pixel). Using adaptive higher-order filtering with $\mathbf{D}_{min} > 0$, we can improve the performance further. The solid line for ${}_{\square}\mathbf{C}^{2D}_{12}$ shows the performance when groups of 4 $\mathbf{D}$-terms are formed in a predefined order and a BOP is skipped only when *all* $\mathbf{D}$-terms in a group are below the threshold. The dashed line shows the ideal case that dynamically groups the $\mathbf{D}$-terms for maximum performance. Our biquadratic interpolation starts at a cheaper cost of 2 BOPs at $\mathbf{D}_{min} = 0$ and decreases slowly with increasing $\mathbf{D}_{min}$.

To see how our bicubic rectangular interpolation works in prac-

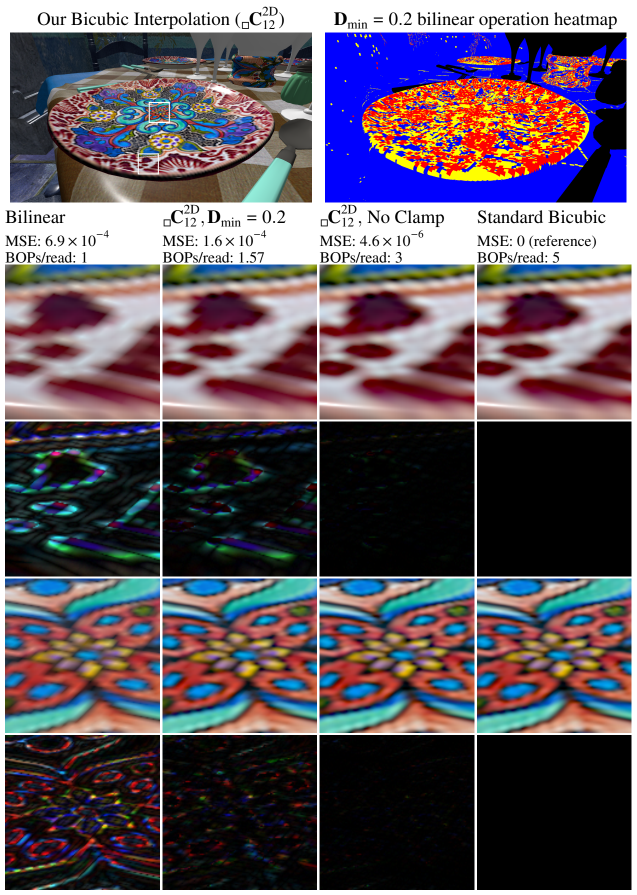

Our Bicubic Interpolation (${}_{\square}\mathbf{C}^{2D}_{12}$)
$\mathbf{D}_{\min}$ = 0.2 bilinear operation heatmap
Bilinear
MSE: 6.9×10^{-4}
BOPs/read: 1
${}_{\square}\mathbf{C}^{2D}_{12}$, $\mathbf{D}_{\min}$ = 0.2
MSE: 1.6×10^{-4}
BOPs/read: 1.57
${}_{\square}\mathbf{C}^{2D}_{12}$, No Clamp
MSE: 4.6×10^{-6}
BOPs/read: 3
Standard Bicubic
MSE: 0 (reference)
BOPs/read: 5

Figure 11: *Comparing texture magnification filtering quality of bilinear interpolation and our bicubic interpolation with no clamping and adaptive clamping with* $\mathbf{D}_{\min}$ = 0.2*. A bilinear operation heatmap at* $\mathbf{D}_{\min}$ = 0.2 *is provided (Red: 3 bops, Yellow: 2 bops, Blue: 1 bop). Third and fifth row:* 4× *difference images with respect to standard bicubic interpolation of the insets in the rows above.*

tice, we test the texture filtering quality in the San Miguel scene, shown in Figure 11. We observe that our ${}_{\square}\mathbf{C}^{2D}_{12}$ produces indistinguishable result to standard bicubic interpolation ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$. Setting $\mathbf{D}_{\min}$ = 0.2 produces visually very similar result with only 1.57 BOPs on average, in contrast to 3 BOPs without clamping, and 5 BOPs with Csébfalvi's method [Csé18] for standard bicubic interpolation.

7.2. Temporal Anti-aliasing

Another use case of bicubic interpolation is temporal anti-aliasing (TAA). TAA is widely used in today's game engines. It reuses subpixel samples accumulated in previous frames to achieve the effect of supersampling. This is done by jittering the camera to cover many sample positions in each pixel across several frames. When the camera or the scene objects are moving, motion vectors need to be calculated to fetch the pixel that corresponds to the same shading point in the history buffer (i.e. *reprojection*).

One problem of TAA is known as *resampling blur* [YLS20]. This

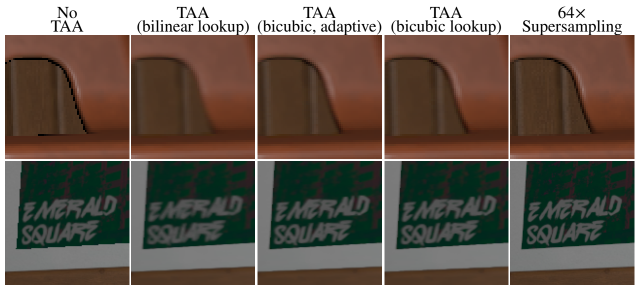


Figure 12: *A comparison of 3 different history buffer lookup methods. Note that jagged edges can be seen when not using TAA. A rendering using* 64× *supersampling is provided as the ground truth. Note that bicubic lookup avoids the blurring of object edges and texture details in as can be seen in result using bilinear lookup. Bicubic adaptive lookup using* $\mathbf{D}_{\min}$ = 0.003 *produces highly similar result to using no clamping.*

is due to reprojected positions landing at fractional pixel locations, which requires interpolation to gather colors from nearby pixels. As can be seen in Figure 12, using bicubic interpolation reduces resampling blur as compared to bilinear interpolation.

Using our method with hardware modification, it is possible to bring down the cost of bicubic interpolation to a level close to bilinear interpolation, while preserving most of the quality, as shown in Figure 13. Without clamping, our ${}_{\square}\mathbf{C}^{2D}_{12}$ brings down the BOPs per pixel to 3 to produce visually identical results standard bicubic interpolation. Using $\mathbf{D}_{\min}$ = 0.003, more than 80% of bilinear operations related to higher-order terms can be skipped. The result is that only 39% more bilinear operations than pure bilinear interpolation are executed to achieve a result almost identical to full bicubic interpolation. The bilinear operation heatmap in Figure 13 shows that these discarded bilinear operations are mainly in regions with slowly varying color, while bicubic interpolation are preserved at high frequency regions like object edges and texture gradient edges, which suffers more from resampling blur. Even though these results use many state-of-art TAA improvement techniques provided by the Falcor engine [Kar14], including neighborhood clamping and adaptive blending, they are insufficient for removing the resampling blur caused by bilinear interpolation. Bicubic interpolation, however, visibly improves the results.

7.3. Volume Rendering

We use volume rendering as an application to test our tricubic interpolation ${}_{\square}\mathbf{C}^{3D}_{32}$. Our test scene contains a volumetric teapot with isotropic scattering and a directional light shown in Figure 14. The density data of the volume is defined in a 3D grid. We use ray marching to compute the radiance reaching the camera. At each ray marching step, a density value is queried for computing the extinction coefficient. The lighting is gathered from a directional light with the visibility along the shadow ray computed via ray marching. We compare our performance to Csébfalvi's tricubic volume density filtering method [Csé19]. Note that the Csébfalvi's method uses 7 trilinear taps on the current hardware, which is equivalent to 14 bilinear operations. In comparison, our method only uses 8

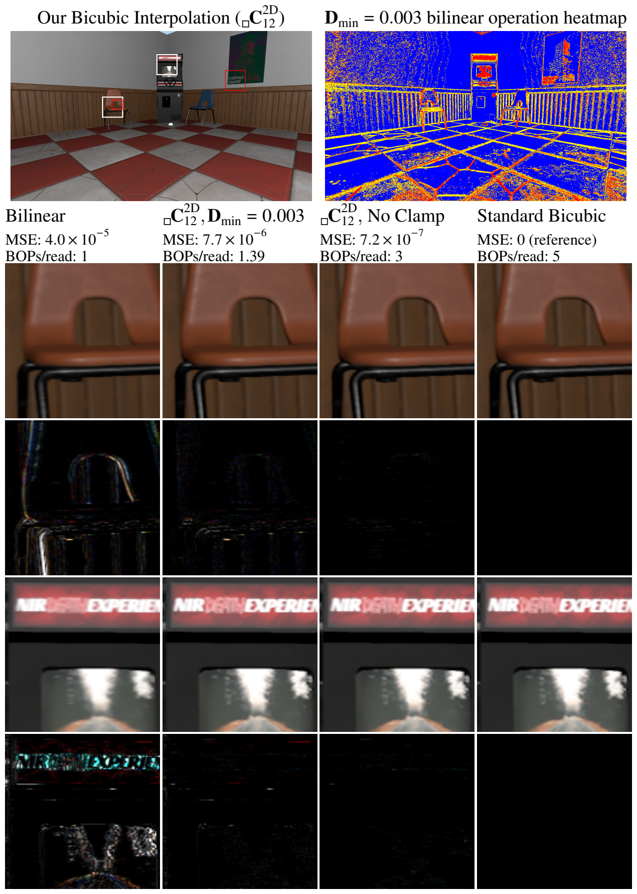

Our Bicubic Interpolation (${}_{\square}\mathbf{C}^{2D}_{12}$)
$\mathbf{D}_{\min}$ = 0.003 bilinear operation heatmap
Bilinear
MSE: 4.0×10^{-5}
BOPs/read: 1
${}_{\square}\mathbf{C}^{2D}_{12}, \mathbf{D}_{\min} = 0.003$
MSE: 7.7×10^{-6}
BOPs/read: 1.39
${}_{\square}\mathbf{C}^{2D}_{12}$, No Clamp
MSE: 7.2×10^{-7}
BOPs/read: 3
Standard Bicubic
MSE: 0 (reference)
BOPs/read: 5

Figure 13: *Comparing TAA results using bilinear history lookup and our bicubic history lookup with and without clamping. The results are obtained by repeatly shaking the camera horizontally and freezing the rendering. A bilinear operation heatmap at* $\mathbf{D}_{\min}$ = 0.003 *is provided (Red: 3 bops, Yellow: 2 bops, Blue: 1 bop). Third and fifth row:* 8× *difference images in comparison to the standard bicubic interpolation.*

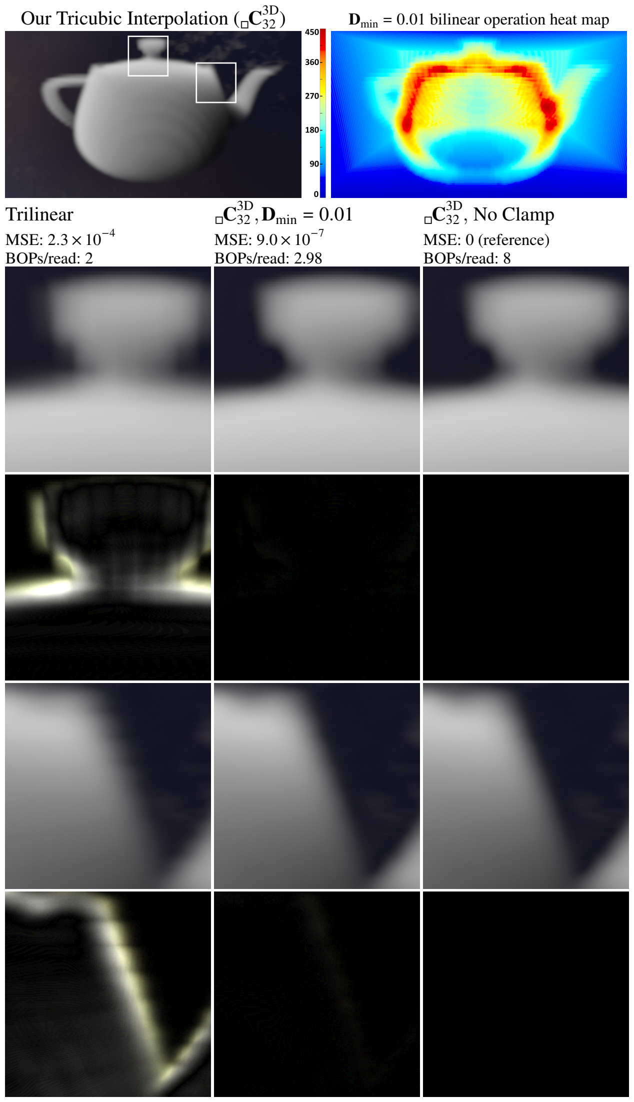

Our Tricubic Interpolation (${}_{\square}\mathbf{C}^{3D}_{32}$)
$\mathbf{D}_{\min}$ = 0.01 bilinear operation heat map
450
360
270
180
90
0
Trilinear
MSE: 2.3×10^{-4}
BOPs/read: 2
${}_{\square}\mathbf{C}^{3D}_{32}, \mathbf{D}_{\min} = 0.01$
MSE: 9.0×10^{-7}
BOPs/read: 2.98
${}_{\square}\mathbf{C}^{3D}_{32}$, No Clamp
MSE: 0 (reference)
BOPs/read: 8

Figure 14: *Comparison of trilinear and our tribubic interpolation with and without clamping. The heatmap shows the number of bilinear operations per pixel. For clamping with* $\mathbf{D}_{\min}$ = 0.01*, only 16.5% bilinear operations related to high-order interpolation are executed. Only 11% of total* **D***-terms are non-zero. Third and fifth rows: 8× differences from tricubic interpolation with no clamping.*

bilinear operations. Note that our ${}_{\square}\mathbf{C}^{3D}_{32}$ function is mathematically identical to Csébfalvi's method, but our formulation needs fewer bilinear operations.

The performance advantage of our method is improved with adaptive high-order interpolation. By using a low threshold of $\mathbf{D}_{\min} = 0.01$, our method reduces more than 80% of bilinear interpolations related to high-order interpolation, most of which are at the center region of the teapot (with almost uniform density) and the region outside the teapot (with zero density). By exploiting the sparseness of high frequency signal, our adaptive tricubic interpolation only uses about 1.5× the bilinear operation required for trilinear interpolation, while producing visually identical quality as full tricubic interpolation using ${}_{\square}\mathbf{C}^{3D}_{32}$ (4× more bilinear operation than trilinear).

7.4. Precomputed Radiance Transfer

In precomputed radiance transfer (PRT) [SKS02], transfer vectors and transfer matrices are precomputed for all vertices in spherical harmonic (SH) basis such that given an incoming radiance field expressed in SH basis, the exitant radiance field can be easily obtained using vector dot product (for diffuse reflection) or matrix-vector product (for specular refleciton). This allows real-time, noise-free, high-fidelity self-shadowing and self-reflection of a object in any dynamic, low frequency lighting environments (e.g. a rotating environment map).

We limit our evaluation to 1-bounce diffuse reflection with self-

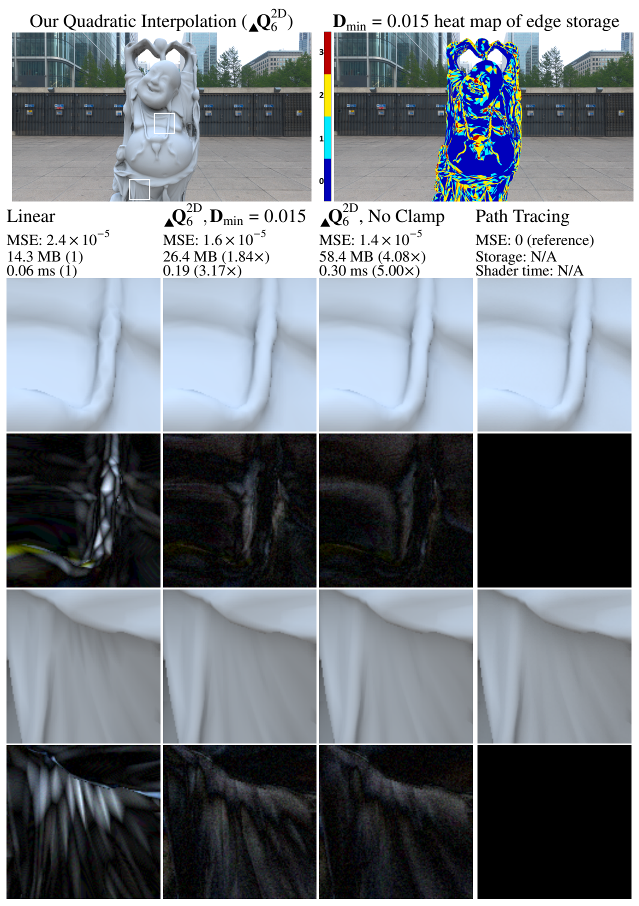

Our Quadratic Interpolation (${}_{\blacktriangle}\mathbf{Q}_6^{2D}$)
$\mathbf{D}_{\min}$ = 0.015 heat map of edge storage
3
2
1
0
Linear
MSE: 2.4×10^{-5}
14.3 MB (1)
0.06 ms (1)
${}_{\blacktriangle}\mathbf{Q}_6^{2D}, \mathbf{D}_{\min} = 0.015$
MSE: 1.6×10^{-5}
26.4 MB (1.84×)
0.19 (3.17×)
${}_{\blacktriangle}\mathbf{Q}_6^{2D}$, No Clamp
MSE: 1.4×10^{-5}
58.4 MB (4.08×)
0.30 ms (5.00×)
Path Tracing
MSE: 0 (reference)
Storage: N/A
Shader time: N/A

Figure 15: *Comparison of linear triangular interpolation of transfer vectors to our quadratic triangular interpolation with and without clamping for precomputed radiance transfer. With* $\mathbf{D}_{\min}$ = 0.015*, only 25% of edge data are stored and 23% three-multiplies related to quadratic terms are executed in the fragment shader, as compared to no clamping. The heatmap visualizes the number of edges with storage in the corresponding triangle of the pixel. The third and fifth rows: 4× differences from path tracing reference.*

shadowing, using the Happy Buddha model, shown in Figure 15. We precompute a (RGB) transfer vector using the first 5 SH bands ($5^2 \times 3 = 75$ floats) for each vertex. To enable quadratic interpolation, we also precompute a transfer vector (in fact, the difference terms) at the mid point of each edge. We traverse 4096 shadow rays to precompute a shadowed diffuse transfer vector in SH basis. When using linear interpolation, dot product between transfer vector and the lighting vector is computed in vertex shader, and the resulting color is interpolated automatically for the fragment shader. When using quadratic interpolation, we perform manual interpolation of quadratic **D**-terms (of ${}_{\blacktriangle}\mathbf{Q}_6^{2D}$) in the fragment shader and dot product with lighting vector, then add the result on top of the color from hardware linear interpolation. Thus, only the high-order terms are handled in software. Notice that quadratic interpolation avoids the faceted appearance produced by linear interpolation and provides closer results to the path tracing reference (Figure 15).

Our adaptive quadratic interpolation both reduces the edge storage and the computation in the fragment shader. Figure 15 shows that using $\mathbf{D}_{\min}$ = 0.015, edge storage is eliminated in relatively flat surface regions such that only 25% of the edge storage remains and the quality is almost identical to full quadratic interpolation. This makes the total edge and vertex storage for adaptive quadratic interpolation less than twice the vertex storage for linear interpolation. This is a significant reduction compared to non-adaptive quadratic interpolation that requires more than four times total storage than linear interpolation. Even only testing the performance on the current hardware, skipping reading and computing quadratic terms adaptively in the fragment shader translates to actual performance improvement. The shader execution time on the GPU is reduced from 0.30 ms to 0.19 ms when using adaptive interpolation, making it closer to the 0.06 ms required by linear interpolation. With the proposed hardware support, edge colors can be computed in geometry shader and the fragment color would be automatically interpolated. Without clamping, this would double the three-multiplier passes required by linear interpolation. With $\mathbf{D}_{\min}$ = 0.015, only 1.43× more passes are required (since only 43% of the triangles have non-zero edge storages).

7.5. Embedded Deformation

FEM simulation on high-poly tetrahedral mesh (tet-mesh) can be expensive. A popular way to overcome this problem is *embedded deformation* that simulates on a low-poly tet-mesh and interpolates the deformation to produce the vertex position for the high-poly triangular mesh that is used for rendering.

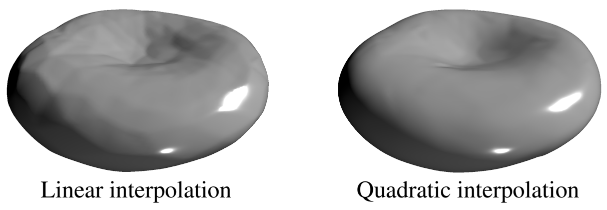


Figure 16: *A comparison of linear interpolation and our quadratic interpolation for embedded deformation of a sphere. Edge mid points generated using vertex gradients computed using the method in Phong Deformation [Jam20].*

Linear interpolation with embedded deformation produces undesirable faceted artifacts when the deformation is large, as can be seen in Figure 16. Phong deformation [Jam20] solves this problem by computing per-vertex gradients from neighboring tetrahedra and blending them with per-tetrahedron gradients used for linear interpolation to provide a result with a reduced truncation error than linear interpolation.

Figure 17 shows a more complex example. Our quadratic tetrahedral interpolation ${}_{\blacktriangle}\mathbf{Q}_{10}^{3D}$ produces identical results to Phong deformation, but requires more storage, as shown in Table 1. Phong deformation stores a 3×3 matrix (vertex gradient) at each vertex, while ${}_{\blacktriangle}\mathbf{Q}_{10}^{3D}$ stores 3D vector positions on each edge of the tet mesh. Although each vertex stores three times more data than each edge, a tetrahedral mesh can have 7× as many edges as vertices, requiring

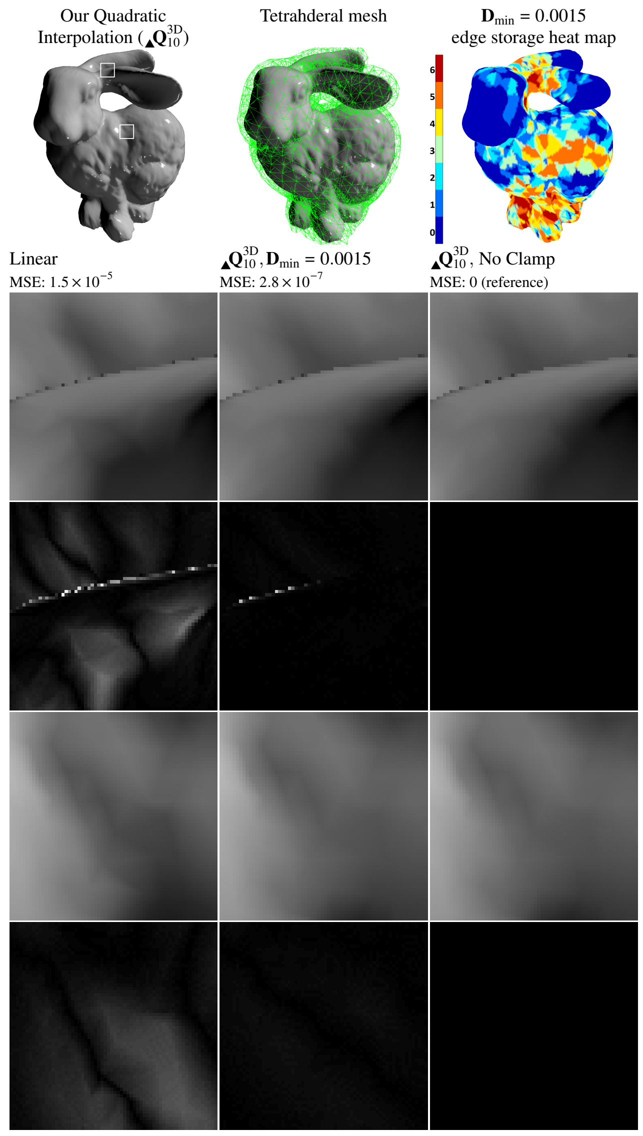

Our Quadratic Interpolation ($_{\blacktriangle}\mathbf{Q}^{3D}_{10}$)
Tetrahderal mesh
$\mathbf{D}_{\min} = 0.0015$ edge storage heat map
6
5
4
3
2
1
0
Linear
MSE: 1.5×10^{-5}
$_{\blacktriangle}\mathbf{Q}^{3D}_{10}$, $\mathbf{D}_{\min} = 0.0015$
MSE: 2.8×10^{-7}
$_{\blacktriangle}\mathbf{Q}^{3D}_{10}$, No Clamp
MSE: 0 (reference)

Figure 17: *Embedded deformation of a mesh (69664 vertices) using a tet-mesh with only 1654 vertices. We compare the quality of linear tetrahedral interpolation to our quadratic interpolation with and without clamping. Notice that quadratic interpolation reduces the faceted appearance (looks like darkend "dents" under lighting). The third and fifth rows: 4× differences w.r.t to No Clamp. The heat map visualizes the density of edge storage (number of edges in the corresponding tetrahedra that are not clamped).*

more storage when using $_{\blacktriangle}\mathbf{Q}^{3D}_{10}$ [Jam20]. However, we show that using $_{\blacktriangle}\mathbf{Q}^{3D}_{10}$, the shader execution time for embedded deformation can be reduced, as compared to Phong deformation.

Frames from an animation sequence are shown in Figure 18. In our implementation, we precompute and store the $\mathbf{D}$-terms on edges for a pre-generated deformation sequence of the enclosing tet-mesh and invoke a vertex shader each frame to deform the embedded triangular mesh. Such edge value precomputation would need to be

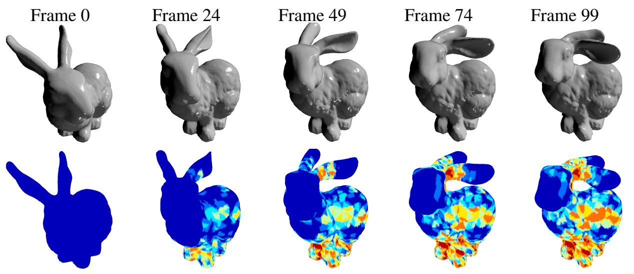


Figure 18: *A visualization of the entire deformation of 100 frames. The tet-meshes are generated from FEM simulation in Houdini 18.5. Note that less edge data are clamped as the amount of deformation increases (Frame 0 is the rest pose). The color legend is the same as in Figure 17.*

Table 1: *Average shader execution times and total storage cost of interpolation methods for embedded deformation shown in Figure 18.*

Method	**Time**	**Storage**
Linear Interpolation	109 μs	2.0 MB
Our $_{\blacktriangle}\mathbf{Q}^{3D}_{10}$ (No clamp)	141 μs	12.7 MB
Our $_{\blacktriangle}\mathbf{Q}^{3D}_{10}$ ($\mathbf{D}_{\min} = 0.0015$)	147 μs	9.8 MB
Phong Deformation	166 μs	7.8 MB

done at each frame if the deformation is simulated on the fly. This introduces an additional step, as compared to Phong deformation. In our evaluation, however, we do not consider the additional precomputation time and only compare the shader execution time with precomputed values for the entire sequence. Since we have shorter vertex shader execution time, we expect that when the embedded triangular mesh is much more complex than the enclosing tet-mesh, the vertex shader cost will be much higher than computing and storing the edge data for the tet-mesh, making our method faster for the on-the-fly deformation.

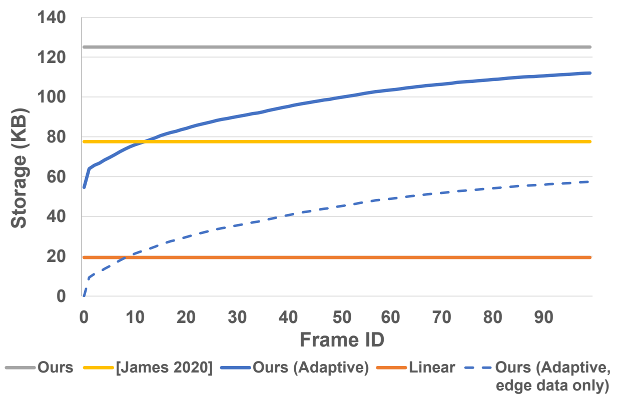


Figure 19: *Per-frame storage of different interpolation methods for the test scene in Figure 18. Ours (Adaptive) uses* $\mathbf{D}_{\min} = 0.0015$.

Figure 19 shows the storage costs of different methods for the sequence in Figure 18. As expected, our quadratic interpolation with no clamping requires more storage than Phong deformation on average. We can reduce the storage using adaptive interpolation by setting $\mathbf{D}_{\min} = 0.0015$ to obtain a result almost identical to no clamping. This reduces the storage by more than 50% in the best case, though we still require more storage than Phong deformation. In Figure 19, we also include a dashed line that shows the storage of only the edge data of our method. Notice that at Frame 0, no edge data is required, because all **D**-terms are zero. Adaptive interpolation also needs an additional buffer to keep the offsets in the compressed edge data for each edge. This results in additional storage and shader execution times.

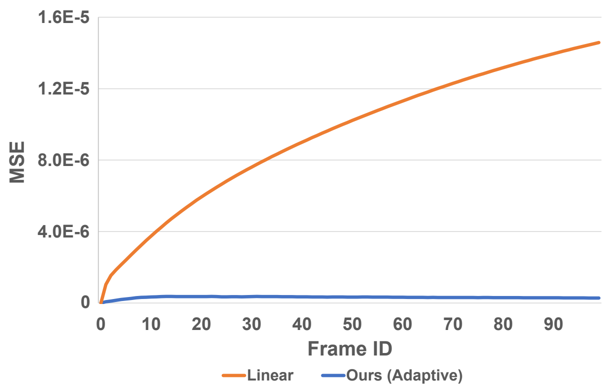


Figure 20: *A comparison between linear interpolation and our adaptive quadratic interpolation using* $\mathbf{D}_{\min}$ *in terms of mean square error (MSE) of the triangular mesh vertex positions with respect to the full quadratic interpolation positionin all 100 frames. The test scene is shown in Figure 18.*

The interpolation errors for linear and our adaptive interpolation for the animation squence in Figure 18 are shown in Figure 20. In comparison to linear interpolation, which shows increasingly larger error as the deformation progresses, our quadratic adaptive interpolation shows a stable amount of error that is also much smaller than the average error of linear interpolation.

8. Conclusions

We have presented high-order interpolation for grids and simplexes. We have introduced a unified, computationally-efficient mathematical formulation which adds higher-order terms on top of linear interpolation to achieve quadratic and cubic interpolation. Our formulation allows adaptively skipping computation of high-order terms. We propose relatively minor modifications to existing graphics hardware to support our adaptive high-order interpolation. Our proposed hardware implementation can bring the cost of adaptive high-order interpolation close to linear interpolation, dramatically improving the visual quality of a wide range of real-time graphics applications at low cost.

References

[ASS*01] ABBAS, AM, SZIRMAY-KALOS, L, SZIJARTO, G, et al. "Quadratic interpolation in hardware rendering". *Spring Conference of Computer Graphics*. 2001 3.

[BA08] BOUBEKEUR, TAMY and ALEXA, MARC. "Phong Tessellation". *ACM Trans. Graph.* 27.5 (Dec. 2008). ISSN: 0730-0301 3.

[BBGB20] BOUKHTACHE, SEYFEDDINE, BLAYSAT, BENOIT, GRÉDIAC, MICHEL, and BERRY, FRANCOIS. "Alternatives to Bicubic Interpolation Considering FPGA Hardware Resource Consumption". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020) 2.

[BC14] BARGTEIL, ADAM W and COHEN, ELAINE. "Animation of deformable bodies with quadratic Bézier finite elements". *ACM Transactions on Graphics (TOG)* 33.3 (2014), 1–10 3.

[Bjo04] BJORKE, KEVIN. "High-quality filtering". *GPU Gems* 2 (2004), 391–415 2.

[Bro99] BROWN, RUSS. *Modeling specular highlights using Bézier triangles*. Sun Microsystems, Inc., 1999 2.

[Csé18] CSÉBFALVI, BALÁZS. "Fast Catmull-Rom Spline Interpolation for High-Quality Texture Sampling". *Computer Graphics Forum*. Vol. 37. 2. Wiley Online Library. 2018, 455–462 1, 2, 10, 11.

[Csé19] CSÉBFALVI, BALÁZS. "Beyond trilinear interpolation: higher quality for free". *ACM Transactions on Graphics (TOG)* 38.4 (2019), 1–8 2, 11.

[Far93] FARIN, GERALD. "Chapter 18 - Bézier Triangles". *Curves and Surfaces for Computer-Aided Geometric Design (Third Edition)*. Ed. by FARIN, GERALD. Third Edition. Boston: Academic Press, 1993, 321–351. ISBN: 978-0-12-249052-1 2.

[GNSS14] GOUR, PRANAV NARAYAN, NARUMANCHI, SUJAY, SAURAV, SUMEET, and SINGH, SANJAY. "Hardware accelerator for real-time image resizing". *18th International Symposium on VLSI Design and Test*. IEEE. 2014, 1–6 2.

[Jam20] JAMES, DOUG L. "Phong deformation: a better C 0 interpolant for embedded deformation". *ACM Transactions on Graphics (TOG)* 39.4 (2020), 56–1 3, 13, 14.

[Kar14] KARIS, BRIAN. "High Quality Temporal Supersampling". *ACM SIGGRAPH Courses: Advances in Real-Time Rendering in Games (2014)*. SIGGRAPH '14. New York, NY, USA: Association for Computing Machinery, 2014 11.

[Key81] KEYS, ROBERT. "Cubic convolution interpolation for digital image processing". *IEEE transactions on acoustics, speech, and signal processing* 29.6 (1981), 1153–1160 2.

[KNPH95] KASSON, JAMES M, NIN, SIGFREDO I, PLOUFFE, WIL, and HAFNER, JAMES LEE. "Performing color space conversions with three-dimensional linear interpolation". *Journal of Electronic Imaging* 4.3 (1995), 226–250 8.

[LSC*08] LIN, CHUNG-CHI, SHEU, MING-HWA, CHIANG, HUANN-KENG, et al. "The efficient VLSI design of BI-CUBIC convolution interpolation for digital image processing". *2008 IEEE International Symposium on Circuits and Systems*. IEEE. 2008, 480–483 2.

[LSC*10] LIN, CHUNG-CHI, SHEU, MING-HWA, CHIANG, HUANN-KENG, et al. "An Efficient Architecture of Extended Linear Interpolation for Image Processing." *J. Inf. Sci. Eng.* 26.2 (2010), 631–648 2.

[Map06] MAPLESON, IAN. *Infinite Reality Technical Report*. 2006. URL: http://www.sgidepot.co.uk/onyx/IR_techreport.pdf 2.

[MBDM97] MONTRYM, JOHN S, BAUM, DANIEL R, DIGNAM, DAVID L, and MIGDAL, CHRISTOPHER J. "InfiniteReality: A real-time graphics system". *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, 293–302 2.

[MDSB03] MEYER, MARK, DESBRUN, MATHIEU, SCHRÖDER, PETER, and BARR, ALAN H. "Discrete differential-geometry operators for triangulated 2-manifolds". *Visualization and mathematics III*. Springer, 2003, 35–57 8.

[MH15] MAHAJAN, SHRUTI H and HARPALE, VARSHA K. "Adaptive and non-adaptive image interpolation techniques". *2015 International Conference on Computing Communication Control and Automation*. IEEE. 2015, 772–775 2.

[ML94] MARSCHNER, STEPHEN R and LOBB, RICHARD J. "An evaluation of reconstruction filters for volume rendering". *Proceedings Visualization'94*. IEEE. 1994, 100–107 2.

[MMMY97] Moller, Torsten, Machiraju, Raghu, Mueller, Klaus, and Yagel, Roni. "Evaluation and design of filters using a Taylor series expansion". *IEEE transactions on Visualization and Computer Graphics* 3.2 (1997), 184–199 2.

[MN88] Mitchell, Don P and Netravali, Arun N. "Reconstruction filters in computer-graphics". *ACM Siggraph Computer Graphics* 22.4 (1988), 221–228 2.

[MSY19] Mallett, Ian, Seiler, Larry, and Yuksel, Cem. "Patch textures: Hardware implementation of mesh colors". (2019) 6, 8.

[NA05] Nuño-Maganda, Marco Aurelio and Arias-Estrada, Miguel O. "Real-time FPGA-based architecture for bicubic interpolation: an application for digital image scaling". *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*. IEEE. 2005, 8–pp 2.

[Pho75] Phong, Bui Tuong. "Illumination for Computer Generated Pictures". *Communications of the ACM* 18.6 (June 1975), 311–317. ISSN: 0001-0782 2.

[Sei98] Seiler, Larry. "Quadratic interpolation for near-Phong quality shading". *ACM SIGGRAPH 98 Conference Abstracts and Applications*. 1998, 268 3.

[SH05] Sigg, Christian and Hadwiger, Markus. "Fast third-order texture filtering". *GPU gems* 2 (2005), 313–329 2.

[SKH16] Sanaullah, Ahmed, Khoshparvar, Arash, and Herbordt, Martin C. "FPGA-Accelerated Particle-Grid Mapping". *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2016, 192–195 2.

[SKS02] Sloan, Peter-Pike, Kautz, Jan, and Snyder, John. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments". *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, 527–536 12.

[VPBM01] Vlachos, Alex, Peters, Jörg, Boyd, Chas, and Mitchell, Jason L. "Curved PN triangles". *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, 159–166 3, 8.

[WDLY11] Wang, Xiang, Ding, Yong, Liu, Ming-yu, and Yan, Xiao-lang. "Efficient implementation of a cubic-convolution based image scaling engine". *Journal of Zhejiang University SCIENCE C* 12.9 (2011), 743–753 2.

[YKH10] Yuksel, Cem, Keyser, John, and House, Donald H. "Mesh colors". *ACM Transactions on Graphics* 29.2 (2010), 15:1–15:11. ISSN: 0730-0301 8.

[YLS20] Yang, Lei, Liu, Shiqiu, and Salvi, Marco. "A survey of temporal antialiasing techniques". *Computer Graphics Forum*. Vol. 39. 2. Wiley Online Library. 2020, 607–621 2, 11.

[Yuk17] Yuksel, Cem. "Mesh color textures". *Proceedings of High Performance Graphics*. 2017, 1–11 8.

[ZLZ*10] Zhang, Yunshan, Li, Yuhui, Zhen, Jie, et al. "The hardware realization of the bicubic interpolation enlargement algorithm based on FPGA". *2010 Third International Symposium on Information Processing*. IEEE. 2010, 277–281 2.

[ZTZ05] Zienkiewicz, Olek C, Taylor, Robert L, and Zhu, Jian Z. *The finite element method: its basis and fundamentals*. Elsevier, 2005 2, 3.

Appendix A: L1 Cache Banking for High Order Filtering

Consider 2D cubic interpolations ${}_{\square}\mathbf{C}^{2D}_{12}$ and ${}_{\blacksquare}\mathbf{C}^{2D}_{16}$. They require reading texels in an unaligned 4×4 block in parallel so that *all* **D**-terms can be computed in one clock cycle. That in turn permits the filter operation to be completed in one clock if all **D**-terms are below the threshold $\mathbf{D}_{\min}$, so that the operation reduces to linear interpolation.

Note that ${}_{\square}\mathbf{C}^{2D}_{12}$ does not need the four corner texels of the 4×4 block, so a small amount of power can be saved by not reading from those banks. 4×4 banks are still required since the texels emerge from the banks in rotated order, as described at the end of Section 4.2. Therefore, the unused texels could come from any of the 16 banks, depending on the alignment of the 4×4 block.

Next, consider 3D cubic interpolations ${}_{\square}\mathbf{C}^{3D}_{32}$ and ${}_{\blacksquare}\mathbf{C}^{3D}_{64}$. They require reading texels in an unaligned $4\times4\times4$ block (though ${}_{\square}\mathbf{C}^{3D}_{32}$ does not use the corner values). However, if all **D**-terms are below $\mathbf{D}_{\min}$ and the filter operation reduces to trilinear interpolation, they require two BOP cycles to process $2\times2\times2$ texels. Therefore, it is only necessary to be able to read $4\times4\times2$ texels in parallel on each of two clocks to allow cubic cuboid interpolation to perform at the same speed as trilinear interpolation.

Biquadratic and triquadratic filtering are also supported using unaligned 4×4 or $4\times4\times4$ arrays of texels. The mid-edge values are computed from the larger array. So, although the quadratic filtering modes require fewer **D**-terms, they require accessing the same number of texels.

Therefore, to accommodate peak performance we must access up to 32 texels per clock from the texel L1 cache. The left side of Figure 4 illustrates this. The 32 banks are interleaved based on the low order two bits of each of the U and V indices and the low order bit of the slice number for the 3D texel array. Then, two accesses provide the unaligned $4\times4\times4$ of texels needed to perform tricubic interpolation.

It is not necessary for an implementation to support 32 banks of texel L1 cache in order to support tricubic interpolation. E.g. if only 16 banks were supported, tricubic interpolation would require a minimum of four cycles instead of two. This would also reduce the cost of computing the **D**-terms, since only one quarter of them would need to be computed per clock.

Similarly, an implementation could choose to provide only 8 banks. This would cause bicubic interpolation to take a minimum of two cycles because it takes two accesses to read an unaligned 4×4 of texels from the texel L1 cache. Therefore, the cost of increasing the number of texel L1 cache banks can be traded off against the desired performance for higher order filtering.