EUROGRAPHICS 2018 / D. Gutierrez and A. Sheffer
(Guest Editors)

Volume 37 (2018), Number 2

Approximate Program Smoothing Using Mean-Variance Statistics, with Application to Procedural Shader Bandlimiting

Y. Yang[1] and C. Barnes[1,2]

[1]University of Virginia, USA [2]Adobe Research

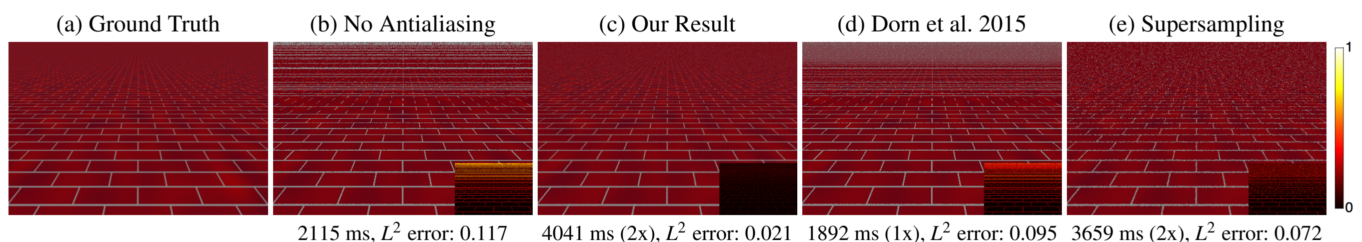


Figure 1: *Our paper gives a novel compiler framework for smoothing programs. Here we show how our smoothing framework can be applied to bandlimiting (antialiasing) procedural shader programs. In (a) is the ground truth result for a brick shader, estimated by using 1000 samples; (b) is the aliased result due to naively evaluating the original shader program; (c) is our result; (d) is the result of previous work; and (e) is supersampling, chosen to use comparable run-time as our result. The L^2 errors are reported in sRGB color space, with the inset heatmap depicting per-pixel L^2 error. Our result has significantly less error, noise, and aliasing than other approaches.*

Abstract

We introduce a general method to approximate the convolution of a program with a Gaussian kernel. This results in the program being smoothed. Our compiler framework models intermediate values in the program as random variables, by using mean and variance statistics. We decompose the input program into atomic parts and relate the statistics of the different parts of the smoothed program. We give several approximate smoothing rules that can be used for the parts of the program. These include an improved variant of Dorn et al. [DBLW15], a novel adaptive Gaussian approximation, Monte Carlo sampling, and compactly supported kernels. Our adaptive Gaussian approximation handles multivariate Gaussian distributed inputs, gives exact results for a larger class of programs than previous work, and is accurate to the second order in the standard deviation of the kernel for programs with certain analytic properties. Because each expression in the program can have multiple approximation choices, we use a genetic search to automatically select the best approximations. We apply this framework to the problem of automatically bandlimiting procedural shader programs. We evaluate our method on a variety of geometries and complex shaders, including shaders with parallax mapping, animation, and spatially varying statistics. The resulting smoothed shader programs outperform previous approaches both numerically and aesthetically.

CCS Concepts

•Software and its engineering → Compilers; •Computing methodologies → Rendering;

1. Introduction

In many contexts, functions that have aliasing or noise could be viewed as undesirable. In this paper, we develop a general compiler-driven machinery to approximately smooth arbitrary programs, and thereby suppress aliasing or noise. We then apply this machinery to bandlimit procedural shader programs. In order to motivate our approach concretely by an application, we first discuss how procedural shaders may be bandlimited, and then return to our smoothing compiler.

Procedural shaders are important in rendering systems, because they can be used to flexibly specify material appearance in virtual scenes [AMHH08]. In this work we focus on purely procedural shaders that do not contain texture lookups or other references to buffers. One visual error that can appear in procedural shaders is *aliasing*. *Aliasing* artifacts occur when the sampling rate is below the Nyquist limit [Cro77]. There are two more conventional approaches used to reduce such aliasing: supersampling and prefiltering. We discuss these before discussing our smoothing compiler.

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

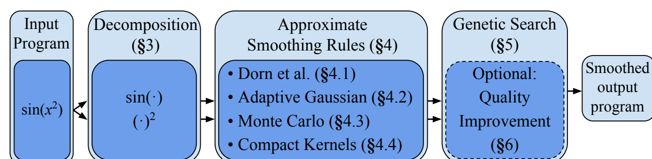


Figure 2: *An overview of our compiler framework. The components of our framework are discussed in the introduction.*

Supersampling increases the spatial sampling rate, so that the output value for each pixel is based on multiple samples. The sampling rate can be uniform across the image. The sampling rate can also be chosen adaptively based on measurements such as local contrast [DW, Mita, HJW*08, Mitb]. This approach in the limit recovers the ground truth image, but can be time-consuming due to requiring multiple samples per pixel.

Prefiltering typically stores precomputed integrals in mipmaps [Wil83] or summed area tables [Cro84]. This approach offers the benefit of accurate solutions with a constant number of operations, provided that the shading function can be spatially tiled or otherwise represented on a compact domain. However, in practice many interesting shaders do not tile, so this limits the applicability of this method. Further, prefiltering increases storage requirements and may replace inexpensive computations with more expensive memory accesses. This approach is not practical for functions of more than two or three variables because memory costs scale exponentially.

An alternative strategy is to construct a bandlimited variant of the shading function by symbolic integration. This can be expressed by convolving the shading function with a low-pass filter [NRS82]. Exact analytic band-limited formulas are known for some specialized functions such as noise functions [LLDD09]. In most cases, however, the shader developer must manually calculate the convolution integral. However, frequently the integrals cannot be solved in closed form, which limits this strategy.

We take a different approach than most previous work, by using a compiler framework to smooth an input program. We show an overview of this process in Figure 2. Our goal is to smooth an arbitrary input function represented as a program, by approximately convolving it with a Gaussian filter. This convolution could be multidimensional: for shader programs, the dimension is typically 2D for spatial coordinates. We would also like the output program to be as efficient as possible. The compiler takes the program as input, and decomposes it into atomic parts whose bandlimited solutions are easier to obtain (Section 3). We then relate the statistics of the different atomic parts, under the desired smoothing process. Specifically, we treat each intermediate value in the computation as a random variable with a certain probability distribution. We use mean and variance statistics to model these random variables. A key insight in our work is that we derive more accurate rules for modeling the variance in addition to the mean statistics considered in the previous work of Dorn et al. [DBLW15]. Each part of the program accepts one or more inputs, which are assumed to be Gaussian distributed according to these mean and variance statistics, and outputs a single variable, which is also assumed to be Gaussian distributed. In this manner, we can smooth arbitrary programs that operate over floating-point numbers. Our approach can be applied to bandlimit shader programs, because we take as input an original shader that may have aliasing, and produce as output bandlimited approximations that have been convolved with the Gaussian kernel.

For the different atomic parts of the input program, we need rules for how to approximate the mean and variance of the smoothed result. The previous work of Dorn et al. [DBLW15] has one such rule. We improve the accuracy of this rule, relate it to our framework, and explain a class of functions (or programs) for which it gives exact results (Section 4.1). We also introduce new rules that are more precise, but also more complex to compute. Specifically, we develop a novel adaptive Gaussian approximation (Section 4.2). This approximation handles multivariate Gaussian distributed inputs, is exact for a larger class of functions than previous work, and accurate to the second power of the standard deviation for functions with certain analytic properties. We also relate Monte Carlo sampling (Section 4.3) to our framework. For our last approximation rule, we discuss how compactly supported kernels (Section 4.4) can be used for parts of the computation that would otherwise be undefined. As an illustrative example, in Figure 3, we show the application of each of our approximate smoothing rules to a simple 1D function. In this case, smoothing is applied only to the single input dimension (x). In particular, the previous work of Dorn et al. performs poorly, as shown in Figure 3, when a function changes in frequency across spatial coordinates. This happens often for shaders because of foreshortening: frequency changes occur as a texture becomes distant from the camera.

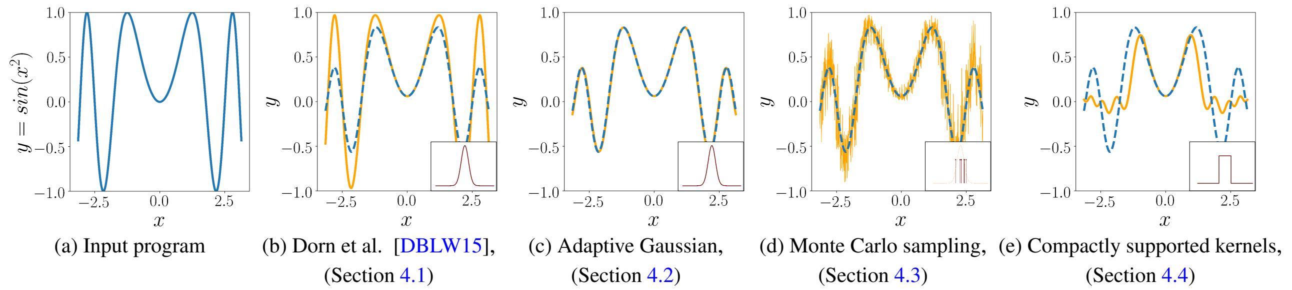
For each atomic part of the input program, we have different options for approximations, so we use a genetic search to apply rules to individual and connected groups of atomic parts. The search algorithm finds Pareto-optimal shader variants that optimally trade off running time and approximation error (Section 5). We also show how we can make minor quality improvement to the resulting programs by applying denoising (Section 6).

To evaluate our framework, we applied to three geometries a variety of complex shaders, including shaders with parallax mapping, animation, and spatially varying statistics. We compare the performance with Dorn et al. [DBLW15] and commonly used supersampling. Our framework gives a wider selection of band-limited programs with less error than Dorn et al. [DBLW15]. Our shaders are frequently an order of magnitude faster than supersampling for comparable errors.

2. Related work

Mathematics and smoothing. Smoothing a function is beneficial in domains such as optimizing non-convex or non-differentiable objectives [Nes05, CX99, CC99]. In numerical optimization, this approach is sometimes known as the continuation method or mollification [ENW95, EN97, Wu]. In our framework, we model the smoothing process on the input program by relating the statistics of each variable, and apply a variety of approximations to smooth the program. Our idea of associating a range with each intermediate value of a program is conceptually similar to interval analysis [Moo79]. Chaudhuri and Solar-Lezama [CSL11] developed a

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.



(a) Input program (b) Dorn et al. [DBLW15], (Section 4.1) (c) Adaptive Gaussian, (Section 4.2) (d) Monte Carlo sampling, (Section 4.3) (e) Compactly supported kernels, (Section 4.4)

Figure 3: *A visual example of our approximate smoothing rules. (a) The input program is the function $y = f(x) = \sin(x^2)$. This program is decomposed in our framework as the composition of two atomic parts that we do know how to smooth: $\sin()$ and x^2. The "ground truth" correctly smoothed program (or function) is shown in blue dashed curves in subfigures (b-e). This is determined by a convolution that is sampled at a very high sample rate. The orange lines in subfigures (b-e) approximate the ground truth convolution by using different approximation rules. The dark red subplots in (b-e) give an abstract illustration of the kernels that were actually used to evaluate these. (b) The approximation by Dorn et al. [DBLW15] (Section 4.1); (c) Our adaptive Gaussian approximation (Section 4.2); (d) Monte Carlo sampling approximation with 8 samples (Section 4.3); (e) Compactly supported kernels approximation: here we use a box kernel (Section 4.4). We use a standard deviation of $\sigma = 0.25$ for all input distributions.*

smoothing interpreter that uses intervals to reason about smoothed semantics of programs. The homogeneous heat equation with initial conditions given by a nonsmoothed function results in a smoothing process, via convolution with its Green's function, the Gaussian. Thus, connections can be made between convolution with a Gaussian and results for the heat equation, such as Łysik [Łys12].

Procedural shader antialiasing. The use of *antialiasing* to remove sampling artifacts is important and well studied in computer graphics. The most general and common approach is to numerically approach the band-limited signal using supersampling [AGB00]. Stochastic sampling [DW, Cro77] is one effective way to achieve this. The sampling rate can be effectively lowered if it is adaptively chosen according to the contrast of the pixel [DW, Mita, HJW*08, Mitb]. In video rendering, samples from previous frames can also be reused for computation efficiency [YNS*09]. An alternative to sample-based *antialiasing* is to create a band-limited version of a procedural shader. This can be a difficult task because analytically integrating the function is often infeasible. There are several practical approaches [Ebe03] that approximate the band-limited shader functions by sampling. This includes clamping the high-frequency components in the frequency domain [NRS82], and producing lookup tables for static textures using mipmapping [Wil83] and summed area tables [Cro84].

Like our work, and unlike most other work in this area, Dorn et al. [DBLW15] use a compiler-driven technique to approximate a smoothing convolution by decomposing an arbitrary input program into atomic parts that we know how to individually smooth. Like our work, Dorn et al. use a genetic search to select between these rules. We adapt Dorn et al. as one of the approximation rules into our framework with two improvements: better standard deviation estimates and the collection of a Pareto frontier of smoothed programs instead of one single output program. Unlike Dorn et al. [DBLW15], which models only mean statistics, our framework flexibly incorporates both mean and variance statistics. We also use several approximations that have higher accuracy, which can better model textures that change in spatial frequency due to foreshortening.

Heuristic search over programs. Genetic algorithms and genetic programming (GP) are general machine learning strategies that use an evolutionary methodology to search for a set of programs that optimize some fitness criterion [Koz92]. In computer graphics, Kensler and Shirley [KS] demonstrated that genetic algorithms could be used to optimize ray-triangle intersection routines. Sitthi-Amorn et al. [SAMWL11] described a GP approach to the problem of automatic procedural shader simplification. Other researchers have also investigated automatic shader simplification by heuristic search methods that simplify programs [OKS, Pel, HFTF15], and by jointly modifying shaders and geometry [WYY*14]. Brady and colleagues [BLPW14] showed how to use GP to discover new analytic reflectance functions. We use a similar approach as [SAMWL11] to automatically generate the Pareto frontier of approximately smoothed functions.

3. Decomposition and Associated Notation

In this section, we first explain in Section 3.1 how the input program is decomposed into atomic parts. Next, in Section 3.2, we define math notation associated with these atomic parts.

3.1. Decomposing the Input Program into Atomic Parts

Most input programs lack a closed-form solution for their convolution with a Gaussian kernel. We therefore decompose the computation graph into atomic parts that individually have known closed-form solutions. We then compute approximate mean and variance statistics for each part, and substitute the mean and variance that are output from one group of compute nodes as the inputs for any subsequent compute nodes.

Our compiler-based framework assumes the input program has a compute graph, where each node represents a floating-point computation, and the graph is a directed acyclic graph (DAG). This compute graph is constructed directly by the programmer using atomic operations such as addition, multiplication, trigonometric functions, and others: please see the supplemental document for a full list. We use lower-case letters such as x and y to represent real values (scalars) in the input program. These can be either input, output, or intermediate values. We use corresponding capital letters

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

such as X and Y to represent random variables for the distribution of these values given that the input variables are assumed to be independently Gaussian distributed. In our implementation, although we assume that the input variables are independent, this is not limiting because dependencies such as e.g. a joint Gaussian distribution can easily be created in the program by passing the inputs through a linear transformation. For each node X in the computation, we use μ_X to denote its mean and σ_X^2 for its variance. Note we use these random variables as a helpful conceptual device to determine statistics, but in most cases, we never actually sample from these random variables (except for Monte Carlo sampling (Section 4.3), which is sampled). Our compiler then carries mean and variance computations forward through the compute graph, using the various approximate smoothing rules of Section 4, and collects the output by taking the mean value of the output variable.

As an example, for shader bandlimiting, the input variables are the 2D screen coordinate (u,v), with associated random variables, U and V. For the random variables, the means are the pixel positions, $\mu_U = u$, and $\mu_V = v$, and the standard deviations are $\sigma_U = \sigma_V = 0.5$, i.e. half a pixel, to suppress aliasing beyond the Nyquist rate. Our compiler then gathers the mean of the output random variables to obtain the rendered color.

3.2. Math Notation For Smoothing a Single Atomic Part

In this subsection, we define the notation we will use for smoothing a single atomic part of a program. This can be done by either using convolutions or random variables, in two equivalent notations. First, we note that throughout the paper, we use bold to indicate vectors and matrices. In some cases, we might consider the case where a random variable is scalar, which we could denote as X, and then we might later consider the case of a random vector, which we might similarly denote as $\mathbf{X}$. To avoid confusion between these similar symbols, in this situation we first indicate in the text whether the variable is a scalar or vector quantity.

We now present our smoothing operator. Assume we are smoothing a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, which maps inputs $\mathbf{x}$ to outputs $\mathbf{y}$. We use the $\hat{}$ operator to denote smoothing using convolution, so the smoothed function is $\hat{\mathbf{f}}(\mathbf{x},\mathbf{\Sigma})$, defined as:

$$\begin{aligned}\hat{\mathbf{f}}(\mathbf{x},\mathbf{\Sigma}) &= (\mathbf{f} * G)(\mathbf{x}) \\ &= \int_{\mathbb{R}^n} \mathbf{f}(\mathbf{x}-\mathbf{u})G(\mathbf{u},\mathbf{\Sigma})d^n\mathbf{u} \\ &= \int_{\mathbb{R}^n} \mathbf{f}(\mathbf{u})G(\mathbf{x}-\mathbf{u},\mathbf{\Sigma})d^n\mathbf{u}\end{aligned} \tag{1}$$

In Equation (1), $G(\mathbf{u},\mathbf{\Sigma})$ is the smoothing kernel that is used to smooth the original function $\mathbf{f}(\mathbf{x})$, $\mathbf{\Sigma}$ is a covariance matrix associated with the kernel (more precisely, $\mathbf{\Sigma}$ is the covariance matrix of the random vector with a probability density function given by the kernel G), and the convolution is over the first variable of each function. To more explicitly identify the kernel as being G, we can also use the notation $\hat{\mathbf{f}}^G(\mathbf{x},\mathbf{\Sigma})$. For isotropic kernels, which have the same standard deviation σ for all dimensions, we also use $\hat{\mathbf{f}}(\mathbf{x},\sigma^2)$ as shorthand for $\hat{\mathbf{f}}(\mathbf{x},\mathbf{I}\sigma^2)$, where $\mathbf{I}$ is the identity matrix. The convolution kernel $G(\mathbf{x},\mathbf{\Sigma})$ can be any non-negative kernel that integrates over $\mathbb{R}^n$ to one. This allows us to interpret the kernel also as a probability density function. In this paper, we frequently use the Gaussian kernel, which we conveniently center at the origin:

$$G(\mathbf{u},\mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n|\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathbf{\Sigma}^{-1}\mathbf{u}\right) \tag{2}$$

If $\mathbf{f}(\mathbf{x})$ happens to be a shader program, then as is discussed in [DBLW15], $\hat{\mathbf{f}}(\mathbf{x},\mathbf{\Sigma})$ is simply a band-limited version of the same procedural shader function.

We now show the connection between the convolution of Equation (1) and the random variables associated with a program's computations. We assume that in the input program, an intermediate scalar random value Y is computed by applying a scalar-valued function f to an input random vector $\mathbf{X}$ (in $\mathbb{R}^k$), i.e., $Y = f(\mathbf{X})$. If the probability density function of $\mathbf{X}$ is $g_{\mathbf{X}}$, then by the law of the unconscious statistician, μ_Y is:

$$\mu_Y = E[f(\mathbf{X})] = \int_{\mathbb{R}^k} f(\mathbf{u})g_{\mathbf{X}}(\mathbf{u})d^k\mathbf{u} \tag{3}$$

As an example, if the input random vector $\mathbf{X}$ is normally distributed as $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{X}}, \mathbf{\Sigma}_{\mathbf{X}})$, then Equation (3) becomes:

$$\begin{aligned}\mu_Y &= \int_{\mathbb{R}^k} f(\mathbf{u})G(\mathbf{u}-\boldsymbol{\mu}_{\mathbf{X}},\mathbf{\Sigma}_{\mathbf{X}})d^k\mathbf{u} \\ &= (f(\mathbf{u}) * G(\mathbf{u},\mathbf{\Sigma}_{\mathbf{X}}))(\boldsymbol{\mu}_{\mathbf{X}}) \\ &= \hat{f}(\boldsymbol{\mu}_{\mathbf{X}},\mathbf{\Sigma}_X)\end{aligned} \tag{4}$$

Thus, we find that we can switch between two equivalent notations. In "convolution notation," we can write $\mu_Y = \hat{f}(\boldsymbol{\mu}_{\mathbf{X}},\mathbf{\Sigma}_{\mathbf{X}})$. This is the same as using "random variable notation" and writing $E[f(\mathbf{X})]$. This gives some intuition for how we can either use convolutions or expectations of random variables to smooth programs.

4. Approximate Smoothing Rules

Using the machinery from the previous section, we can now decompose the input program into atomic parts, and represent these as a directed acyclic graph (DAG). Each part accepts one or more inputs, which are Gaussian distributed according to mean and variance statistics, and outputs a variable, which is also Gaussian distributed. This section develops different approximation rules used to compute the mean and variance of the output variable. These rules allow for different trade-offs between efficiency, accuracy, and noise. The approximation rules are visualized in Figure 3.

4.1. Approximation of Dorn et al. 2015

We integrate the approximation rule described in Dorn et al. [DBLW15] as one of our approximation options. Dorn's rule involves computing the smoothed function by convolving with a Gaussian kernel. Suppose an intermediate scalar variable y is computed from another scalar variable x, and the associated random variables are Y and X, respectively, where $Y = f(X)$. Then μ_Y is:

$$\mu_Y = \hat{f}(\mu_X,\sigma_X^2) \tag{5}$$

This is the same as the result we derived in Equation (4). Here $\hat{f}(\mu_X,\sigma_X^2)$ is computed from its definition in Equation (1). In the supplemental document Table 3, we show commonly used functions f and their corresponding smoothed functions $\hat{f}$. This table of commonly used functions includes polynomials, reciprocal,

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

sine, cosine, tangent, hyperbolic trigonometric functions, exponent, Heaviside step, fract, floor, and ceiling, and the squares of these functions. For example, if $y = \sin(x)$, and we are using a Gaussian kernel, then we can use Equation (5) and look up in the supplemental document Table 3 to obtain $\mu_Y = \sin(\mu_X)\exp(-\sigma_X^2/2)$.

In Dorn's paper, the output σ_Y is determined based on the following simplifying assumption: output σ is a linear combination of the axis-aligned input σs in each dimension. Simple rules are used, such as σ for addition and subtraction are the sum of input σs, and σ for multiplication or division are the product or quotient, respectively, of the input σs. In all other cases, including function calls, the output σ is the average of the non-zero σs of all the inputs.

We make two improvements to Dorn et al. [DBLW15], and use the improved variant of this approximation rule for all comparisons in our paper. The first improvement gives better standard deviation estimates, and the second collects a Pareto frontier. For the standard deviations (known as "sample spacing" in Dorn et al. [DBLW15]), we detect the case of multiplication or division by a constant and adjust the standard deviation accordingly (i.e. $\sigma_{aX} = |a|\sigma_X$). This improvement helps give more accurate estimates of the standard deviations and thus reduces the problem seen in Dorn et al.'s Figure 5(c), where the initial program has substantially wrong variances. Our second improvement is to collect not just a single program variant with least error, but instead a Pareto frontier of program variants that optimally trade off running time and error. This process is described later in Section 5.

One simple question we could ask is: *for what class of functions does the improved Dorn et al. [DBLW15] approximation result in the exact answer?* More precisely, we could apply this rule to a compute graph whose inputs are independently Gaussian distributed, and determine a class of functions whose compute graph results in an exact output for the mean. Even for linear functions, these approximation rules give incorrect variance. For example, Dorn's estimate gives incorrectly $\mathrm{Var}(X - X)$ as $(2\sigma_X)^2$, when it should be zero. However, if a Gaussian distributed input variable is multiplied by or added to a constant, this rule results in the correct mean and variance. Thus, this rule gives exact results for the mean for linear combinations or separable products of functions $f(ax+b)$ that we know smoothed $\hat{f}$ for, where a, b can be any constants. For example, for $g(x,y,z) = ((2x)^2 + \cos(y))z^2$ it produces an exact result, since smoothed results are available for polynomials and cosine.

4.2. Adaptive Gaussian Approximation

In this novel approximation, we model the input variables to a compute node as being distributed as a multivariate Gaussian, and then also approximate the output of the node as Gaussian by collecting its mean and standard deviation. This rule therefore allows correlations between variables to be modelled, and the variance term of the output Gaussian to adapt, based on the inputs and the previous computation.

Suppose that a scalar-valued random variable Y is computed from a jointly Gaussian distributed random vector $\mathbf{X}$ as $Y = f(\mathbf{X})$. We assume we know the distribution for $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{X})$. Similarly as in Section 4.1, μ_Y can be determined from Equation (4) and Equation (1). The result μ_Y can then be looked up in a table of convolutions such as the supplemental document Table 3, or from multi-dimensional formulas that we will see shortly. However, the standard deviation σ_Y is determined differently based on the definition of the variance of Y:

$$\begin{aligned} \sigma_Y^2 &= E[Y^2] - E[Y]^2 \\ &= \widehat{f^2}(\boldsymbol{\mu}_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{X}) - \hat{f}^2(\boldsymbol{\mu}_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{X}) \end{aligned} \tag{6}$$

For this rule, we can ask, *for what functions does the adaptive Gaussian approximation rule result in the exact answer?* More precisely, we could apply the adaptive Gaussian approximation rule to compute the mean and variance of every node of a compute graph whose inputs are independently Gaussian distributed random variables. Then we can determine a class of functions whose associated compute graphs result in the output mean being exact. We know that any affine transform of a multidimensional Gaussian results in another multidimensional Gaussian. We can construct the affine functions using the binary scalar operators $(+), (-), (\cdot)$, and can calculate the mean and variance exactly for such affine functions (using either the sampled or affine estimations for correlations that are discussed later in this section). Thus, this approximation rule gives exact results for the mean for linear combinations or separable products of functions $f(\mathbf{A}\mathbf{x})$ that we know smoothed $\hat{f}$ for, where $\mathbf{A}$ is any affine transformation, and $\mathbf{x}$ is the vector of input variables. For example, for $g(x,y,z) = ((2x+y)^2 + \cos(y-2x))z^2$ the adaptive Gaussian rule produces an exact result, since exact smoothing results are available for polynomials and cosine.

A second question we can ask is: *if the answer is not exact, to what order is the result accurate?* Suppose for simplicity that the input variables are independent and Gaussian distributed, each with a standard deviation of σ. By using Green's function [BS03] on the convolution of Equation (1) , we can find a Taylor expansion for the function $\hat{f}(\mathbf{x}, \sigma^2)$ in terms of $f(\mathbf{x})$:

$$\hat{f}(\mathbf{x}, \sigma^2) = f(\mathbf{x}) + \frac{1}{2}\sigma^2\nabla^2 f(\mathbf{x}) + \frac{1}{(2!)2^2}\sigma^4\nabla^4 f(\mathbf{x}) + \dots \tag{7}$$

The derivation of Equation (7) assumes that f is *real analytic* on $\mathbb{R}^n$, and can be extended to a holomorphic function on $\mathbb{C}^n$, so that all the derivatives exist, and the Taylor series has an infinite radius of convergence [Wik17]. This class of functions includes polynomials, sines, cosines, and compositions of these. It is necessary to assume that the function is bounded by exponentials: the precise conditions are discussed by Łysik [Łys12]. These properties could hold for some shader programs, but even if they do not hold for an entire program, they often hold for program sub-parts. We show in the supplemental document Section 13 that a single function composition gives a result accurate to σ^2 for this rule. Similarly, this property can be proved via induction for multiple function compositions. We conclude that for functions with certain analytic properties, the adaptive Gaussian rule is accurate to σ^2.

There are also other second order accurate approximations, such as simply truncating the Taylor expansion in Equation (7) to use only the first and second term. To illustrate why adaptive Gaussian gives a more accurate approximation, we show an example in Figure 4. The truncated Taylor expansion results in amplifying high frequencies, instead of attenuating them.

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

(a) Input function (b) Adaptive Gaussian approximation (§4.2) $\sigma = 0.5$ (c) Truncated Taylor expansion (Equation (7)) $\sigma = 0.5$

Figure 4: *A comparison of different approximation techniques. (a) The input function $f(x) = \sin(x^2) + \frac{1}{100}x^2$. The ground truth correctly band-limited functions $\hat{f}(x)$ are shown in blue in subfigures (b-c). These were determined by sampling at a high sample rate. (b) Our adaptive Gaussian approximation (Section 4.2) is shown in orange and compared against the ground truth in blue. The approximation is good. (c) A truncated Taylor expansion with 10 terms does not result in smoothing.*

We now explain how binary functions can be handled using this approximation rule. Suppose a binary function $f(a,b)$ takes scalar inputs a,b and the associated random variables are A and B, respectively. We make the assumption that A and B are distributed according to a bivariate Gaussian:

$$A,B \sim \mathcal{N}\left(\begin{bmatrix}\mu_A \\ \mu_B\end{bmatrix}, \begin{bmatrix}\sigma_A^2 & \rho\sigma_A\sigma_B \\ \rho\sigma_A\sigma_B & \sigma_B^2\end{bmatrix}\right) \tag{8}$$

Here, σ_A and σ_B are standard deviations of A and B. These can be determined directly by applying the approximation rules to the input nodes A and B. Here ρ is the correlation term between A and B. We will talk about how we choose ρ later in this section. The mean and standard deviation for the binary functions of addition $(A+B)$, subtraction $(A-B)$ and multiplication $(A \cdot B)$ can be derived from these assumptions based on properties of the Gaussian distribution [PP*08]:

$$\begin{aligned}
\mu_{(A\pm B)} &= \mu_A \pm \mu_B \\
\sigma^2_{(A\pm B)} &= \sigma_A^2 + \sigma_B^2 \pm 2\rho\sigma_A\sigma_B \\
\mu_{(A\cdot B)} &= \mu_A\mu_B + \rho\sigma_A\sigma_B \\
\sigma^2_{(A\cdot B)} &= \mu_A^2\sigma_B + \sigma_A\mu_B^2 + 2\rho\mu_A\mu_B\sigma_A\sigma_B + \sigma_A^2\sigma_B^2(1+\rho^2)
\end{aligned} \tag{9}$$

For the binary function of division, we reduce this to multiplication by using the substitution $a/b = a \cdot b^{-1}$. The mean and standard deviation for division can then be calculated via the composition rules. Here, $g(b) = b^{-1}$ is an univariate function with singularity at $b = 0$. Technically, the mean and variance therefore do not exist if the Gaussian kernel is used. We work around this singularity by approximating using a compact kernel with finite support. This will be described in detail in Section 4.4. In the supplemental Section 10, we derive formulas for additional multivariate functions, such as modulo, comparisons, and a ternary select function.

As we discussed before, for binary functions, we approximate the input random variables A and B as bivariate Gaussian with correlation coefficient ρ (Equation (8)). In general, it is difficult to determine ρ, because determining ρ exactly involves an integral over the entire subtrees of the computation. In our framework, we provide three options to approximate ρ: (1) Assume ρ is zero; (2) Assume ρ is a constant for each node. The constant value is estimated at training stage by sampling; (3) Estimate ρ based on a simplified assumption that the given nodes are affine functions of the inputs. For case (3) we simply take the gradients with respect to the program's input variables of the terms a and b, normalize these gradients, and take their dot product, which recovers ρ. We do this using reverse mode automatic differentiation. Note that the resulting ρ estimate for case (3) is exact if the nodes are in fact affine with respect to the inputs, and otherwise is accurate to second order in σ. We explored these different rules in our genetic search. In practice, we find that for shader programs, using only rule (1), $\rho = 0$ typically gives good results. If the other rules (2) and (3) are also included, minor quality improvements are gained, but these rules are used relatively rarely by our genetic search process of Section 5. We include in the supplemental document Section 11 more details about these other choices for correlation coefficients.

4.3. Monte Carlo Sampling

We adapt Monte Carlo stochastic sampling [Coo86, DW] to our framework. We first collect each largest connected sub-graph of the computation that has been specified to use Monte Carlo sampling. Then we collect the inputs to the sub-graph, $X_1, \dots, X_m$. These could either be input random variables to the entire input program, or intermediate variables calculated as the output from other parts of the compute graph. For simplicity, here we assume these inputs are independently Gaussian distributed based on their specified means μ_{X_i} and standard deviations σ_{X_i}. For each output f of the sub-graph that is used by later computations, we compute the mean and standard deviation statistics of $Y = f(X_1, \dots, X_m)$ by sampled estimators:

$$\begin{aligned}
\mu_Y &= \frac{1}{n}\sum_{i=1}^{n} f(\mu_{X_1} + \mathcal{N}_{i,1}\sigma_{X_1}, \dots, \mu_{X_m} + \mathcal{N}_{i,m}\sigma_{X_m}) \\
\sigma_Y^2 &= \frac{1}{n}\sum_{i=1}^{n} f^2(\mu_{X_1} + \mathcal{N}_{i,1}\sigma_{X_1}, \dots, \mu_{X_m} + \mathcal{N}_{i,m}\sigma_{X_m}) - \mu_Y^2
\end{aligned} \tag{10}$$

Here, each $\mathcal{N}_{ij}$ are random numbers independently drawn from normal distribution $\mathcal{N}(0,1)$, and n is the number of samples. We experimented with applying the Bessel's correction [So08] to correct the bias in variance that occurs for small sample counts n. In practice, we found it does not have a significant improvement on the result for our system.

The approximation converges to the ground truth for large sample numbers, and the output program simplifies to supersampling [Coo86] when the entire input program is approximated under Monte Carlo sampling. The error of the Monte Carlo sampling σ_M is estimated as follows [Fei11].

$$\sigma_M \approx \frac{\sigma_Y}{\sqrt{n}} \tag{11}$$

Here, σ_Y is the standard deviation computed from Equation (10) and n is the number of samples.

4.4. Compactly Supported Kernels Approximation

Because the Gaussian kernel has infinite support, it cannot be used on functions with undefined regions. For example, $\sqrt{x}$ is only defined on non-negative x, and its convolution with Gaussian using

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

Equation (1) does not exist. However, even if an input program contains such functions as sub-parts, the full program may have a well-defined result, so smoothing should still be possible for such programs. To handle this case, we use compactly supported kernels.

Results for certain compactly supported kernels can be obtained by using repeated convolution [Hec86] of boxcar functions. This is because such kernels approximate the Gaussian by the central limit theorem [Wel86]. In our framework, we use box and tent kernels to approximately smooth functions with undefined values. Because the convolution with a box kernel is easier to compute, this approximation can also be used when the Gaussian convolution does not have a closed-form solution. In the supplemental document Table 3, we list smoothed results for commonly used functions using the box kernel.

When integrating against a function that has an undefined region, it is important to make sure that the integral is not applied at any undefined regions. Our solution to this is to make the kernel size adapt to the location at which the integral is evaluated at. Thus, the integral is no longer technically a convolution, because it is not shift-invariant. We first measure the distance r from the value x that we are determining the integral at to the function's nearest undefined point. If the kernel half-width was h before re-scaling, then we rescale the half-width to be $\min(h, \lambda r)$. Here λ is a constant less than one, and in practice we use $\lambda = \frac{1}{2}$.

We can also use this truncation mechanism to better model functions such as $\text{fract}(x) = x - \lfloor x \rfloor$, which have many discontinuities. Clearly, fract() is discontinuous at integer x. If we input a distribution that spans a discontinuity, such as $X \sim \mathcal{N}(0, 0.1^2)$, into fract(), we find the output $Y = \text{fract}(X)$ may be bimodal, with some values close to zero, and others close to one. If we fit a Gaussian to this bimodal distribution, as our approximation rules propose, then the mean would be $\frac{1}{2}$, which is far away from the two modes. This may result in a poor approximation, which can show up in tiled pattern shaders (which use fract) as a bias towards the center of the tile's texture. One fix would be to randomly select either mode, based on the probability contained in each mode. However, this introduces sampling noise. Instead, we truncate the filter at the location of the discontinuity when the original kernel support is smaller than a truncation constant T_1 (in practice, we use $T_1 = 1/4$). When the original kernel support is above a larger truncation constant T_2 (we use $T_2 = 1/2$) we do not truncate. In between kernel sizes T_1 and T_2 we rescale the kernel size linearly between these endpoints.

5. Genetic Search

In this section, we describe the genetic search algorithm. This automatically assigns approximation rules to each computation node. The algorithm finds the Pareto frontier of approximation choices that optimally trade off the running time and error of the program.

We developed this genetic search because it allows users to explore the trade-off between efficiency and accuracy of the smoothed program. Although developers can manually assign approximation rules, we found this to be a time-consuming process that can easily overlook beneficial approximation combinations. This is because the search space for the approximations is combinatoric.

Our genetic search closely follows the method of Sitthi-Amron et al. [SAMWL11]. We adopt their fitness function and tournament selection rules, and we use the same method to compute the Pareto frontier of program variants that optimally trade-off running time and error with ground truth.

We start with "decent initial guesses." For each approximation rule, we create a program variant where the rule is applied to all the expression nodes. For such initial guesses, we also apply single-point cross-over. The cross-over operation partitions the program into two parts separated by an arbitrary node, assigns approximation rules from one variant to the first part of the program, and rules from another variant to the other part. Next, we employ cross-over and mutation operations to explore the search space. The mutation step chooses a new approximation rule, and with equal probability, assigns this new rule to 1, 2, or 4 adjacent expression nodes in depth-first order. As an alternative, with equal probability, the new approximation rule can also be assigned to the whole subtree of an arbitrary node. We use tournament selection to select program variants for mutation and crossover. Our tournament selection works by randomly sampling 4 program variants from the population, eliminating variants that are not Pareto optimal, and then randomly choosing a remaining program with optimal running time and error.

For the Monte Carlo sampling approximation, during initialization and mutation, we select sample counts with equal probability from the set $\{2, 4, 8, 16, 32\}$. For the determination of correlation coefficients described in Section 4.2, we pick with equal probability one of the three options.

6. Optional Quality Improvement

At this point, we assume we have applied the approximation rules described in Sections 4.1 through 4.4 to an input program. We can optionally improve the approximation quality by applying denoising to program variants that use Monte Carlo sampling.

When Monte Carlo sampling is used as part of the approximation, noise is introduced because of the relatively small sample count. A variety of techniques have been developed to filter such noise [KBS, BVM*17, RKZ]. We implement the non-local means denoising method [BCM05, BCM11] with Laplacian pyramid [LWC*08]. We find that aesthetically appealing denoising results can be obtained using a three level Laplacian pyramid, with a patch size of 5, search radius of 10, and denoising parameter h is 10 for the lower resolutions, and searched over or set by the user for the finest resolution. In the genetic search process (Section 5), we experimented with allowing the algorithm to search from a variety of denoising parameters for the best result. However, because our denoising algorithm incurs some time overhead, it ends up being only rarely chosen. Thus, in our current setup, denoising is typically specified by the user manually choosing that he or she wants to denoise a result.

7. Evaluation

The previous work of Dorn et al. [DBLW15] was evaluated only on planar geometry, with relatively simple shaders. To provide a more challenging and realistic benchmark, we authored 21 shaders and applied these to three geometries. Unlike the simple shaders of Dorn et al., these include shaders that have a Phong lighting model,

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

Table 1: *A table of our 21 shaders. At the top we list our 7* base shaders*, which are each combined with 3 different choices for parallax mapping, listed at the bottom. We also report the number of non-comment lines and expressions in each program fragment.*

Shader	Lines	Exprs	Description
Base shaders			
Bricks	38	192	Bricks with noise pattern
Checkerboard	20	103	Greyscale checkerboard
Circles	16	53	Tiled greyscale circles
Color circles	26	164	Aperiodic colored circles
Fire	49	589	Animating faux fire
Quadratic sine	26	166	Animating sine of quadratic
Zigzag	24	224	Colorful zigzag pattern
Parallax mappings			
None	0	0	No parallax mapping
Bumps	21	203	Spherical bumps
Ripples	23	178	Animating ripples

animation, spatially varying statistics, and which include parallax mapping. For the parallax mapping, we implemented the "safer mapping" formula from Section 4.1.4 of Szirmay-Kalos and Umenhoffer [SKU08]. Our 21 shaders were produced by combining 7 *base shaders* with 3 choices for parallax mapping: none, bumps, and ripples. In Table 1, we describe our base shaders, the choices for parallax mapping, and the associated code complexity. We applied the shaders on 3 different geometries: an infinite plane and two curved geometries sphere and hyperboloid. Each shader program is tuned independently on each of the geometries.

We performed our evaluation on an Intel Core i7 6950X 3 GHz (Broadwell), with 10 physical cores (20 hyperthreaded), and 64 GB DDR4-2400 RAM. All shaders were evaluated on the CPU using parallelization. The tuning of each shader took between 1 and 6 hours of wall clock time, with 1 to 3 hours for planar geometry. However, we note that good program variants are typically available after minutes to low tens of minutes, and most of the remaining tuning time is spent making slight improvements to the best individuals. Please see the supplemental Section 14 for results available after tuning for 10 minutes. Also, our tuner is intentionally a research prototype that is not particularly optimized: it could be significantly faster if the code generator were optimized, it was parallelized more effectively, cached more redundant computations, or targeted the GPU. We found that getting the code generation and math details right was challenging, so we only targeted CPU code for simplicity in our prototype.

7.1. Evaluation for Planar Geometry

In this section, shaders are evaluated on an infinite plane. Results for 7 of our shaders are presented in Figure 1 and Figure 5, including one result for each base shader. The result for our method was selected by a human choosing for each shader a program variant that has sufficiently low error. Dorn et al. [DBLW15] typically cannot reach sufficiently low errors to remove the aliasing, so we simply selected the program variant from Dorn et al. that reaches the lowest error. The supersampling result was selected based on evaluating supersampling program variants that use 2, 4, 8, 16, 32 samples, and selecting the one that has most similar time as ours. Please see our supplemental video for results with a rotating camera for all 21 shaders.

We also show in Figure 6 time versus error plots for the Pareto frontiers associated with these 7 shaders. Note that Dorn et al. typically has significantly higher error, which manifests in noticeable aliasing. Also note that the supersampling method frequently takes an order of magnitude more time for equal error. Plots for all 21 of our shaders are included in the supplemental document.

Statistics for the approximations used are presented in Table 2. Note that a rich variety of approximation strategies are used: all four choices for approximation are selected for different programs. Adaptive Gaussian and Monte Carlo sampling are important when high result quality is important but fast running time is less important, as indicated in the bottom row of Table 2. In contrast, the Dorn et al. approximation is mainly useful when fast running time is desired in exchange for higher error. For the correlation term discussed in Section 4.2, when aggregated across all 21 shaders, nearly all approximations for programs on the Pareto frontier prefer the simple choice of $\rho = 0$. We weight each shader's contribution equally, and find 87% of program variants prefer $\rho = 0$, whereas only 4% use ρ a constant, and 6% use ρ estimated based on the affine assumption. We conclude that for shader programs, the simple choice of $\rho = 0$ in most cases suffices.

Note that our brick shader (shown in Figure 1) gives poor results for the method of Dorn et al. [DBLW15], while in that paper, a brick shader with similar appearance shows good results. This is because the brick shader in Dorn et al. [DBLW15] was implemented using floor() functions which can each be bandlimited independently, and then a good result is obtained by linearity of the integral. In our paper, we implemented a number of shaders using the fract() function to perform tilings that are exactly or appropriately periodic, including the brick shader. The fract() function ends up being more challenging to bandlimit for the framework of Dorn et al. [DBLW15], but our method can handle such shaders.

7.2. Evaluation for Curved Geometry

In this section, shaders are evaluated on two curved geometries: sphere and hyperboloid. Variables such as surface normal have more complicated distributions on curved geometries, while in planar geometry (Section 7.1), they are just constants. Because of this, shaders are tuned separately on each of the geometries.

Results for 7 of the shaders are presented in Figure 7, including one result for each base shader. The program variant shown in the result is chosen the same way as in Section 7.1. Please see our supplemental video to see these shaders rendered from a moving camera.

7.3. Geometry Transfer

We also performed some preliminary experiments related to geometry transfer. We wondered whether shaders trained on one geometry A when transferred to another geometry B would be competitive with directly training on geometry B. We found that transferring shaders between the two curved geometries generally gave

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

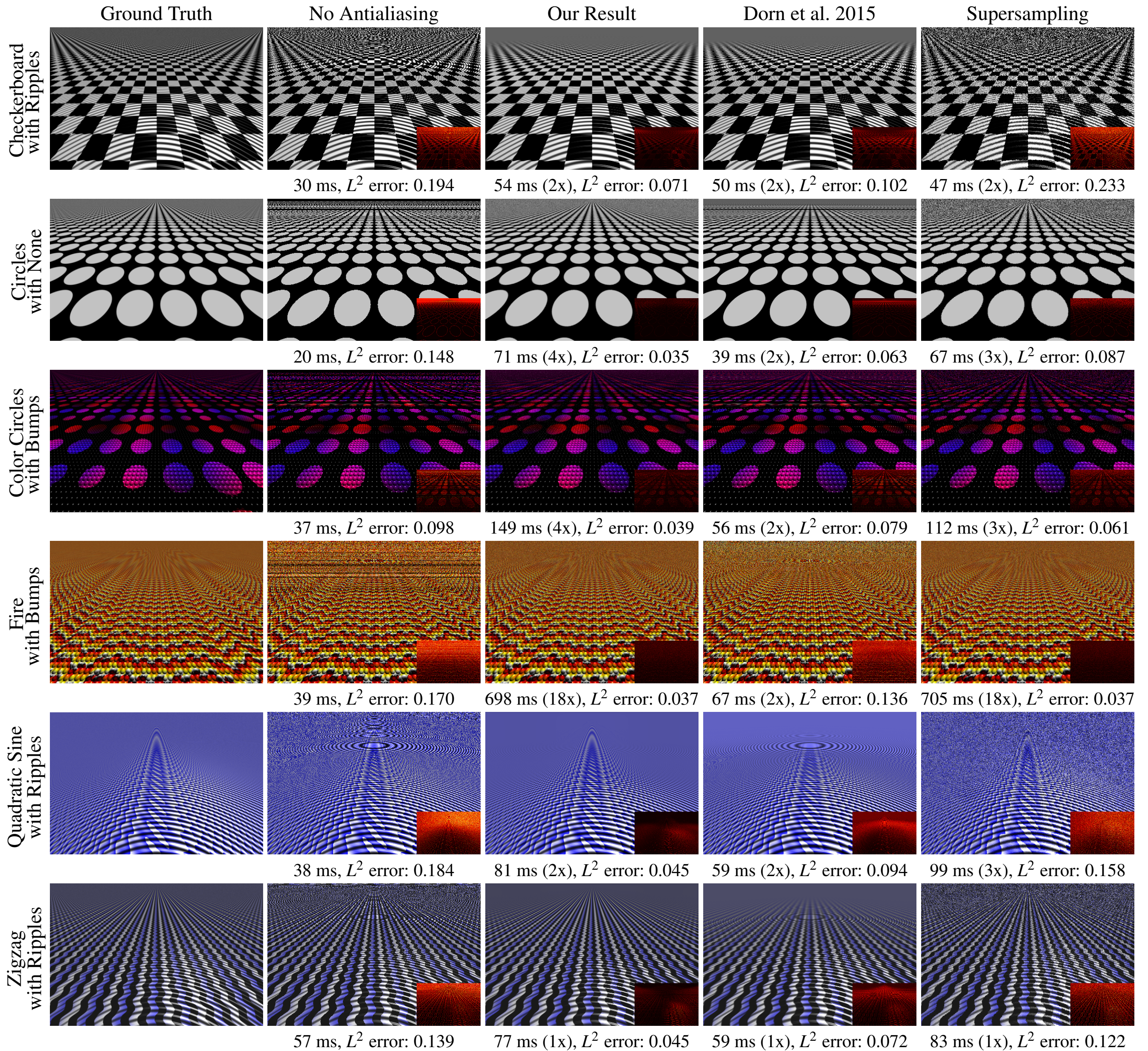
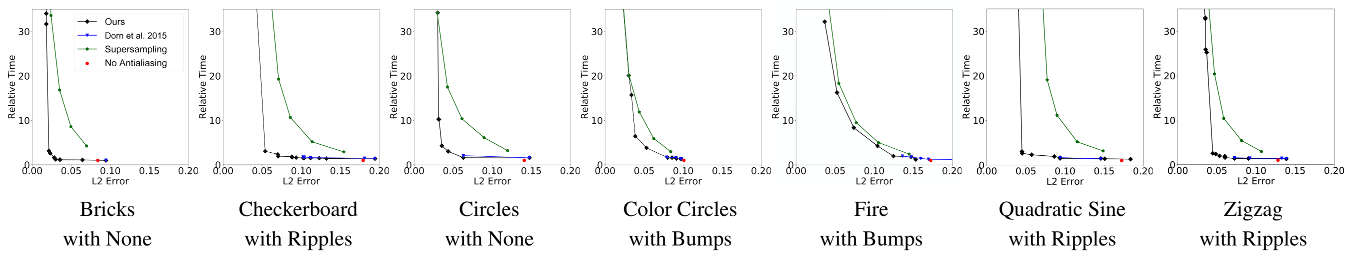


Figure 5: *Selected result images for 6 shaders on an infinite plane. Please see the supplemental video for a comprehensive comparison of all shaders. Reported below each shader are the time to render a frame, time relative to no antialiasing, and L^2 error. Please zoom in to see aliasing and noise patterns in the different methods. Program variants with comparable time were selected: see Section 7.1 for more details. Note that the amount of aliasing and error for our result is significantly less than Dorn et al. [DBLW15]. We typically have significantly less error and noise than the comparable supersampled results. Note also that for supersampling, the times relative to no antialiasing do not exactly match the sample count due to cache effects and variations in the running time depending on exactly where samples intersect geometry.*

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.



Bricks with None | Checkerboard with Ripples | Circles with None | Color Circles with Bumps | Fire with Bumps | Quadratic Sine with Ripples | Zigzag with Ripples

Figure 6: *Time versus error plots for planar geometry and the 7 shaders in Figure 1 and Figure 5. Here we show the Pareto frontier of program variants that optimally trade off running time and L^2 error. We show results for our method, Dorn et al [DBLW15], supersampling with varying numbers of samples, and the input shader without antialiasing. Note that our approach typically has significantly less error than Dorn et al [DBLW15] and is frequently an order of magnitude faster than supersampling for comparable error.*

Table 2: *Statistics of which approximations were chosen for different shaders on an infinite plane. We show statistics for the 7 program variants for the shaders presented in Figure 1 and Figure 5. We also show aggregate statistics over all 21 shaders, with each shader's contribution weighted equally. For the aggregate statistics we report statistics from the entire Pareto frontier, as well as for each shader choosing only the slowest, fastest, or median speed program variant. Our results show that a rich variety of our different approximation rules are needed for the best performance.*

Shader	Dorn et al. [DBLW15]	Adaptive Gaussian	Monte Carlo Sampling	None
Bricks w/ None	28%	0%	30%	29%
Checkerboard w/ Ripples	66%	34%	0%	1%
Circles w/ None	4%	21%	71%	4%
Color Circles w/ Bumps	8%	47%	44%	0%
Fire w/ Bumps	1%	7%	33%	60%
Quadratic sine w/ Ripples	13%	80%	0%	8%
Zigzag w/ Ripples	0%	91%	1%	8%
All shaders (Pareto frontier)	29%	15%	25%	30%
All shaders (fastest time)	13%	10%	0%	77%
All shaders (median time)	20%	19%	49%	13%
All shaders (slowest time)	10%	27%	49%	14%

good results for antialiasing that are competitive with direct training, transfer from curved geometries to plane tended to give good antialiasing results that are slower (due to optimizations made by the genetic search for the plane having a constant tangent, normal, binormal frame), and transfer from plane to curved geometry gave bad results (due to lack of diversity of training data for the frame). Please see the supplemental Section 15 for more details and results.

8. Discussion and Conclusion

Our approach has a number of important limitations. First, it can be forced to resort to Monte Carlo sampling especially if the input program has many discontinuities. When the other approximations are used, and the results are not exact, there can be small amounts of residual aliasing or biases. This is a limitation of assuming that distributions are Gaussian when they are not. For the curved geometries, we noticed that highly aliased regions are exposed for just a small percent of the pixel count, so this can cause the genetic search to focus more on areas with less aliasing. Future work might address this by diversely sampling from regions with different amounts of aliasing. We also do not currently handle texture, lookup tables, nor do we target the GPU. Although we sample across the time domain during the genetic search, we do not currently use any loss that discourages temporal aliasing, so there may be small amounts of temporal aliasing. Further, we currently handle conditionals in a limited manner by executing both branches, as we do for the select() function in the supplemental document Section 10, but future work could more comprehensively address conditionals. Finally, our genetic search algorithm may not be efficient enough for more complicated production shaders. Future work on static selection of approximation rules would be necessary to scale to significantly longer shaders.

In summary, in this paper, we presented a novel compiler framework that smoothes an arbitrary program over the floats by convolving it with a Gaussian kernel. We explained several different approximations and discussed the accuracy of each. We then demonstrated that our framework allows shader programs to be automatically bandlimited. This shader bandlimiting application achieves state-of-the-art results: it often has substantially better error than Dorn et al. [DBLW15] even after our improvements, and is frequently an order of magnitude faster than supersampling. Our framework is quite general, and we believe it could be useful for other problems in graphics, mathematics, and other disciplines. In order to facilitate reproducible research, we intend to release our source code under an open source license.

Acknowledgements

We thank Zack Verham for authoring some shaders, Ning Yu for helping produce the supplementary video, and Francesco Di Plinio for providing references about the heat equation and its Taylor expansion. This project was partially funded by NSF grants HCC 1011444 and SHF 1619123.

References

[AGB00] APODACA A. A., GRITZ L., BARZEL R.: *Advanced RenderMan: Creating CGI for motion pictures*. Morgan Kaufmann, 2000. 3

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

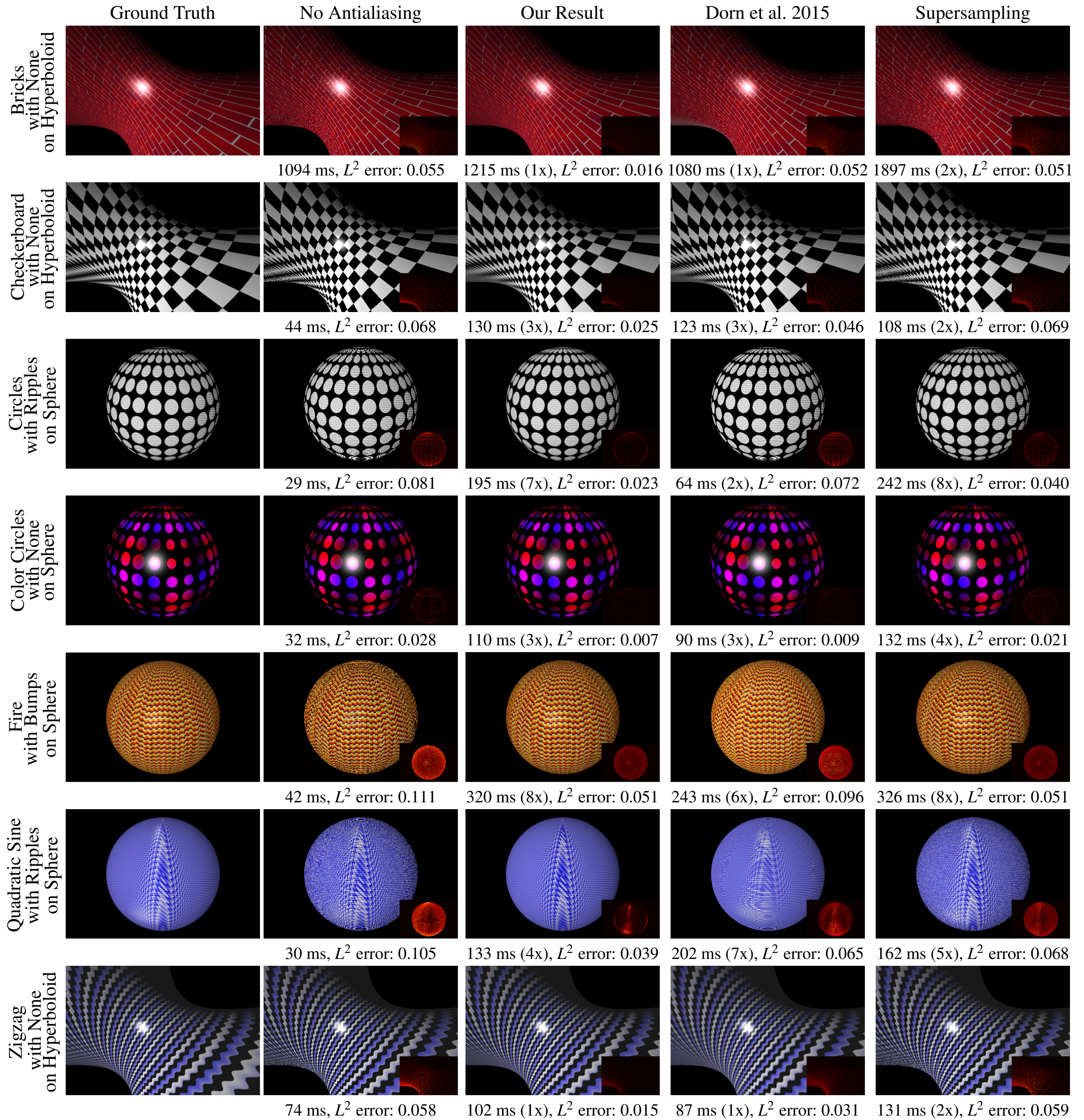


Figure 7: *Selected result images for 7 shaders on curved geometries. Please see the supplemental video for these shaders with a moving camera. Reported below each shader are time to render a frame, time relative to no antialiasing, and L^2 error.*

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-time rendering*. CRC Press, 2008. 1

[BCM05] BUADES A., COLL B., MOREL J.-M.: A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on* (2005), vol. 2, IEEE, pp. 60–65. 7

[BCM11] BUADES A., COLL B., MOREL J.-M.: Non-local means denoising. *Image Processing On Line 1* (2011), 208–212. 7

[BLPW14] BRADY A., LAWRENCE J., PEERS P., WEIMER W.: genbrdf: Discovering new analytic brdfs with genetic programming. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 114. 3

[BS03] BAKER M., SUTLIEF S.: Green's functions in physics version 1. 5

[BVM*17] BAKO S., VOGELS T., MCWILLIAMS B., MEYER M., NOVÁK J., HARVILL A., SEN P., DEROSE T., ROUSSELLE F.: Kernel-predicting convolutional networks for denoising monte carlo renderings. *ACM Transactions on Graphics (TOG) 36*, 4 (July 2017). 7

[CC99] CHEN B., CHEN X.: A global and local superlinear continuation-smoothing method for p 0 and r 0 ncp or monotone ncp. *SIAM Journal on Optimization 9*, 3 (1999), 624–645. 2

[Coo86] COOK R. L.: Stochastic sampling in computer graphics. *ACM Trans. Graph. 5*, 1 (Jan. 1986), 51–72. URL: http://doi.acm.org/10.1145/7529.8927, doi:10.1145/7529.8927. 6

[Cro77] CROW F. C.: The aliasing problem in computer-generated shaded images. *Communications of the ACM 20*, 11 (1977). 1, 3

[Cro84] CROW F. C.: Summed-area tables for texture mapping. *ACM SIGGRAPH computer graphics 18*, 3 (1984), 207–212. 2, 3

[CSL11] CHAUDHURI S., SOLAR-LEZAMA A.: Smoothing a program soundly and robustly. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 277–292. 2

[CX99] CHEN B., XIU N.: A global linear and local quadratic noninterior continuation method for nonlinear complementarity problems based on chen–mangasarian smoothing functions. *SIAM Journal on Optimization 9*, 3 (1999), 605–623. 2

[DBLW15] DORN J., BARNES C., LAWRENCE J., WEIMER W.: Towards automatic band-limited procedural shaders. In *Computer Graphics Forum* (2015), vol. 34, Wiley. 1, 2, 3, 4, 5, 7, 8, 9, 10

[DW] DIPPÉ M. A., WOLD E. H.: Antialiasing through stochastic sampling. *ACM Siggraph Computer Graphics 19*, 3. 2, 3, 6

[Ebe03] EBERT D. S.: *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003. 3

[EN97] ERMOLIEV Y. M., NORKIN V. I.: On nonsmooth and discontinuous problems of stochastic systems optimization. *European Journal of Operational Research 101*, 2 (1997), 230–244. 2

[ENW95] ERMOLIEV Y. M., NORKIN V. I., WETS R. J.: The minimization of semicontinuous functions: mollifier subgradients. *SIAM Journal on Control and Optimization 33*, 1 (1995), 149–167. 2

[Fei11] FEIGUIN A.: Monte carlo error analysis, 2011. [Online; accessed 22-May-2017]. URL: https://www.northeastern.edu/afeiguin/phys5870/phys5870/node71.html. 6

[Hec86] HECKBERT P. S.: Filtering by repeated integration. In *ACM SIGGRAPH Computer Graphics* (1986), vol. 20, ACM, pp. 315–321. 7

[HFTF15] HE Y., FOLEY T., TATARCHUK N., FATAHALIAN K.: A system for rapid, automatic shader level-of-detail. *ACM Transactions on Graphics (TOG) 34*, 6 (2015), 187. 3

[HJW*08] HACHISUKA T., JAROSZ W., WEISTROFFER R. P., DALE K., HUMPHREYS G., ZWICKER M., JENSEN H. W.: Multidimensional adaptive sampling and reconstruction for ray tracing. In *ACM Transactions on Graphics (TOG)* (2008), vol. 27, ACM, p. 33. 2, 3

[KBS] KALANTARI N. K., BAKO S., SEN P.: A machine learning approach for filtering monte carlo noise. *ACM Trans. Graph. 34*. 7

[Koz92] KOZA J. R.: *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992. 3

[KS] KENSLER A., SHIRLEY P.: Optimizing ray-triangle intersection via automated search. In *Interactive Ray Tracing, IEEE Symp., 2006*. 3

[LLDD09] LAGAE A., LEFEBVRE S., DRETTAKIS G., DUTRÉ P.: Procedural noise using sparse gabor convolution. In *ACM Transactions on Graphics (TOG)* (2009), vol. 28, ACM, p. 54. 2

[LWC*08] LIU Y.-L., WANG J., CHEN X., GUO Y.-W., PENG Q.-S.: A robust and fast non-local means algorithm for image denoising. *Journal of computer science and technology 23*, 2 (2008), 270–279. 7

[Łys12] ŁYSIK G.: Mean-value properties of real analytic functions. *Archiv der Mathematik 98*, 1 (2012), 61–70. 3, 5

[Mita] MITCHELL D. P.: Generating antialiased images at low sampling densities. In *ACM SIGGRAPH 1987*, vol. 21. 2, 3

[Mitb] MITCHELL D. P.: Spectrally optimal sampling for distribution ray tracing. In *ACM SIGGRAPH 1991*, vol. 25. 2, 3

[Moo79] MOORE R. E.: *Methods and applications of interval analysis*. SIAM, 1979. 2

[Nes05] NESTEROV Y.: Smooth minimization of non-smooth functions. *Mathematical programming 103*, 1 (2005), 127–152. 2

[NRS82] NORTON A., ROCKWOOD A. P., SKOLMOSKI P. T.: Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. In *ACM SIGGRAPH* (1982), vol. 16, ACM, pp. 1–8. 2, 3

[OKS] OLANO M., KUEHNE B., SIMMONS M.: Automatic shader level of detail. In *Proceedings of 2003 ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*. 3

[Pel] PELLACINI F.: User-configurable automatic shader simplification. In *ACM Transactions on Graphics 2005*, vol. 24. 3

[PP*08] PETERSEN K. B., PEDERSEN M. S., ET AL.: The matrix cookbook. *Technical University of Denmark 7* (2008), 15. 6

[RKZ] ROUSSELLE F., KNAUS C., ZWICKER M.: Adaptive rendering with non-local means filtering. *ACM TOG 2012, vol. 31, 6*. 7

[SAMWL11] SITTHI-AMORN P., MODLY N., WEIMER W., LAWRENCE J.: Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG) 30*, 6 (2011), 152. 3, 7

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement mapping on the gpu — state of the art. In *Computer Graphics Forum* (2008), vol. 27, Wiley Online Library, pp. 1567–1592. 8

[So08] SO S.: Why is the sample variance a biased estimator? *Griffith University, Tech. Rep., 09* (2008). 6

[Wel86] WELLS W. M.: Efficient synthesis of gaussian filters by cascaded uniform filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2 (1986), 234–239. 7

[Wik17] WIKIPEDIA: Entire function, 2017. [Online; accessed 2017-05-20]. URL: http://bit.ly/2DLtkm9. 5

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *Acm siggraph computer graphics* (1983), vol. 17, ACM, pp. 1–11. 2, 3

[Wu] WU Z.: The effective energy transformation scheme as a special continuation approach to global optimization with application to molecular conformation. *SIAM Journal on Optimization, 1996 6*, 3. 2

[WYY*14] WANG R., YANG X., YUAN Y., CHEN W., BALA K., BAO H.: Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics (TOG) 33*, 6 (2014), 226. 3

[YNS*09] YANG L., NEHAB D., SANDER P. V., SITTHI-AMORN P., LAWRENCE J., HOPPE H.: Amortized supersampling. In *ACM Transactions on Graphics (TOG)* (2009), vol. 28, ACM, p. 135. 3

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.