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

YUTING YANG, University of Virginia
CONNELLY BARNES, University of Virginia

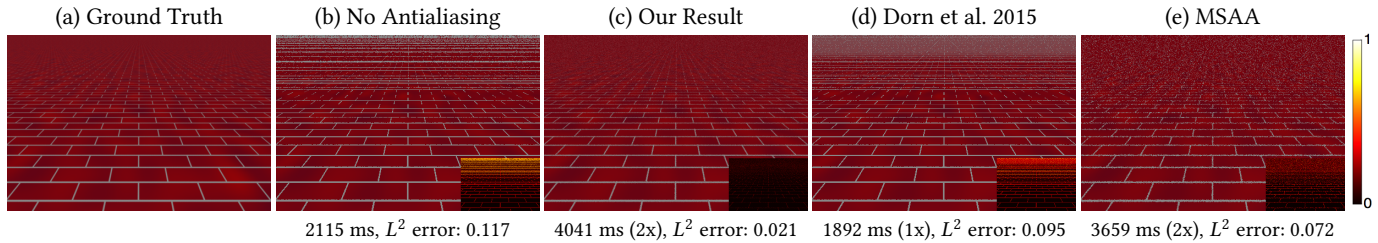| (a) Ground Truth	(b) No Antialiasing	(c) Our Result	(d) Dorn et al. 2015	(e) MSAA
	2115 ms, L^2 error: 0.117	4041 ms (2x), L^2 error: 0.021	1892 ms (1x), L^2 error: 0.095	3659 ms (2x), L^2 error: 0.072

Fig. 1. Our paper gives a novel compiler framework for smoothing out programs in an approximate manner. 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 multi-sample antialiasing (MSAA), 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.

This paper introduces a general method to approximate the convolution of an arbitrary program with a Gaussian kernel. This process has the effect of smoothing out a program. Our compiler framework models intermediate values in the program as random variables, by using mean and variance statistics. Our approach breaks the input program into parts and relates the statistics of the different parts, under the smoothing process. We give several approximations that can be used for the different parts of the program. These include the approximation of Dorn et al. [2015], a novel adaptive Gaussian approximation, Monte Carlo sampling, and compactly supported kernels. Our adaptive Gaussian approximation is accurate up to the second order in the standard deviation of the smoothing kernel, and mathematically smooth. We show how to construct a compiler that applies chosen approximations to given parts of the input program. Because each expression 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 complex shaders, including shaders with parallax mapping, animation, and spatially varying statistics. The resulting smoothed shader programs outperform previous approaches both numerically, and aesthetically, due to the smoothing properties of our approximations.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Rendering**;

Additional Key Words and Phrases: Compilers, rendering, shaders

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 out 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 [Akenine-Möller et al. 2008]. One visual error that can appear in procedural shaders is *aliasing*. *Aliasing* artifacts occur when the sampling rate is below the Nyquist limit [Crow 1977]. There are two more conventional approaches used to reduce such aliasing: supersampling and prefiltering. We discuss these before discussing our smoothing compiler.

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 [Dippé and Wold 1985; Hachisuka et al. 2008; Mitchell 1987, 1991]. 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 [Williams 1983] or summed area tables [Crow 1984]. This approach offers the benefit of exact solutions with a constant number of operations, provided that the shading function can be 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 [Norton et al.

1982]. Exact analytic band-limited formulas are known for some specialized functions such as noise functions [Lagae et al. 2009]. In most cases, however, the shader developer must manually calculate the convolution integral. But frequently the integrals cannot be solved in closed form, which limits this strategy.

Our framework takes a different approach from most previous work. Our goal is to smooth out an arbitrary input function represented as a program, by approximately convolving it with a Gaussian filter. We take the program as input, break it into different parts, and relate the statistics of the different 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. In this manner, we can smooth out 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.

In our framework, we explore a number of approximation rules. We first improve the approximations of Dorn et al. [2015] (Section 4.3) and relate them to the mean and variance statistics in our framework. We then develop a novel adaptive Gaussian approximation (Section 4.4). For a class of programs that are real analytic, this approximation if used at all nodes in the computation results in a smoothed program that is accurate to the second power of the standard deviation. We next relate Monte Carlo sampling (Section 4.5) to our framework. This can give good approximations for non-analytic functions, because it converges for large numbers of samples to the ground truth. Finally, we discuss how compactly supported kernels (Section 4.6) can be used for parts of the computation that would otherwise be undefined or interact with infinite values.

Because each computation node can choose from a variety of approximation methods, the search space for the optimal approximations is combinatoric. We use genetic search to find the *Pareto frontier* of approximation choices that optimally trade off the running time and error of the program. A programmer can then simply select a program variant from the Pareto frontier according to the desired running time or error level.

To evaluate our framework, we developed a variety of complex shaders, including shaders with parallax mapping, animation, and spatially varying statistics, and compare the performance with Dorn et al. [2015] and commonly used multisample antialiasing (MSAA). Our framework gives a wider selection of band-limited programs with less error than Dorn et al. [2015]. Our shaders are frequently an order of magnitude faster than MSAA for comparable errors.

2 RELATED WORK

Mathematics and smoothing. Smoothing a function is beneficial in domains such as optimizing non-convex, or non-differentiable problems [Chen and Chen 1999; Chen and Xiu 1999; Nesterov 2005]. For our purposes, smoothing can be understood as convolving a function with smooth kernels. When used in numerical optimization, this approach is sometimes known as the continuation method or mollification [Ermoliev and Norkin 1997; Ermoliev et al. 1995;

Wu 1996]. The amount of smoothing can be controlled simply by changing the width of the kernel. In our framework, we model the input program by relating the statistics of each variable in the program, 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 [Moore 1979]. Chaudhuri and Solar-Lezama [2011] developed a 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 the Gaussian that is the Green's function of the heat equation. Thus, connections can be made between convolution with a Gaussian and results for the heat equation, such as Łysik [2012].

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 [Apodaca et al. 2000]. Stochastic sampling [Crow 1977; Dippé and Wold 1985] 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 [Dippé and Wold 1985; Hachisuka et al. 2008; Mitchell 1987, 1991]. In video rendering, samples from previous frames can also be reused for computation efficiency [Yang et al. 2009]. 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 [Ebert 2003] that approximate the band-limited shader functions by sampling. This includes clamping the high-frequency components in the frequency domain [Norton et al. 1982], and producing lookup tables for static textures using mipmapping [Williams 1983] and summed area tables [Crow 1984]. Like our work, and unlike most other work in this area, Dorn et al. [2015] uses a compiler-driven technique to apply closed-form integral approximations to compute nodes of an arbitrary program. Unlike Dorn et al. [2015], our framework flexibly incorporates both mean and variance statistics, and we use several approximations that have higher accuracy. Our approach is general and can apply to arbitrary programs: we simply explore shaders as an example application.

Genetic algorithms. 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 [Koza 1992]. In computer graphics, recent work by Sitthi-Amorn et al. [2011] describes a GP approach to the problem of automatic procedural shader simplification. Brady and colleagues [Brady et al. 2014] showed how to use GP to discover new analytic reflectance functions. We use a similar approach as [Sitthi-Amorn et al. 2011] to automatically generate the Pareto frontier of approximated smoothed functions.

3 OVERVIEW

This section gives an overview of our system. We first discuss the goal of our smoothing process. Next, we give an overview of the key assumptions and components in our framework.

The goal for our framework is to take an arbitrary program and produce a smoothed output program which closely approximates

the convolution of the input program with a Gaussian kernel. This convolution could be multidimensional: for shader programs, the dimension is typically 2D for spatial coordinates. When producing such approximations, we would also like to keep high computation efficiency.

In our compiler-based framework, we assume the input program has a compute graph, where each node represents a floating-point computation, and the graph is a directed acyclic graph (DAG). We label each node in the DAG as a random variable and compute its mean and variance statistics. 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.[1] We assume the input variables have specified mean and variance, and for simplicity assume they are independent. For example, in a shader, the input variables might be the (x, y) pixel coordinate of the shader, and the output might be the color. We carry mean and variance computations forward through the compute graph, and collect the output by taking the mean value of the output variable.

We now give an brief conceptual example of how random variables can be used to collect statistics associated with values in a program. Suppose that a part of an input program contains a function application $y = f(x)$, where both x and y are intermediate values, and f is a built-in mathematical function (e.g. cosine). In our framework, we model this computation as $Y = f(X)$. Here, X and Y are random variables with mean and variance statistics. In our framework, we simplify the problem of smoothing out the overall function by breaking the program into sub-parts, which can each be separately smoothed. In this paper, we use lower-case letters such as x and y to represent real values (scalars) in the original non-smoothed input function. These can be either input, output, or intermediate values. We use corresponding capital letters such as X and Y to represent random variables for the corresponding smoothed computation node.

As we just mentioned, the random variables in our framework have two key values associated with each individual: mean and variance. We denote these as μ_X and σ_X^2 respectively.

An overview of the components of this framework is shown in Figure 2. Our framework has three main parts: approximation rules, quality improvements, and genetic search. Under the random variable assumption, the compiler first approximates the smoothed function by determining the mean and variances of each compute node (discussed in Section 4 - 4.6). Next, additional quality improvements can be optionally made. These include heuristically adjusting the variance of each node using simplex search, and denoising (discussed in Section 4.7). Finally, we use genetic search to select the best-performing program variants based on the criterion of fast run-time and low error (discussed in Section 5).

As shown in Figure 3, we implemented several different approximation methods to calculate the mean and variance of each random variable. These approximations are:

- Dorn et al. [2015] (Section 4.3): we improve the approximations described in [Dorn et al. 2015] and relate them to the mean and variance of our framework. The variance calculation here is naive yet fast.
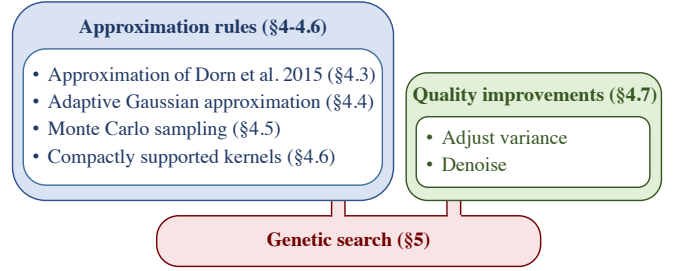

Fig. 2. An overview of our system showing its components. The key components are: approximation rules, quality improvements, and genetic search. In approximation rules (Section 4 - 4.6), a variety of approximation methods are implemented to smooth the input function. We can optionally improve the quality of the output program by using the simplex method to adjust variances and denoising the output (Section 4.7). All smoothed program variants are selected through a genetic search (Section 5), which finds a Pareto frontier that optimally trades off program running time and error.

- Adaptive Gaussian approximations (Section 4.4): these use convolution to estimate both the mean and variance for a given compute stage assuming Gaussian distributed inputs and outputs. When applied to a whole program that is in a certain real analytic class, this approximation rule gives accuracy that is accurate to the second power of σ, the standard deviation of the input variables.
- Monte Carlo sampling: we integrate this widely used method into our framework. The mean and variance are given by sampled estimators. For large number of samples, this converges to the ground truth.
- Compactly supported kernels: for a sub-part of the computation that contains undefined or infinite values, (e.g. $y = f(x) = 1/x$ for $x = 0$), the corresponding integrals with the Gaussian for mean and variance may not exist. However, the full program may have a well-defined result, so smoothing should still be possible through such a sub-part. To handle this case, we use compactly supported kernels such as box or tent. The kernel size is limited based on the distance to the nearest undefined point.

In Figure 3(b-e), we show an example of using these different approximations to smooth the function $y = \sin(x^2)$. The approximations are shown as orange curves and the ground truth as blue dashed curves. Specifically, we have considered this function as the composition of two primitive functions $\sin()$ and x^2, which have each been atomically smoothed. The adaptive Gaussian rule in Figure 3(c) gives a close approximation to the ground truth for small σ. For demonstration purposes, in Figure 3(e) we also use $y = \sin(x^2)$ to show the approximation for compactly supported kernels. But in our implementation, only functions with undefined regions, or those that do not have a closed form Gaussian convolution will be approximated using this method.

---

[1]Except for Monte Carlo sampling (Section 4.5), which of course is sampled.

(a) Input function   (b) [Dorn et al. 2015],   (c) Adaptive Gaussian,   (d) Monte Carlo sampling   (e) Compactly supported kernels
                        (Section 4.3)   (Section 4.4)   (Section 4.5)   (Section 4.6)
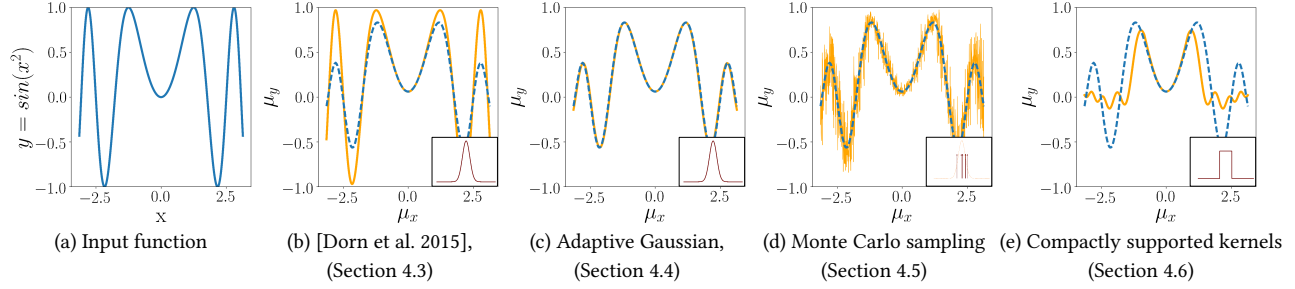
Fig. 3. An overview of different approximation techniques. (a) The input function $y = f(x) = \sin(x^2)$. This function is evaluated in our framework as the composition of two "atomic" functions that we do know how to smooth: $\sin()$ and x^2. The "ground truth" correctly smoothed function $\hat{f}(x)$ is shown in blue dashed curves in subfigures (b-e). This is determined by sampling at a very high sample rate. The orange lines in subfigures (b-e) are the mean of the output random variable calculated by different approximations. 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. [2015] (Section 4.3); (c) Our adaptive Gaussian approximation (Section 4.4); (d) Monte Carlo sampling approximation with 8 samples (Section 4.5); (e) Compactly supported kernels approximation: here we use a box kernel (Section 4.6). We use a standard deviation of $\sigma = 0.25$ for all input distributions.

4 APPROXIMATION RULES

Suppose that the programmer writes down a given input program that we wish to smooth. In our framework, we carry out smoothing by conceptually replacing each intermediate float value in the computation with a random variable having a known probability density function. We thus represent each compute node by having instead of a specific value, having specified mean and standard deviation statistics. For each node X in the computation, we use μ_X to denote its mean, and σ_X for its standard deviation. The output of the program is then the mean value for the output variable.

As a concrete example, for shader bandlimiting, each input pixel coordinate (x, y) could be regarded as two independent random variables, X and Y. The means of these random variables could represent pixel positions, $\mu_X = x$, and $\mu_Y = y$. Because we wish to evaluate an antialiased variant of the shader, we can model the standard deviations of the inputs as $\sigma_X = \sigma_Y = 0.5$, i.e. half a pixel, to suppress aliasing beyond the Nyquist rate. Then the mean of the output variables in the computation is simply used as the rendered color.

In this section, we will first use an example to illustrate how we use this approach to smooth functions. Then, we will describe how our composition rules can be used to combine different smoothed sub-parts of a program. After that, we will talk about different approximation rules to estimate the mean and standard deviation statistics.

4.1 Motivating example

Assume we are smoothing a function $y = f(x)$. The smoothed function can be denoted as $\hat{f}$, and the output value can be computed as $y = \hat{f}(x, \sigma^2)$. Here, σ controls the smoothness. Higher σ means $\hat{f}$ is more smooth. We will use the $\hat{}$ operator throughout this paper to denote that a function is being smoothed. We use convolution to define $\hat{f}(x, \sigma^2)$ as follows.

$$\hat{f}(x, \sigma^2) = f(x) * G(x, \sigma^2)$$
$$= \int_{-\infty}^{\infty} f(x - u) \cdot G(u, \sigma^2) du \qquad (4.1)$$

In equation (4.1), $G(x, \sigma^2)$ is the smoothing kernel that is used to smooth out the original function $f(x)$. To more explicitly identify the kernel as being G, we can also use the notation $\hat{f}^G(x, \sigma^2)$. The convolution kernel $G(x, \sigma^2)$ can be any non-negative kernel that has an integral over x from $-\infty$ to ∞ of 1. This allows us to interpret the kernel also as a probability density function. For example, we might use the Gaussian kernel:

$$G(x, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right) \qquad (4.2)$$

If $f(x)$ happens to be a shader program, then as is discussed in [Dorn et al. 2015], $\hat{f}(x, \sigma^2)$ is simply a band-limited version of the same procedural shader function.

We now show how the convolution from equation (4.1) fits into our framework. We assume that in the input program, an intermediate value Y is computed by applying a function to a previous value X, i.e., $Y = f(X)$. But in our framework, both X and Y are random variables. If the probability density function of X is f_X, then μ_Y can be computed from μ_X and σ_X as:

$$\mu_Y = E[f(X)] = \int_{-\infty}^{\infty} f(u) \cdot f_X(u) du \qquad (4.3)$$

As an example, if we assume X is normally distributed, $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$, then equation (4.3) can be rewritten as:

$$\mu_Y = \int_{-\infty}^{\infty} f(u) \cdot \frac{1}{\sqrt{2\pi}\sigma_X} \exp\left(-\frac{(u - \mu_X)^2}{2\sigma_X^2}\right) du$$
$$= f(\mu_X) * G(\mu_X, \sigma_X^2) \qquad (4.4)$$
$$= \hat{f}(\mu_X, \sigma_X^2)$$

In equation (4.4), $*$ indicates convolution. If X has a different probability density function, then $G(x, \sigma^2)$ will be a different kernel. Thus, μ_Y is the smoothed value for $f(X)$. This gives some intuition for how our model is used for smoothing functions. Our framework provides different methods to estimate μ and σ. We will describe them in the following subsections.

4.2 Composition rules

In principle equation (4.1) can be used to determine the correct smoothed function $\hat{f}$ from any input function f. However, in practice, the associated integrals often do not have a closed-form solution. Therefore, a key observation used in our paper is that we can break up the computation into different sub-parts, and compute approximate mean and standard deviation statistics for each sub-part. We do this by simply substituting the smoothed mean and standard deviation that are output from one compute node as the inputs for any subsequent compute nodes.

As an example, suppose we have the computation of Figure 3. From the input x_0, we compute $x_1 = x_0^2$, and then the output $x_2 = \sin(x_1)$, with associated random variables X_0, X_1, X_2, respectively. We are given μ_{X_0}, σ_{X_0}, the mean and standard deviation of the input. Using the approximation rule chosen for the node X_1, we compute μ_{X_1}, σ_{X_1} from μ_{X_0}, σ_{X_0}. Then from μ_{X_1}, σ_{X_1}, using the approximation rule for the node X_2, we compute μ_{X_2}, σ_{X_2}: μ_{X_2} is the smoothed output of the program (σ_{X_2} can be discarded).

4.3 Approximation of Dorn et al. 2015

We integrate the approximation methods described in Dorn et al. [2015] as one of our approximation options. Dorn's method involves computing the mean for a smoothed function by convolving with a Gaussian kernel. Suppose an intermediate variable y is computed from another variable x, and the associated random variables are Y and X, respectively, where $Y = f(X)$. Then μ_Y is:

$$\mu_Y = \hat{f}(\mu_X, \sigma_X^2) \tag{4.5}$$

This is the same as the result we derived in equation (4.4). Here $\hat{f}(x, \sigma_X^2)$ is computed from equation (4.1). In Table 3, we show commonly used functions f and their corresponding smoothed functions $\hat{f}$. In Dorn's method, 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. [2015], and use the improved variant of this method 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. [2015]), 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-d), where the initial program found by the genetic search is quite different from the final program after the variances have been adjusted by a simplex search. The adjustment process is described later in more detail in Section 4.7.

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.

4.4 Adaptive Gaussian Approximations

In this novel approximation, we model the input variables to a compute node as being Gaussian distributed, and then fit a Gaussian to the produced output by collecting its mean and standard deviation. Thus, this rule more accurately and adaptively calculates standard deviations.

We first consider the case that we have a function of one variable. In the same manner as Section 4.3, if a random variable Y is computed from another random variable X as $Y = f(X)$, then μ_Y can be determined from equation (4.4), equation (4.1) and Table 3. However, the standard deviation σ_Y is determined differently based on the definition of variance of Y:

$$\begin{aligned}\sigma_Y^2 &= E[Y^2] - E[Y]^2 \\ &= \widehat{f^2}(\mu_X, \sigma_X^2) - \hat{f}^2(\mu_X, \sigma_X^2)\end{aligned} \tag{4.6}$$

The approximation of $\hat{f}$ can also be extended to multiple dimensions. Suppose $f(\mathbf{x})$ is a function applied to an n-dimensional vector. Then $\hat{f}$ can be computed by the convolution of $f(\mathbf{x})$ and an n-dimensional multivariate Gaussian with zero mean and covariance matrix Σ. Suppose for simplicity that the input variables have zero correlation ($\rho = 0$) and equal standard deviation, so $\Sigma = \mathbf{I}\sigma^2$, where $\mathbf{I}$ is the identity matrix. By using Green's function on this convolution [Baker and Sutlief 2003], we can find a Taylor expansion for the function $\hat{f}(\mathbf{x}, \sigma^2)$ in terms of $f(\mathbf{x})$:

$$\hat{f}(\mathbf{x}, \sigma^2) = f(\mathbf{x}) + \frac{1}{2}\sigma^2\nabla^2 f(\mathbf{x}) + \frac{1}{(2!)2^2}\sigma^4\nabla^4 f(\mathbf{x}) + \frac{1}{(3!)2^3}\sigma^6\nabla^6 f(\mathbf{x}) + \dots \tag{4.7}$$

The derivation of equation (4.7) 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 [Wikipedia 2017c]. This class of functions includes polynomials, sines, cosines, and compositions of these. It is also necessary to assume that the function is bounded by exponentials: the precise conditions are discussed by Łysik [2012]. These properties could hold for some shader programs, but even if they do not hold for an entire program, they could often hold for sub-parts of a program. We show in Appendix D that a single function composition gives a result accurate to second order in σ for this rule. Similarly, this property can be proved via induction for multiple function compositions.

There are also other second order accurate approximations, such as simply truncating the Taylor expansion in equation (4.7) to use only the first and second term. Why would we bother to propose an adaptive Gaussian approximation?

To illustrate why adaptive Gaussian gives a more accurate approximation, we show an example in Figure 4. Here, we use the function $f(x) = \sin(x^2) + \frac{1}{100}x^2$. We show the approximation using adaptive Gaussian and simply truncating the Taylor expansion in equation (4.7). In Figure 4, the blue lines represent the ground truth, and the orange lines represent different approximations. From Figure 4(c), we can see that simply truncating the Taylor expansion results in large extrapolation errors and actually amplifies high frequencies, instead of smoothing them out. In Figure 4(b), we show that the function is smoothed more accurately using the adaptive Gaussian approximation.

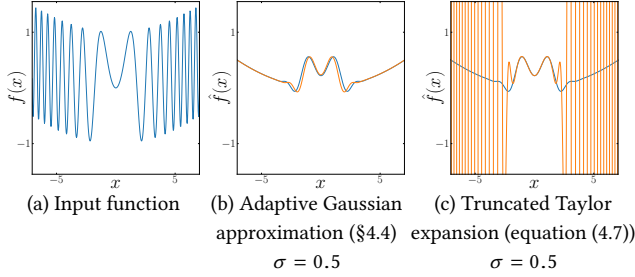| (a) Input function | (b) Adaptive Gaussian approximation (§4.4) $\sigma = 0.5$ | (c) Truncated Taylor expansion (equation (4.7)) $\sigma = 0.5$ |

Fig. 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 4.4) 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.

For binary functions, we can still show that adaptive Gaussian is accurate to second order if a correlation term is carefully chosen. But for brevity, we are only going to show the rules to compute the mean and standard deviation for binary functions we used in our compiler.

In general, the inputs of a binary function $f(a, b)$ can be considered as two random variables A and B (corresponding to a and b). We make the assumption that A and B are distributed according to a bivariate Gaussian:

$$A, B \sim \mathcal{N}\left(\begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}, \begin{bmatrix} \sigma_A^2 & \rho\sigma_A\sigma_B \\ \rho\sigma_A\sigma_B & \sigma_B^2 \end{bmatrix}\right) \quad (4.8)$$

Here, σ_A and σ_B are standard deviations of A and B. These can be determined directly from the approximation of previous computation. 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 binary function plus (+), minus (−) and multiplication (·) can be derived from these assumptions based on properties of the Gaussian distribution [Petersen et al. 2008]:

$$\mu_{\text{plus}} = \mu_A + \mu_B$$
$$\sigma_{\text{plus}}^2 = \sigma_A^2 + \sigma_B^2 + 2\rho\sigma_A\sigma_B$$

$$\mu_{\text{minus}} = \mu_A - \mu_B$$
$$\sigma_{\text{minus}}^2 = \sigma_A^2 + \sigma_B^2 - 2\rho\sigma_A\sigma_B \quad (4.9)$$

$$\mu_{\text{mul}} = \mu_A\mu_B + \rho\sigma_A\sigma_B$$
$$\sigma_{\text{mul}}^2 = \mu_A^2\sigma_B + \sigma_A\mu_B^2 + 2\rho\mu_A\mu_B\sigma_A\sigma_B + \sigma_A^2\sigma_B^2(1 + \rho^2)$$

For the binary function divide $f_{\text{div}}(a, b) = \frac{a}{b}$, we reduce this to multiplication by using (·) as $f_{\text{div}}(a, b) = f_{\text{mul}}(a, b^{-1})$. The mean and standard deviation for division can 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 4.6. The modulo function, $f_{\text{mod}}(a, b) = a\%b$, can be rewritten as $f_{\text{mod}}(a, b) = b \cdot \text{fract}(\frac{a}{b})$. Here, $\text{fract}(x)$ is the fractional part of x. We make the simplifying assumption that the second argument b of mod is an ordinary (non-random) variable (so $\sigma_B = 0$), to obtain:

$$\mu_{mod}^2 = \mu_B \cdot \widehat{\text{fract}}(\frac{\mu_A}{\mu_B}, \frac{\sigma_A^2}{\mu_B^2})$$
$$\sigma_{mod} = \mu_B^2 \cdot \widehat{\text{fract}^2}(\frac{\mu_A}{\mu_B}, \frac{\sigma_A^2}{\mu_B^2}) - \mu_{mod}^2 \quad (4.10)$$

Comparison functions ($>, \geq, <, \leq$) are approximated by converting them to univariate functions including the Heaviside step function $H(x)$. As an example, the function greater than ($>$) can be rewritten as $f_>(a, b) = H(a - b)$. This can be approximated using rules described previously.

One other important multi-variate function we approximated is the *select* function. We approximated this in the same manner as Dorn et al. [2015] as a linear interpolation: $select(a, b, c) = a \cdot b + (1 - a) \cdot c$. But in our case we apply the above univariate and binary function approximations to this formula.

As we discussed before, for binary functions, we approximate the input random variables A and B as bivariate Gaussian with correlation coefficient ρ (equation (4.8)). 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 to accuracy that is second order in σ, 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 input of the terms a and b, normalize these gradients, and take the dot product, which recovers ρ. This can be done using reverse mode automatic differentiation.

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, 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 Appendix B the details for the these other choices for correlation coefficients, because they may be more beneficial for other applications, and the second order accuracy property is interesting.

4.5 Monte Carlo Sampling

We relate the Monte Carlo stochastic sampling [Cook 1986; Dippé and Wold 1985] to our framework. Here n is the number of samples. The computation for a node computing f is modeled as $Y = f(X_1, ..., X_m)$, where the output mean and standard deviation are given by sampled estimators as follows:

$$\mu_Y = \frac{1}{n}\sum_{i=1}^{n} f(X_1 + \mathcal{N}_{i,1}\sigma_{X_1}, ..., X_m + \mathcal{N}_{i,m}\sigma_{X_m})$$
$$\sigma_Y^2 = \frac{1}{n}\sum_{i=1}^{n} f^2(X_1 + \mathcal{N}_{i,1}\sigma_{X_1}, ..., X_m + \mathcal{N}_{i,m}\sigma_{X_m}) - \mu_Y^2 \quad (4.11)$$

Here, each $\mathcal{N}_{ij}$ are random numbers independently drawn from normal distribution $\mathcal{N}(0, 1)$. We experimented with applying the Bessel's correction [Wikipedia 2017a] to correct the bias 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. This is mainly because the variance can also be adjusted in the "quality improvement" phase (discussed in Section 4.7).

We choose sample numbers from $(2, 4, 6, 8, 16, 32)$. The approximation converges to the ground truth for large sample numbers, and the output program simplifies to MSAA [Cook 1986] when the entire input program is approximated under Monte Carlo sampling.

The error of the Monte Carlo sampling σ_M is estimated as follows [Feiguin 2011].

$$\sigma_M \approx \frac{\sigma}{\sqrt{n}} \tag{4.12}$$

Here, σ is the standard deviation computed from equation (4.11) and n is sample number. This error estimate becomes more accurate in the limit of large sample numbers.

4.6 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 equation (4.1) does not exist. Monte Carlo sampling may also encounter such problem. 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 [Heckbert 1986] of boxcar functions. This is because such kernels approximate the Gaussian by the central limit theorem [Wells 1986]. 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 Table 3, we list the smoothed result using the box kernel for commonly used functions.

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 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 fract(x) (the fractional part of x), which have discontinuities. We observe that 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(), then we find that the output $Y = \text{fract}(X)$ may be bimodal, with some values close to zero (due to x being a positive value), and other values close to one (due to x being negative with small absolute value). If we fit a Gaussian to this resulting bimodal distribution, as our adaptive Gaussian rule proposes, then the mean

would be $\frac{1}{2}$, even though most of the values of Y are either near 0 or 1. This may result in a poor approximation, which can show up in tiled pattern shaders (which use fract) as an improper 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 in the result, which we wish to avoid. Therefore, our solution in practice is to first identify whether the output distribution is bimodal: for fract() we can do this by simply checking if input distribution when represented as a uniform (box) distribution contains exactly one discontinuity, i.e. one integer. If so, we simply truncate the filter at the location of the discontinuity.

4.7 Quality Improvements

At this point, we assume we have now applied the approximation rules described in Sections 4.3 through 4.6 to an input program. We can optionally improve the approximation quality in two ways: a) adjust the standard deviation made in the approximations, and b) apply denoising to program variants that use Monte Carlo sampling.

Because our approximations are not exact, the standard deviation of some nodes may be too high or too low. Following [Dorn et al. 2015], we learn coefficients to refine the standard deviation estimates. By comparing with the ground truth image for the shader rendering, we use the Nelder-Mead simplex search [Nelder and Mead 1965] to learn multiplicative factors for standard deviations.

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 [Bako et al. 2017; Kalantari et al. 2015; Rousselle et al. 2012]. We implement the non-local means denoising method [Buades et al. 2005, 2011] with Laplacian pyramid [Liu et al. 2008]. We find that aesthetically appealing denoising results can be 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.

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 gives users the opportunity to explore the trade-off between efficiency and accuracy of the smoothed program. Although developers can manually assign approximation rules to each node, we found this to be a time-consuming process that can easily overlook beneficial approximation combinations. This is because each individual computation node may choose from several approximation rules, and the search space for this is combinatoric.

Our genetic search closely follows the method of Sitthi-Amron et al. [2011]. 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 method, we create a program where the rule is applied to all the expression nodes. For such initial guesses, we also apply cross-over with a probability of 0.5. Then we employ standard mutation and cross-over 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. In the genetic search algorithm, we choose our probability of crossover as 0.4, the probability of retaining elite individuals to be 0.25, and the mutation probability to be 0.35. Also, we use a tournament size of 4, and population size of 40, with 20 generations. Finally, we run 3 random restarts for the algorithm.

For the Monte Carlo sampling approximation, during initialization and mutate, we select sample counts with equal probability from the set $\{2, 4, 8, 16, 32\}$. For the determination of correlation coefficients described in Section 4.4, we pick with equal probability one of the three options.

6 EVALUATION

The previous work of Dorn et al. [2015] was evaluated primarily on relatively simple shaders. To provide a more challenging and realistic benchmark, we authored 21 shaders. Unlike the simple shaders of Dorn et al., these include shaders that have a Phong lighting model, animation, spatially varying statistics, and which include parallax mapping [Szirmay-Kalos and Umenhoffer 2008]. 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.

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. [2015] 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 MSAA result was selected based on evaluating MSAA 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 MSAA 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 five choices for approximation are selected for different programs. For the correlation term discussed in Section 4.4, when aggregated

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 used to construct the shader program, and the number of scalar expressions in the program's compute graph.

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 sinewave of quadratic (non-stationary)
Zigzag	24	224	Colorful zigzag pattern
Parallax mappings			
None	0	0	No parallax mapping
Bumps	21	203	Spherical bumps
Ripples	23	178	Animating ripples

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 to second order accuracy. We conclude that for shader programs, the simple choice of $\rho = 0$ in most cases suffices.

Note that our brick shader gives poor results for the method of Dorn et al. [2015], while in that paper, a brick shader with similar appearance shows good results. This is because the brick shader in Dorn et al. [2015] 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. [2015], but our method can handle such shaders.

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 3 hours of wall clock time. However, we note that good program variants are available after just a few minutes, and most of the remaining tuning time is spent making slight improvements to the best individuals. Also, our tuner is intentionally a research prototype that is not particularly optimized: it could be significantly faster if it was parallelized more effectively, cached more redundant computations, or targeted the GPU.

7 DISCUSSION AND CONCLUSION

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

	Ground Truth	No Antialiasing	Our Result	Dorn et al. 2015	MSAA

Checkerboard with Ripples: 30 ms, L^2 error: 0.194 | 54 ms (2x), L^2 error: 0.071 | 50 ms (2x), L^2 error: 0.102 | 47 ms (2x), L^2 error: 0.233

Circles with None: 20 ms, L^2 error: 0.148 | 71 ms (4x), L^2 error: 0.035 | 39 ms (2x), L^2 error: 0.063 | 67 ms (3x), L^2 error: 0.087

Color Circles with Bumps: 37 ms, L^2 error: 0.098 | 149 ms (4x), L^2 error: 0.039 | 56 ms (2x), L^2 error: 0.079 | 112 ms (3x), L^2 error: 0.061

Fire with Bumps: 39 ms, L^2 error: 0.170 | 698 ms (18x), L^2 error: 0.037 | 67 ms (2x), L^2 error: 0.136 | 705 ms (18x), L^2 error: 0.037

Quadratic Sine with Ripples: 38 ms, L^2 error: 0.184 | 81 ms (2x), L^2 error: 0.045 | 59 ms (2x), L^2 error: 0.094 | 99 ms (3x), L^2 error: 0.158

Zigzag with Ripples: 57 ms, L^2 error: 0.139 | 77 ms (1x), L^2 error: 0.045 | 59 ms (1x), L^2 error: 0.072 | 83 ms (1x), L^2 error: 0.122
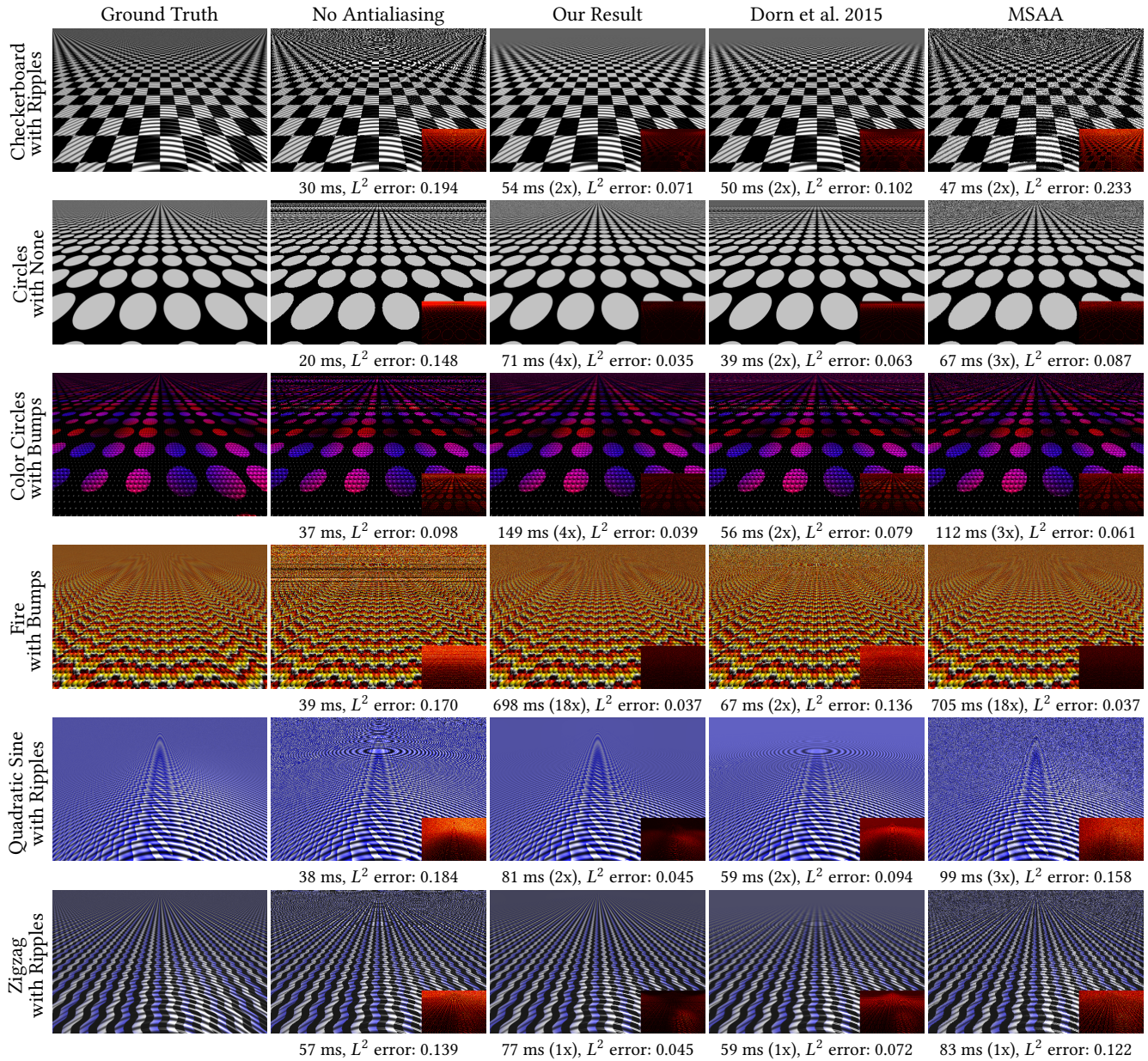
Fig. 5. Selected result images for 6 shaders. Please see the supplemental video for a comprehensive comparison of all shaders. Reported below each shader are the time to render a frame, time relative to no antialiasing, and L^2 error. Please zoom in to see aliasing and noise patterns in the different methods. Program variants with comparable time were selected: see Section 6 for more details. Note that the amount of aliasing and error for our result is significantly less than Dorn et al. [2015]. Note that we typically have significantly less error and noise than the comparable MSAA results.

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. [2015] even after our improvements, and is frequently an order of magnitude faster than multi-sample antialiasing (MSAA). Our framework is quite general, and we believe it could be useful for other problems in graphics and

across the sciences. In order to facilitate reproducible research, we intend to release our source code under an open source license.
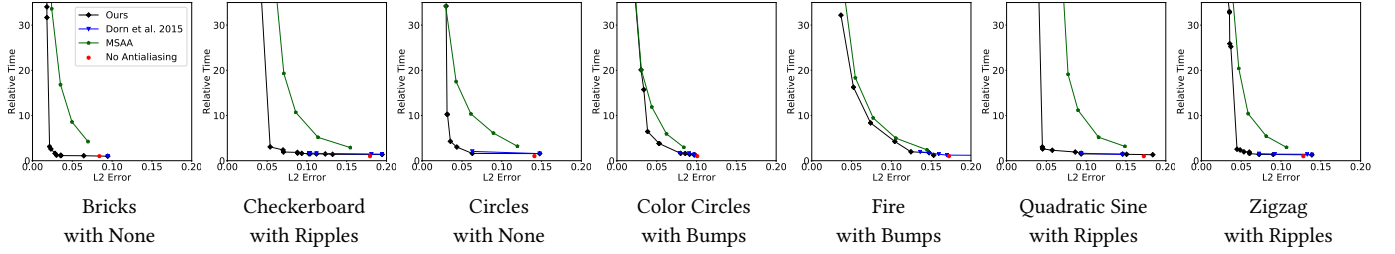
ACKNOWLEDGEMENTS

Fig. 6. Time versus error plots for 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 [2015], MSAA with varying numbers of samples, and the input shader without antialiasing. Note that our approach typically has significantly less error than Dorn et al [2015] and is frequently an order of magnitude faster than MSAA for comparable error.

Table 2. Statistics of which approximations were chosen for different shaders. 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. [2015]	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%

A TABLE OF SMOOTHED FORMULAS

In Table 3, we show a table of functions and their corresponding convolutions with box and Gaussian kernels. These are needed for the approximations we developed in Section 4. This table can be viewed as an extension of the table presented in Dorn et al. [2015]. Note that in particular, for each function $f(x)$, we report not only the result of smoothing $f(x)$ but also smoothing $f^2(x)$ (e.g. if we report $\cos(x)$ then we also report $\cos^2(x)$). This is needed to determine the standard deviations output by a given compute stage for the adaptive Gaussian approximation rule of Section 4.4.

In Table 3, we give that bandlimiting x^n by a Gaussian is a generalized Hermite polynomial $He_n^{[\alpha]}(x)$. This can be derived trivially from a property of generalized Hermite polynomials: the nth non-central moment of a Gaussian distribution X with expected value μ and variance σ is a generalized Hermite polynomial [Wikipedia

2017d]. The ordinary Hermite polynomial is defined as:

$$He_n(x) = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{n!}{(n-2k)!k!}(-2)^{-k}x^{n-2k} \tag{A.1}$$

Then $He_n^{[\alpha]}(x)$ is the generalized Hermite polynomial with parameter α, defined by:

$$He_n^{[\alpha]}(x) = \alpha^{\frac{n}{2}} He_n\left(\frac{x}{\sqrt{\alpha}}\right)$$

$$= \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{n!}{(n-2k)!k!}(-2)^{-k}x^{n-2k}\alpha^k \tag{A.2}$$

B CORRELATION COEFFICIENTS FOR MULTIVARIATE FUNCTIONS

In this appendix, we describe rules to compute the correlation coefficient ρ, which is briefly discussed in Section 4.4. Specifically, we are given a binary function $f(a, b)$, which receives two inputs a and b, with associated random variables A and B, respectively. We discuss the following two rules: a) assume ρ is constant and estimate by sampling and b) compute ρ under the assumption that computations are affine.

Estimate ρ by sampling. In a training stage, we use n samples to approximate ρ of two random variables A and B. The samples drawn from these two distributions are represented as a_i and b_i with corresponding sample mean $\overline{a}$ and $\overline{b}$. Thus, ρ can be estimated by equation (B.1) [Wikipedia 2017b].

$$\rho = \frac{\sum_{i=1}^n (a_i - \overline{a})(b_i - \overline{b})}{\sqrt{\sum_{i=1}^n (a_i - \overline{a})^2}\sqrt{\sum_{i=1}^n (b_i - \overline{b})^2}} \tag{B.1}$$

Estimate ρ by an affine assumption. When we calculate ρ under this rule, we assume the variables a and b input to f are affine transformations of the variables $x_1, ..., x_n$ which are input to the program. Here, x_is are inputs to the function that is being smoothed, which for a shader program could be the (x, y) spatial coordinate. Under this assumption, a and b can be expressed as follows.

Table 3. A table of univariate functions, and their corresponding bandlimited result, using a box kernel B and a Gaussian G. The box kernel is the PDF of the uniform random variable $U[-\sqrt{3}\sigma, \sqrt{3}\sigma]$. The Gaussian kernel is the PDF of the random variable $\mathcal{N}(0, \sigma^2)$. Each random variable has standard deviation σ. We define $\text{sinc}(x) = \sin(x)/x$, and the Heaviside step function $H(x)$ is 0 for $x \leq 0$ and 1 for x positive. Note that functions with undefined regions, such as x^p for negative or fractional p have σ limited as described in Section 4.6.

Function $f(x)$	Bandlimited with box kernel: $\hat{f}^B(x)$	Bandlimited with Gaussian kernel: $\hat{f}^G(x)$
$x^p, p \neq -1$	$\frac{1}{\sqrt{12}\sigma(p+1)}\left[(x+\sqrt{3}\sigma)^{p+1} - (x-\sqrt{3}\sigma)^{p+1}\right]$	$He_p^{[-\sigma^2]}(x)$
x^{-2}	$(x^2 - 3\sigma^2)^{-1}$	
x^{-1}	$\frac{1}{\sqrt{12}\sigma}\log\left|\frac{x+\sqrt{3}\sigma}{x-\sqrt{3}\sigma}\right|$	
x	x	x
x^2	$x^2 + \sigma^2$	$x^2 + \sigma^2$
x^3	$x^3 + 3x\sigma^2$	$x^3 + 3x\sigma^2$
x^4	$x^4 + 6x^2\sigma^2 + \frac{9}{5}\sigma^4$	$x^4 + 6x^2\sigma^2 + 3\sigma^4$
x^5	$x^5 + 10x^3\sigma^2 + 9x\sigma^4$	$x^5 + 10x^3\sigma^2 + 15x\sigma^4$
x^6	$x^6 + 15x^4\sigma^2 + 27x^2\sigma^4 + \frac{27}{7}\sigma^6$	$x^6 + 15x^4\sigma^2 + 45x^2\sigma^4 + 15\sigma^6$
x^7	$x^7 + 21x^5\sigma^2 + 63x^3\sigma^4 + 27x\sigma^6$	$x^7 + 21x^5\sigma^2 + 105x^3\sigma^4 + 105x\sigma^6$
x^8	$x^8 + 28x^6\sigma^2 + 126x^4\sigma^4 + 108x^2\sigma^6 + 9\sigma^8$	$x^8 + 28x^6\sigma^2 + 210x^4\sigma^4 + 420x^2\sigma^6 + 105\sigma^8$
$\sin(x)$	$\sin(x)\,\text{sinc}(\sqrt{3}\sigma)$	$\sin(x)e^{-\frac{\sigma^2}{2}}$
$\cos(x)$	$\cos(x)\,\text{sinc}(\sqrt{3}\sigma)$	$\cos(x)e^{-\frac{\sigma^2}{2}}$
$\tan(x)$	$\frac{-1}{\sqrt{12}\sigma}\log\left|\frac{\cos(x+\sqrt{3}\sigma)}{\cos(x-\sqrt{3}\sigma)}\right|$	
$\sinh(x)$	$\frac{1}{\sqrt{12}\sigma}(\cosh(x+\sqrt{3}\sigma) - \cosh(x-\sqrt{3}\sigma))$	$\frac{1}{2}(e^{x+\frac{1}{2}\sigma^2} - e^{-x+\frac{1}{2}\sigma^2})$
$\cosh(x)$	$\frac{1}{\sqrt{12}\sigma}(\sinh(x+\sqrt{3}\sigma) - \sinh(x-\sqrt{3}\sigma))$	$\frac{1}{2}(e^{x+\frac{1}{2}\sigma^2} + e^{-x+\frac{1}{2}\sigma^2})$
$\tanh(x)$	$\frac{1}{\sqrt{12}\sigma}(\log(\cosh(x+\sqrt{3}\sigma)) - \log(\cosh(x-\sqrt{3}\sigma)))$	
$\sinh^2(x)$	$\frac{1}{8\sqrt{3}\sigma}(-4\sqrt{3}\sigma + \sinh(2\sqrt{3}\sigma - 2x) + \sinh(2\sqrt{3}\sigma + 2x))$	
$\cosh^2(x)$	$\frac{1}{8\sqrt{3}\sigma}(4\sqrt{3}\sigma + \sinh(2\sqrt{3}\sigma - 2x) + \sinh(2\sqrt{3}\sigma + 2x))$	
$\tanh^2(x)$	$\frac{1}{2\sqrt{3}\sigma}(2\sqrt{3}\sigma - \tanh(\sqrt{3}\sigma - x) - \tanh(\sqrt{3}\sigma + x))$	
e^x	$\frac{1}{\sqrt{12}\sigma}\left(e^{x+\sqrt{3}\sigma} - e^{x-\sqrt{3}\sigma}\right)$	$e^{x+\frac{1}{2}\sigma^2}$
$\sin^2(x)$	$\frac{1}{2} - \frac{1}{2}\cos(2x)\,\text{sinc}(\sqrt{12}\sigma)$	$\frac{1}{2} - \frac{1}{2}\cos(2x)e^{-2\sigma^2}$
$\cos^2(x)$	$\frac{1}{2} + \frac{1}{2}\cos(2x)\,\text{sinc}(\sqrt{12}\sigma)$	$\frac{1}{2} + \frac{1}{2}\cos(2x)e^{-2\sigma^2}$
$\tan^2(x)$	$\frac{1}{\sqrt{12}\sigma}\left(\tan(x+\sqrt{3}\sigma) - \tan(x-\sqrt{3}\sigma)\right) - 1$	
$H(x)$	$\begin{cases} 0 & x \leq -\sqrt{3}\sigma \\ \frac{x}{2\sqrt{3}\sigma} + \frac{1}{2} & -\sqrt{3}\sigma \leq x \leq \sqrt{3}\sigma \\ 1 & x \geq \sqrt{3}\sigma \end{cases}$	$\frac{1}{2}(1 + \text{erf}\,\frac{x}{\sqrt{2}\sigma})$
$\text{fract}(x)$	$\frac{1}{\sqrt{48}\sigma}(\text{fract}^2(x+\sqrt{3}\sigma) + \lfloor x+\sqrt{3}\sigma\rfloor - \text{fract}^2(x-\sqrt{3}\sigma) - \lfloor x-\sqrt{3}\sigma\rfloor)$	
$\text{fract}^2(x)$	$\frac{1}{\sqrt{108}\sigma}(\text{fract}^3(x+\sqrt{3}\sigma) + \lfloor x+\sqrt{3}\sigma\rfloor - \text{fract}^3(x-\sqrt{3}\sigma) - \lfloor x-\sqrt{3}\sigma\rfloor)$	
$\lfloor x\rfloor$	$x - \widehat{\text{fract}}(x)$	
$\lfloor x\rfloor^2$	$\widehat{x^2} + \widehat{\text{fract}^2}(x) - F(x+\sqrt{3}\sigma) + F(x-\sqrt{3}\sigma)$ where $F(x) = 2(\frac{\lfloor x\rfloor}{3} + \frac{\lfloor x\rfloor(\lfloor x\rfloor - 1)}{4} + \frac{\lfloor x\rfloor\widehat{\text{fract}}^2(x)}{2} + \frac{\widehat{\text{fract}}^3(x)}{3})$	
$\lceil x\rceil$	$x + \widehat{\text{fract}}(-x)$	
$\lceil x\rceil^2$	$\widehat{\lfloor -x\rfloor^2}$	

In equation (B.2), a_i and b_i are coefficients of the affine transformation, and a_c and b_c end up not mattering for the ρ computation,

$$a = a_c + \sum_{i=1}^{n} a_i x_i$$

$$b = b_c + \sum_{i=1}^{n} b_i x_i$$

(B.2)

so we ignore these constants. In our implementation, we find a_i and b_i by taking the gradient, via the automatic differentiation of the expression nodes a and b with respect to the inputs x_i. Here ρ is computed as:

$$\rho = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \sqrt{\sum_{i=1}^{n} b_i^2}} \tag{B.3}$$

C SMOOTHING RESULT FOR PERIODIC FUNCTIONS

In this section, we derive a convenient formula that gives the bandlimited result for any periodic function if its integral within a single period is known. We extend the analysis of fract() made by Dorn et al. [2015] to any periodic function. We use Heckbert's technique of repeated integration [Heckbert 1986] to derive the convolution of a periodic function with a box kernel.

Specifically, we assume the periodic function $f(x)$ has period T and its first and second integrals within one period are also known. These are denoted as $F_p(x)$ and $F_{p2}(x)$, respectively.

$$F_p(x) = \int_0^x f(u)du$$
$$F_{p2}(x) = \int_0^x F_p(u)du \tag{C.1}$$
$$x \in [0, T)$$

Using equation (C.1), we derive the first and second integral of $f(x)$ as follows.

$$
\begin{aligned}
F(x) &= \int_0^x f(u)du \\
&= \left(\left\lfloor \frac{x}{T} \right\rfloor + 1\right) \cdot F_p(T) - \int_{x-T \cdot \lfloor \frac{x}{T} \rfloor}^{T} f(u)du \\
&= \left(\left\lfloor \frac{x}{T} \right\rfloor + 1\right) \cdot F_p(T) - F_p(T) + F_p\left(x - T \cdot \left\lfloor \frac{x}{T} \right\rfloor\right) \\
&= \left\lfloor \frac{x}{T} \right\rfloor \cdot F_p(T) + F_p\left(x - T \cdot \left\lfloor \frac{x}{T} \right\rfloor\right)
\end{aligned}
\tag{C.2}
$$

$$
\begin{aligned}
F_2(x) &= \int_0^x F(u)du \\
&= \int_0^x \left\lfloor \frac{u}{T} \right\rfloor \cdot F_p(T)du + \int_0^x F_p(u - T \cdot \left\lfloor \frac{u}{T} \right\rfloor)du \\
&= F_p(T) \cdot T \sum_{i=0}^{\lfloor \frac{x}{T} \rfloor - 1} i + \left(x - T \left\lfloor \frac{x}{T} \right\rfloor\right) \cdot \left\lfloor \frac{x}{T} \right\rfloor \cdot F_p(T) + \\
&\quad \left\lfloor \frac{x}{T} \right\rfloor \cdot F_{p2}(T) + F_{p2}\left(x - T \cdot \left\lfloor \frac{x}{T} \right\rfloor\right) \\
&= F_p(T) \cdot \left(\frac{T \cdot (q-1) \cdot q}{2} + (x - T \cdot q) \cdot q\right) + \\
&\quad F_{p2}(T) \cdot q + F_{p2}(x - T \cdot q)
\end{aligned}
\tag{C.3}
$$

Here, $q = \left\lfloor \frac{x}{T} \right\rfloor$.

Using Heckbert's result, the convolution of the periodic function $f(x)$ with a box kernel that has support $[-\sqrt{3}\sigma, \sqrt{3}\sigma]$ (corresponding to a uniform kernel with standard deviation σ) can be expressed as follows.

$$\hat{f}(x, \sigma) = \frac{F(x + \sqrt{3}\sigma) - F(x + \sqrt{3}\sigma)}{2\sqrt{3}\sigma} \tag{C.4}$$

And the convolution of the periodic function $f(x)$ with a tent kernel that has support $[-\sqrt{6}\sigma, \sqrt{6}\sigma]$ (corresponding to a uniform kernel with standard deviation σ) can be expressed as follows.

$$\hat{f}(x, \sigma) = \frac{F_2(x + \sqrt{6}\sigma) - 2 \cdot F_2(x) + F_2(x - \sqrt{6}\sigma)}{6\sigma^2} \tag{C.5}$$

D PROOF OF SECOND ORDER APPROXIMATION FOR A SINGLE COMPOSITION

Here we show for a univariate function, applying function composition using our adaptive Gaussian approximation from Section 4.4 is accurate up to the second order in standard deviation σ. Suppose we wish to approximate the composition of two functions: $f(x) = f_2(f_1(x))$, where $f_1, f_2 : \mathbb{R} \to \mathbb{R}$. Assume the input random variable is $X_0 \sim \mathcal{N}(x, \sigma^2)$: the Gaussian kernel centered at x. The output from f_1 is an intermediate value in the computation: we can represent this with another random variable $X_1 = f_1(X_0)$. Similarly, the output random variable $X_2 = f_2(X_1)$. We conclude that $f(X_0) = f_2(f_1(X_0)) = f_2(X_1) = X_2$.

We apply equation (4.7) and equation (4.6) to f_1, and obtain the following mean and standard deviation.

$$
\begin{aligned}
\mu_{X_1} &= \hat{f}_1(x, \sigma^2) = f_1(x) + \frac{1}{2}\sigma^2 f_1''(x) + O(\sigma^4) \\
\widehat{f_1^2}(x, \sigma^2) &= f_1^2(x) + \frac{1}{2}\sigma^2 \frac{\partial^2}{\partial x^2}(f_1^2(x)) + O(\sigma^4) \\
&= f_1^2(x) + \frac{1}{2}\sigma^2(2f_1 f_1'' + 2(f_1')^2)(x) + O(\sigma^4) \\
\sigma_{X_1}^2 &= \widehat{f_1^2}(x, \sigma^2) - \hat{f}_1(x, \sigma^2)^2 \\
&= \sigma^2(f_1')^2(x) + O(\sigma^4)
\end{aligned}
\tag{D.1}
$$

Using our composition rule, X_1 is approximated as a normal distribution using the mean and standard deviation calculated from equation (D.1). That is, we approximate X_1 as being distributed as $\mathcal{N}(\mu_{X_1}, \sigma_{X_1}^2)$. Similarly, μ_{X_2}, which is the output we care about, can be computed based on equation (4.7), equation (D.1), and repeated Taylor expansion in σ around $\sigma = 0$.

$$
\begin{aligned}
\mu_{X_2} &= \hat{f}_2(\hat{f}_1(x, \sigma^2), \sigma_{X_1}^2) \\
&= f_2(f_1(x) + \frac{1}{2}\sigma^2 f_1''(x) + O(\sigma^4)) + \\
&\quad \frac{1}{2}\sigma_{X_1}^2 f_2''(\hat{f}_1(x, \sigma^2)) + O(\sigma_{X_1}^4) \\
&= f(x) + \frac{1}{2}\sigma^2 f_2'(f_1(x)) f_1''(x) + \\
&\quad \frac{1}{2}\sigma^2 f_2''(f_1(x))(f_1')^2(x) + O(\sigma^4) \\
&= f(x) + \frac{1}{2}\sigma^2 f''(x) + O(\sigma^4)
\end{aligned}
\tag{D.2}
$$

Comparing equation (D.2) with equation (4.7), the function composition in our framework agrees up to the second order term in the Taylor expansion.

We conclude that our approximation is accurate up to the second order in standard deviation for a single composition of univariate functions. The same property for additional compositions of univariate functions can be shown by induction.

REFERENCES

Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2008. *Real-time rendering.* CRC Press.

Anthony A Apodaca, Larry Gritz, and Ronen Barzel. 2000. *Advanced RenderMan: Creating CGI for motion pictures.* Morgan Kaufmann.

M Baker and S Sutlief. 2003. GreenâĂŹs Functions in Physics Version 1. (2003).

Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. 2017. Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings. *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2017)* 36, 4 (July 2017).

Adam Brady, Jason Lawrence, Pieter Peers, and Westley Weimer. 2014. genBRDF: Discovering new analytic BRDFs with genetic programming. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 114.

Antoni Buades, Bartomeu Coll, and J-M Morel. 2005. A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, Vol. 2. IEEE, 60–65.

Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. 2011. Non-local means denoising. *Image Processing On Line* 1 (2011), 208–212.

Swarat Chaudhuri and Armando Solar-Lezama. 2011. Smoothing a program soundly and robustly. In *International Conference on Computer Aided Verification.* Springer, 277–292.

Bintong Chen and Xiaojun Chen. 1999. 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.

Bintong Chen and Naihua Xiu. 1999. 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.

Robert L. Cook. 1986. Stochastic Sampling in Computer Graphics. *ACM Trans. Graph.* 5, 1 (Jan. 1986), 51–72. DOI: http://dx.doi.org/10.1145/7529.8927

Franklin C Crow. 1977. The aliasing problem in computer-generated shaded images. *Commun. ACM* 20, 11 (1977), 799–805.

Franklin C Crow. 1984. Summed-area tables for texture mapping. *ACM SIGGRAPH computer graphics* 18, 3 (1984), 207–212.

Mark AZ Dippé and Erling Henry Wold. 1985. Antialiasing through stochastic sampling. *ACM Siggraph Computer Graphics* 19, 3 (1985), 69–78.

Jonathan Dorn, Connelly Barnes, Jason Lawrence, and Westley Weimer. 2015. Towards Automatic Band-Limited Procedural Shaders. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library.

David S Ebert. 2003. *Texturing & modeling: a procedural approach.* Morgan Kaufmann.

Yuri M Ermoliev and Vladimir I Norkin. 1997. On nonsmooth and discontinuous problems of stochastic systems optimization. *European Journal of Operational Research* 101, 2 (1997), 230–244.

Yuri M Ermoliev, Vladimir I Norkin, and Roger JB Wets. 1995. The minimization of semicontinuous functions: mollifier subgradients. *SIAM Journal on Control and Optimization* 33, 1 (1995), 149–167.

AE Feiguin. 2011. Monte Carlo error analysis. (2011). https://www.northeastern.edu/afeiguin/phys5870/phys5870/node71.html [Online; accessed 22-May-2017].

Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. In *ACM Transactions on Graphics (TOG)*, Vol. 27. ACM, 33.

Paul S Heckbert. 1986. Filtering by repeated integration. In *ACM SIGGRAPH Computer Graphics*, Vol. 20. ACM, 315–321.

Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. 2015. A machine learning approach for filtering Monte Carlo noise. *ACM Trans. Graph.* 34, 4 (2015), 122.

John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection.* Vol. 1. MIT press.

Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. 2009. Procedural noise using sparse Gabor convolution. In *ACM Transactions on Graphics (TOG)*, Vol. 28. ACM, 54.

Yan-Li Liu, Jin Wang, Xi Chen, Yan-Wen Guo, and Qun-Sheng Peng. 2008. A robust and fast non-local means algorithm for image denoising. *Journal of computer science and technology* 23, 2 (2008), 270–279.

Grzegorz Łysik. 2012. Mean-value properties of real analytic functions. *Archiv der Mathematik* 98, 1 (2012), 61–70.

Don P Mitchell. 1987. Generating antialiased images at low sampling densities. In *ACM SIGGRAPH Computer Graphics*, Vol. 21. ACM, 65–72.

Don P Mitchell. 1991. Spectrally optimal sampling for distribution ray tracing. In *ACM SIGGRAPH Computer Graphics*, Vol. 25. ACM, 157–164.

Ramon E Moore. 1979. *Methods and applications of interval analysis.* SIAM.

John A Nelder and Roger Mead. 1965. A simplex method for function minimization. *The computer journal* 7, 4 (1965), 308–313.

Yu Nesterov. 2005. Smooth minimization of non-smooth functions. *Mathematical programming* 103, 1 (2005), 127–152.

Alan Norton, Alyn P Rockwood, and Philip T Skolmoski. 1982. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. In *ACM SIGGRAPH Computer Graphics*, Vol. 16. ACM, 1–8.

Kaare Brandt Petersen, Michael Syskind Pedersen, and others. 2008. The matrix cookbook. *Technical University of Denmark* 7 (2008), 15.

Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. 2012. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 195.

Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)* 30, 6 (2011), 152.

László Szirmay-Kalos and Tamás Umenhoffer. 2008. Displacement Mapping on the GPUâĂŤState of the Art. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 1567–1592.

William M Wells. 1986. Efficient synthesis of Gaussian filters by cascaded uniform filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2 (1986), 234–239.

Wikipedia. 2017a. Bessel's correction — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Bessel%27s_correction&oldid=764629526 [Online; accessed 23-May-2017].

Wikipedia. 2017b. Correlation and dependence — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Correlation_and_dependence&oldid=778221524 [Online; accessed 23-May-2017].

Wikipedia. 2017c. Entire function — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Entire_function&oldid=778079847 [Online; accessed 20-May-2017].

Wikipedia. 2017d. Hermite polynomials — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Hermite_polynomials&oldid=778044979 [Online; accessed 20-May-2017].

Lance Williams. 1983. Pyramidal parametrics. In *Acm siggraph computer graphics*, Vol. 17. ACM, 1–11.

Zhijun Wu. 1996. The effective energy transformation scheme as a special continuation approach to global optimization with application to molecular conformation. *SIAM Journal on Optimization* 6, 3 (1996), 748–768.

Lei Yang, Diego Nehab, Pedro V Sander, Pitchaya Sitthi-amorn, Jason Lawrence, and Hugues Hoppe. 2009. Amortized supersampling. In *ACM Transactions on Graphics (TOG)*, Vol. 28. ACM, 135.