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

Y. Yang\textsuperscript{1} and C. Barnes\textsuperscript{1}

\textsuperscript{1}University of Virginia, USA

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 break the input program into parts and relate the statistics of different parts of the smoothed program. We give several approximations 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
\begin{itemize}
  \item Software and its engineering \rightarrow Compilers;
  \item Computing methodologies \rightarrow Rendering;
\end{itemize}

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 [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.

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 [DW85, Mit87, HJWil}. 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.

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. 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 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.

For different operations in the input program, we need rules for how to approximate the mean and variance of the 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 4.3). We also introduce new rules that are more precise, but also more complex to compute. Specifically, we develop a novel adaptive Gaussian approximation (Section 4.4). 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 4.5) to our framework. For our last approximation rule, 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. We then show how to make minor quality improvements to the resulting programs by adjusting variances and applying denoising (Section 4.7). For each operation, we thus have different options for approximations, so we use a genetic search to find Pareto-optimal shader variants that optimally trade off running time and approximation error (Section 5).

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, Wu96]. 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 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 [DW85, 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 [DW85, Mit87, HJWil}. In video rendering, samples from previous frames can also be reused for computation efficiency [YNs09]. 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

submitted to EUROGRAPHICS 2018.
area. Dorn et al. [DBLW15] uses a compiler-driven technique to apply closed-form integral approximations to compute nodes of an arbitrary program. Unlike Dorn et al. [DBLW15], 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.

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 [KS06] demonstrated that genetic algorithms could be used to optimize ray-triangle intersection routines. Sithi-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 [OKS03, Pe05, 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. 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 input 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. We would also like the output program to be as efficient as possible.

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 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 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 inputs 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 \( \mu_X \) to denote its mean and \( \sigma_X^2 \) 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 4.5), which is sampled). We then carry mean and variance computations forward through the compute graph, using various approximations, and collect the output by taking the mean value of the output variable.

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

For a second example, in Figure 2, we show the application of each of our approximation rules to a simple 1D function. In this case, smoothing is applied only to the single input dimension \( x \).

4. Approximation Rules

4.1. Definitions: Convolutions and Random Variables

In this subsection, we define the notation we will use for convolutions and random variables, and make connections between these representations. First we note that throughout the paper, we use \( \epsilon \) 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 \( X \). To avoid confusion between these similar symbols, in each scenario we first clearly 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 \( f: \mathbb{R}^n \rightarrow \mathbb{R}^m \), which maps inputs \( x \) to outputs \( y \). We use the \( \ast \) operator to denote smoothing, so the smoothed function is \( \hat{f}(x, \Sigma) \), defined as:

\[
\hat{f}(x, \Sigma) = (f \ast G)(x) = \int_{\mathbb{R}^n} f(x - u)G(u, \Sigma)du
\]

(1)

In Equation (1), \( G(u, \Sigma) \) is the smoothing kernel that is used to smooth out the original function \( f(x) \), \( \Sigma \) is a covariance matrix associated with the kernel (more precisely, \( \Sigma \) 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 \( \hat{f}^G(x, \Sigma) \). For isotropic kernels, which have the same standard deviation \( \sigma \) for all dimensions, we also use \( f(x, \sigma^2) \) as shorthand for \( f(x, \Sigma) \), where \( I \) is the identity matrix. The convolution kernel \( G(x, \Sigma) \) 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:

\[
G(u, \Sigma) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} \exp\left(-\frac{1}{2}u^T\Sigma^{-1}u\right)
\]

(2)

If \( f(x) \) happens to be a shader program, then as is discussed in [DBLW15], \( \hat{f}(x, \Sigma) \) 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 variable \( Y \) is computed by applying a scalar-valued function \( f \) to a previous random vector \( X \) (in \( \mathbb{R}^k \), i.e., \( Y = f(X) \). If
the probability density function of $X$ is $p_X$, then by the law of the unconscious statistician, $\mu_Y$ is:

$$
\mu_Y = E[f(X)] = \int f(u)p_X(u)du
$$

(3)

As an example, if $X$ is normally distributed as $X \sim N(\mu_X, \Sigma_X)$, then Equation (3) becomes:

$$
\mu_Y = \int f(u)G(u - \mu_X, \Sigma_X)du
= \left( (f(u) * G(u, \Sigma_X)) / \mu_X \right) \mu_X
$$

(4)

If $X$ has a different probability density function, then $G(u, \Sigma)$ will be a different kernel. For example, we also consider box filters in this paper. In each case, however, $\mu_Y = \hat{f}(\mu_X, \Sigma_X)$ is the smoothed value for $E[f(X)]$. This gives some intuition for how we use convolutions and expectations of random variables to smooth functions.

### 4.2. Composition rules

In principle, Equation (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, as an approximation, we simply break up the computation into different sub-parts, and compute approximate mean and variance statistics for each sub-part. We then substitute the smoothed mean and variance that are output from one group of compute nodes as the inputs for any subsequent compute nodes.

### 4.3. Approximation of Dorn et al. 2015

We integrate the approximation methods described in Dorn et al. [DBLW15] as one of our approximation options. Dorn’s method 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 $\mu_Y$ is:

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

(5)

This is the same as the result we derived in Equation (4). Here $\hat{f}(x, \sigma_X^2)$ 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, sine, cosine, tangent, hyperbolic trigonometric functions, exponential, 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 method, the output $\sigma_Y$ is determined based on the following simplifying assumption: output $\sigma$ is a linear combination of the axis-aligned input $\sigma_X$ in each dimension. Simple rules are used, such as $\sigma$ for addition and subtraction are the sum of input $\sigma_X$s, and $\sigma$ for multiplication or division are the product or quotient, respectively, of the input $\sigma_X$s. In all other cases, including function calls, the output $\sigma$ is the average of the non-zero $\sigma_X$s of all the inputs.

We make two improvements to Dorn et al. [DBLW15], 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. [DBLW15]), we detect the case of multiplication or division by a constant and adjust the standard deviation accordingly (i.e. $\sigma_{AX} = a\sigma_Y$). 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 \( \text{Var}(X - X) = (2\sigma_x)^2 \), 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.

### 4.4. Adaptive Gaussian Approximations

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 \( X \) as \( Y = f(X) \). We assume we know the distribution for \( X \sim \mathcal{N}(\mu_X, \Sigma_X) \). Similarly as in Section 4.3, we can determine from Equation (4) and Equation (1). The result \( \mu_Y \) 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 \( \sigma_Y \) is determined differently based on the definition of the variance of \( Y \):

\[
\sigma_Y^2 = E[Y^2] - E[Y]^2 = f^2(\mu_X, \Sigma_X) - \hat{f}^2(\mu_X, \Sigma_X)
\]

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 (+), (−), (·), and can calculate the mean and variance exactly for such affine functions (using the affine estimation for correlations that is discussed later in this section). Thus, this approximation rule gives exact results for the mean for linear combinations or separable products of functions \( f(Ax) \) that we know smoothed \( \hat{f} \) for, where \( A \) is any affine transformation, and \( x \) 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 \( \sigma_i \). By using Green’s function [BS03] on the convolution of Equation (1), we can find a Taylor expansion for the function \( \hat{f}(x, \sigma^2) \) in terms of \( f(x) \):

\[
\hat{f}(x, \sigma^2) = f(x) + \frac{1}{2} \sigma^2 \nabla^2 f(x) + \frac{1}{(2!)^2} \sigma^4 \nabla^4 f(x) + \ldots
\]

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 [Wiki17]. 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 Lysik [Lys12]. 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 \( \sigma^2 \) 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 \( \sigma^2 \).

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. 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 3. The truncated Taylor expansion results in amplifying high frequencies, instead of smoothing them out.

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:

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

Here, \( \sigma_A \) and \( \sigma_B \) are standard deviations of \( A \) and \( B \). These can be determined directly from the approximation of the previous computation. Here \( \rho \) is the correlation term between \( A \) and \( B \). We will talk about how we choose \( \rho \) later in this section. The mean and
standard deviation for binary function plus \((A + B)\), minus \((A - B)\) and multiplication \((A \cdot B)\) can be derived from these assumptions based on properties of the Gaussian distribution [PP08]:

\[
\begin{align*}
\mu_{(A \pm B)} &= \mu_A \pm \mu_B \\
\sigma^2_{(A \pm B)} &= \sigma_A^2 + \sigma_B^2 \pm 2\rho \sigma_A \sigma_B \\
\mu_{(A \cdot B)} &= \mu_A \mu_B + \rho \sigma_A \sigma_B \\
\sigma^2_{(A \cdot B)} &= \sigma_A^2 \sigma_B^2 + \sigma_A \sigma_B^2 (1 + \rho^2)
\end{align*}
\]

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 4.6. In the supplemental, we derive formulas for additional multivariate functions, such as modulo, comparisons, and a ternary select function; please see supplemental Section 10.

As we discussed before, for binary functions, we approximate the input random variables \(A\) and \(B\) as bivariate Gaussian with correlation coefficient \(\rho\) (Equation (8)). In general, it is difficult to determine \(\rho\), because determining \(\rho\) exactly involves an integral over the entire subtrees of the computation. In our framework, we provide three options to approximate \(\rho\): (1) Assume \(\rho\) is zero; (2) Assume \(\rho\) is a constant for each node. The constant value is estimated at training stage by sampling; (3) Estimate \(\rho\) 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 \(\rho\). We do this using reverse mode automatic differentiation. Note that the resulting \(\rho\) 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 \(\sigma\). 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.

### 4.5. Monte Carlo Sampling

We adapt Monte Carlo stochastic sampling [Coo86, DW85] 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, \(X_1, \ldots, X_n\). For simplicity, here we assume these inputs are independently Gaussian distributed based on their specified means \(\mu_X\) and standard deviations \(\sigma_X\). For each output \(f\) of the sub-graph that is used by later computations, we compute the mean and standard deviation statistics of \(Y = f(X_1, \ldots, X_m)\) by sampled estimators:

\[
\mu_Y = \frac{1}{n} \sum_{i=1}^{n} f(\mu_{X_i} + N_{i,1}\sigma_{X_i}, \ldots, \mu_{X_m} + N_{i,m}\sigma_{X_m})
\]

\[
\sigma_Y^2 = \frac{1}{n} \sum_{i=1}^{n} f^2(\mu_{X_i} + N_{i,1}\sigma_{X_i}, \ldots, \mu_{X_m} + N_{i,m}\sigma_{X_m}) - \mu_Y^2
\]

Here, each \(N_i\) are random numbers independently drawn from normal distribution \(N(0,1)\), 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. This is mainly because the variance can also be adjusted in the “quality improvement” phase (discussed in Section 4.7).

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 \(\sigma_M\) is estimated as follows [Fei11].

\[
\sigma_M \approx \frac{\sigma_Y}{\sqrt{n}}
\]

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

### 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 (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 \(\lambda\) 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, \( \text{fract}(x) \) 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 \( \text{fract}(\cdot) \), 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 \( \text{fract} \)) 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.

### 4.7. Quality Improvements

At this point, we assume we have 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 [DBLW15], 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 [NM65] 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 [KBS15, BVM+17, RKLZ12]. 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.
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.

6.1. 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 4, 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 5 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 five choices for approximation are selected for different programs. For the correlation term discussed in Section 4.4, 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 $\rho$ a constant, and 6% use $\rho$ 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.

6.2. 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 6.1), 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 6, including one result for each base shader. The program variant shown in the result is chosen the same way as in Section 6.1. Please see our supplemental video to see these shaders rendered from a moving camera.

6.3. 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 good results for antialiasing that are competitive with direct training, transfer from curved geometries to plane tended to give good

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.

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

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 4. 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.

<table>
<thead>
<tr>
<th>Shader</th>
<th>Dorn et al. [DBLW15]</th>
<th>Adaptive Gaussian</th>
<th>Monte Carlo Sampling</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bricks w/ None</td>
<td>28%</td>
<td>0%</td>
<td>30%</td>
<td>29%</td>
</tr>
<tr>
<td>Checkerboard w/ Ripples</td>
<td>66%</td>
<td>34%</td>
<td>0%</td>
<td>1%</td>
</tr>
<tr>
<td>Circles w/ None</td>
<td>4%</td>
<td>21%</td>
<td>71%</td>
<td>4%</td>
</tr>
<tr>
<td>Color Circles w/ Bumps</td>
<td>8%</td>
<td>47%</td>
<td>44%</td>
<td>0%</td>
</tr>
<tr>
<td>Fire w/ Bumps</td>
<td>1%</td>
<td>7%</td>
<td>33%</td>
<td>60%</td>
</tr>
<tr>
<td>Quadratic sine w/ Ripples</td>
<td>13%</td>
<td>80%</td>
<td>0%</td>
<td>8%</td>
</tr>
<tr>
<td>Zigzag w/ Ripples</td>
<td>0%</td>
<td>91%</td>
<td>1%</td>
<td>8%</td>
</tr>
<tr>
<td>All shaders (fastest time)</td>
<td>29%</td>
<td>15%</td>
<td>25%</td>
<td>30%</td>
</tr>
<tr>
<td>All shaders (median time)</td>
<td>13%</td>
<td>10%</td>
<td>6%</td>
<td>77%</td>
</tr>
<tr>
<td>All shaders (slowest time)</td>
<td>20%</td>
<td>19%</td>
<td>49%</td>
<td>13%</td>
</tr>
<tr>
<td>All shaders</td>
<td>10%</td>
<td>27%</td>
<td>49%</td>
<td>14%</td>
</tr>
</tbody>
</table>
Figure 4: 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 6.1 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.
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 plane curved geometry gave bad results (due to lack of diversity of training data for the frame). Please see the supplemental Section 14 for more details and results.

7. 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 or high frequencies. 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 that have less aliasing. Future work might address this by diversely sampling from regions with different amounts and kinds 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 add any losses that specifically discourage temporal aliasing, so there may be small amounts of temporal aliasing. Finally, 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.

In summary, in this paper, we presented a novel compiler framework that smooths 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 approach has a number of important limitations. First, it can currently handle texture, lookup tables, nor do we target the GPU. Future work might address this by diversely sampling from regions with different amounts and kinds 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 add any losses that specifically discourage temporal aliasing, so there may be small amounts of temporal aliasing. Finally, 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.

7. 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 or high frequencies. 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 that have less aliasing. Future work might address this by diversely sampling from regions with different amounts and kinds 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 add any losses that specifically discourage temporal aliasing, so there may be small amounts of temporal aliasing. Finally, 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.

In summary, in this paper, we presented a novel compiler framework that smooths 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.

References

[BS03] Baker M., Sutlief S.: Greenâ€™s functions in physics version 1. 5
[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
[Cro84] Crow F. C.: Summed-area tables for texture mapping. ACM SIGGRAPH computer graphics 18, 3 (1984), 207–212. 2

Figure 5: Time versus error plots for planar geometry and the 7 shaders in Figure 1 and Figure 4. 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.
Figure 6: 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.