Pacific Graphics 2015
N. J. Mitra, J. Stam, and K. Xu
(Guest Editors)

Volume 34 (2015), Number 7

Towards Automatic Band-Limited Procedural Shaders

Jonathan Dorn, Connelly Barnes, Jason Lawrence and Westley Weimer

University of Virginia, USA

Abstract

Procedural shaders are a vital part of modern rendering systems. Despite their prevalence, however, procedural shaders remain sensitive to aliasing any time they are sampled at a rate below the Nyquist limit. Antialiasing is typically achieved through numerical techniques like supersampling or precomputing integrals stored in mipmaps. This paper explores the problem of analytically computing a band-limited version of a procedural shader as a continuous function of the sampling rate. There is currently no known way of analytically computing these integrals in general. We explore the conditions under which exact solutions are possible and develop several approximation strategies for when they are not. Compared to supersampling methods, our approach produces shaders that are less expensive to evaluate and closer to ground truth in many cases. Compared to mipmapping or precomputation, our approach produces shaders that support an arbitrary bandwidth parameter and require less storage. We evaluate our method on a range of spatially-varying shader functions, automatically producing antialiased versions that have comparable error to 4x4 multisampling but can be over an order of magnitude faster. While not complete, our approach is a promising first step toward this challenging goal and indicates a number of interesting directions for future work.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

Procedural shaders are a fundamental component of modern graphics systems, due to their flexibility and expressiveness in specifying the material appearance in a virtual scene [AMHH08]. Despite their prevalence, however, procedural shaders remain sensitive to an especially problematic source of visual error known as *aliasing* any time they are sampled at a rate below the Nyquist limit [Cro77]. Aliasing artifacts may manifest in a number of ways, including as jagged lines in place of smooth ones, small details that seem to appear and disappear, or regular features that appear to cluster instead of being evenly distributed.

There are two common approaches to *antialiasing* procedural shaders: supersampling and prefiltering. In the case of supersampling, the sampling rate is effectively increased by recording multiple samples of the shading function per pixel. Although it helps reduce aliasing artifacts, this approach has the disadvantage of increasing the computational load on the system and thus decreasing the framerate. Additionally, aliasing artifacts will still persist anywhere the frequency content in the shading function exceeds the Nyquist limit of the higher sampling rate.

Prefiltering often takes the form of storing precomputed integrals in mipmaps [Wil83] or summed area tables [Cro84]. The upside to this approach is that it offers the benefit of exact solutions in many cases with a constant number of operations. The downside is that it increases storage requirements and can replace inexpensive computations with relatively expensive memory accesses. Furthermore, these precomputation strategies scale exponentially in the number of dimensions and so it is typically not practical to precompute integrals of functions that depend on more than two or three variables.

An alternative strategy is to construct an analytically band-limited version of the shading function. This can be mathematically expressed as the convolution of the shading function with a low-pass filter [Cro77, NRS82]. For a few specialized types of procedural noise functions, exact analytic band-limiting is a natural side-effect of their construction [LLDD09]. In most cases, however, the shader developer must manually calculate the convolution integral. Because this is usually complicated and time consuming for realistic shaders, this is rarely done in practice. Instead, ad-hoc band-limiting strategies, such as replacing $step(x)$ functions with $smoothstep(x)$, are sometimes employed [AG99].

This paper explores the problem of automatically computing an exact analytic band-limited version of a procedural shader function. Such a system would take as input a shader program written in a high-level language with any number of user-defined parameters and output a modified program that produces the correct band-limited version of the input shader at any specified sampling rate. The approach should also allow limiting the function to different bandwidths without requiring significant rendering times. To the extent that the approach involves approximations, the degree of visual error should be minimized.

We propose a compiler-based approach that transforms an existing shader program into a band-limited version. We exploit the observation that, under certain conditions, band-limited subexpressions may be composed to produce a larger band-limited expression. We apply two new compiler analyses to shader program source code. The first recognizes subexpressions for which we have closed-form solutions to the band-limiting convolution. This allows us to replace the original subexpression with a band-limited expression. The second analysis conservatively approximates the relevant sampling rates for each subexpression. This allows the transformed subexpressions to be limited to the correct bandwidth. In the event that the conditions allowing such composition do not hold, we propose a search-based approach to intelligently select shaders that approximate a band-limited counterpart to the original shader.

The contributions of this paper are as follows:

- We derive closed-form analytic expressions for the integral of many common one-dimensional built-in shader functions with band-limiting kernels parameterized by the kernel width (Table 1).
- We evaluate several strategies for approximating the band-limited integral of the function $fract(x) = x - \lfloor x \rfloor$, which arises in many common procedural shader functions.
- We describe a method for approximating a 2D Gaussian kernel with correlated dimensions as the product of two axis-aligned 1D Gaussian kernels (Section 3.2). This allows factoring many shader functions into products of lower-dimensional functions whose band-limited counterparts are available.
- We describe two search algorithms that use band-limiting transformations to approximately band-limit arbitrary shaders (Section 4) and present empirical results on a set of texture shaders.

2. Related Work

Texture Prefiltering. Mipmapping [Wil83] and summed area tables [Cro84] are precomputation strategies that produce lookup tables allowing a static texture to be band-limited in constant time. In contrast, our approach requires no additional memory tables and allows for dynamically adjusting shader parameters.

Norton and Rockwood [NRS82] propose an approximation for shaders that can be decomposed as a sum of sines. They scale the amplitude of the sines based on the size of a pixel projected on the surface at that point using a power series approximation to a box kernel. Our approach is similar in spirit but more general, addressing a larger class of both shader and kernel functions.

Heitz et al. [HNPN13] describe a technique using precomputed lookup tables to band-limit static or procedural color map textures for which the mean and standard deviation can be efficiently computed. Their technique assumes that a shader to calculate the mean already exists; our approach aims to generate such shaders.

Edge Antialiasing. Much recent effort has been directed at efficient algorithms for antialiasing edges in rendered images. Morphological Antialiasing [Res09] post-processes the rendered image to identify groups of pixels with a large color gradient and particular spatial arrangement then blends their colors locally. The technique explicitly assumes that textures are separately band-limited using other techniques such as mipmapping.

Bala et al. [BWG03] and Chajdas et al. [CML11] reduce the computational cost of multi-sample antialiasing by sampling shading at a lower rate than geometry and using interpolation. These techniques detect edges to avoid interpolating shading between unrelated regions. In the absence of edges, shading quality is predicated on shading having low frequency content relative to the geometry.

In contrast to these techniques, our approach addresses high-frequency and non-band-limited shaders.

Shader Simplification. Several researchers have investigated techniques for accelerating procedural shaders in contexts with reduced requirements for level-of-detail. Olano et al. [OKS03] present a compiler technique for applying local transformations to a procedural shader, replacing memory accesses with constant colors. Pellacini's [Pel05] compiler technique locally simplifies computational logic in the shader as well as removing texture accesses. His approach uses a hill-climbing search to generate a sequence of progressively simpler shaders with increasing error relative to the original.

Sitthi-amorn et al. [SAMWL11] employ a genetic algorithm that applies local syntactic simplifications to optimize the Pareto frontier between rendering time and image error. Wang et al. [WYY*14] also employ a genetic algorithm to search through code transformations to optimize a Pareto frontier over time, error, and memory consumption. Their transformations are informed by modeling the shader function as Bézier functions on the shaded surface.

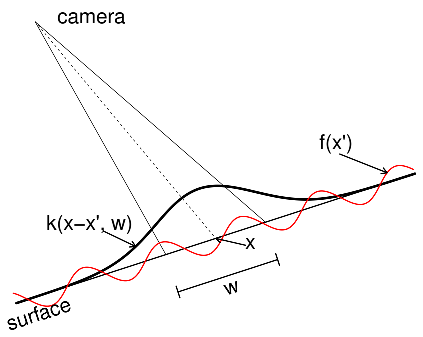


Figure 1: The problem addressed in this paper. The center of a single pixel projects to point x on the surface. The spacing between pixels at the camera corresponds to a local surface spacing of w. To determine the best single color for the pixel, we convolve the shader function $f(x')$ with the band-limiting kernel $k(x-x',w)$. Note that the sample spacing w is a parameter to the kernel function.

Rather than addressing rendering time while tolerating a certain amount of infidelity in the resulting image, our approach explicitly addresses the visual property (lower sampling rate) that enables previous techniques to tolerate error in shaders at lower levels of detail. We apply local transformations to the shader program, but produce a single shader with a dependent frequency spectrum.

Automatic Shader Bounds. Heidrich et al. [HSS98] and Velázquez-Armendáriz et al. [VAZH*09] describe compiler-based transformations that automatically augment shaders to also compute approximate bounds on their output value as a function of the bounds of their inputs. These bounds allow renderers to apply techniques such as importance sampling to more efficiently converge. The transformations applied by our technique are similar in spirit. However, they are designed to band-limit the output function instead of quantifying its bounds.

3. Band-Limited Shaders by Construction

This paper considers the problem of analytically computing the band-limited version of a procedural shader function. Perhaps unsurprisingly, there is currently no known way of analytically computing these band-limited functions in general. This section describes conditions under which exact solutions are possible. Subsequent sections investigate several approximation strategies for when they are not.

Figure 1 illustrates the problem in one dimension. Formally, given a shader function $f(x)$ of a single coordinate x, and a band-limiting kernel function $k(x,w)$ with sample spacing w, we desire the band-limited shader function $\widehat{f}(x,w)$, obtained by convolving f with k:

$$\widehat{f}(x,w) = \int_{-\infty}^{\infty} f(x')k(x-x',w)\,dx'. \qquad (1)$$

We use the Gaussian function as our band-limiting kernel with a standard deviation σ equal to the sample spacing w.

In many cases, Equation 1 will not have a closed-form solution. However, for many built-in functions that are commonly found in procedural shader languages, such as $\lfloor x \rfloor$ or $saturate(x)$, this integral can be computed directly. An important contribution of this paper is deriving these integrals for a number of common built-in functions (Table 1). Notably, although band-limiting the $fract(x)$ function permits an analytical expression it involves a sum with an infinite number of terms. Section 3.3 explores strategies for approximating this sum to achieve practical running times at acceptable error rates.

Additionally, note that because convolution is a linear operator, the band-limited version of any linear combination of the expressions in Table 1 is straightforward to compute. That is, for any functions $f_1 \dots f_n$ and constants $c_0, c_1 \dots c_n$, such that $g(x) = c_0 + c_1 f_1(x) + \dots + c_n f_n(x)$, it is the case that $\widehat{g}(x,w) = c_0 + c_1 \widehat{f_1}(x,w) + \dots + c_n \widehat{f_n}(x,w)$.

3.1. Extension to Multiple Dimensions

The previous derivation considered only one-dimensional shader functions. However, procedural shaders are often functions of multiple variables, $\vec{x} = \langle x_1, x_2, \dots x_n \rangle$ (e.g., texture coordinates, components of the normalized vector towards the camera, etc.). In the special case where the shader function and the band-limiting kernel are multiplicatively separable, the band-limited version of $f(\vec{x})$ may be written in terms of band-limited subexpressions.

More formally, let us partition the components of $\vec{x}$ into two vectors, $\vec{x}_A = \langle x_i | i \in A \rangle$ and $\vec{x}_B = \langle x_j | j \in B \rangle$, where A and B satisfy $A \cap B = \emptyset$ and $A \cup B = \{1, 2, \dots, n\}$. Further, note that the band-limiting kernel k will be a function of $\vec{x}$ as well. We denote the vector of additional parameters to the kernel as $\vec{w}$, capturing the sample rates in each dimension of $\vec{x}$. Now, if there exist sets A and B such that

$$f(\vec{x}) = g(\vec{x}_A)\,h(\vec{x}_B) \quad \text{and} \quad k(\vec{x},\vec{w}) = k_A(\vec{x}_A,\vec{w}_A)\,k_B(\vec{x}_B,\vec{w}_B)$$

then

$$\widehat{f}(\vec{x},\vec{w}) = \widehat{g}(\vec{x}_A,\vec{w}_A)\,\widehat{h}(\vec{x}_B,\vec{w}_B).$$

This provides another means of obtaining exact band-limited shaders: factor the shader function and band-limiting kernel into low-dimensional functions for which the band-limited version is available (e.g., Table 1). However, as the remaining sections explore, this is not always possible and approximation strategies must be employed.

3.2. Approximating the 2D Gaussian Kernel

Most shader functions cannot be factored into simple products or linear combinations of the functions in Table 1. In fact, there are many situations where the Gaussian band-limiting kernel alone cannot be factored. For example, consider the important case of shader functions of two spatial dimensions (e.g., surface texture coordinates). In this case, the region subtended by a single pixel is a quadrilateral, assuming locally planar geometry, and therefore the necessary 2D Gaussian kernel cannot be expressed as the product of two 1D Gaussian kernels in each of the texture coordinates.

To handle this common situation, we start with the common approximation of the quadrilateral as a parallelogram [NRS82]. We further approximate this parallelogram by its axis-aligned bounding rectangle. As illustrated in Figure 2, let x and y denote the image plane coordinate axes and s and t denote the surface texture coordinate axes. The lengths $|\partial s/\partial x| + |\partial s/\partial y|$ and $|\partial t/\partial x| + |\partial t/\partial y|$ define extent of the bounding rectangle in s and t, respectively.

We use these lengths as the widths of two 1D Gaussian functions and approximate the 2D Gaussian kernel as their product. Note that this approach is exact whenever the image plane axes are aligned with the texture coordinate axes and favors blurring over aliasing otherwise.

3.3. The $fract(x)$ Function

In this section we take a closer look at the derivation of the band-limited version of $fract(x) = x - \lfloor x \rfloor$. We chose to focus on this particular function for two reasons. First, many common primitive shader functions such as $\lfloor x \rfloor$, $\lceil x \rceil$, and $trunc(x)$, can be written in terms of $fract(x)$. Second, it turns out that this particular integral requires some form of approximation to make it useful in practice.

Recall that our goal is to solve

$$\widehat{fract}(x,w) = \int_{-\infty}^{\infty} fract(x') \frac{1}{w\sqrt{2\pi}} e^{-\frac{(x-x')^2}{2w^2}}\, dx'.$$

We use the convolution theorem [Bra86], which states that the convolution of two functions f and g can be determined by taking the product of their Fourier transforms and the inverse Fourier transform of the result. The Fourier transform of a function $f(x)$, denoted $\mathcal{F}[f(x)](k)$, is a function of frequency k describing the spectral content of $f(x)$.

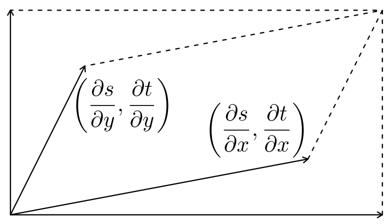

$\left(\frac{\partial s}{\partial y}, \frac{\partial t}{\partial y}\right)$
$\left(\frac{\partial s}{\partial x}, \frac{\partial t}{\partial x}\right)$

Figure 2: A single pixel in the image plane subtends a quadrilateral in the surface texture coordinate system assuming locally planar geometry. We approximate this quadrilateral by the axis-aligned bounding rectangle around the parallelogram formed by two of its edges.

First, we rewrite $fract$ in terms of its Fourier series (the full derivation may be found in the supplemental material):

$$fract(x) = \frac{1}{2} - \sum_{n=1}^{\infty} \frac{1}{\pi n} \sin(2\pi n x).$$

Following the convolution theorem, we take the product of Fourier transforms of this function and of the Gaussian kernel, then apply the inverse Fourier transform to obtain their convolution in the spatial domain,

$$\widehat{fract}(x,w) = \frac{1}{2} - \sum_{n=1}^{\infty} \frac{e^{-2w^2\pi^2 n^2}}{\pi n} \sin(2\pi n x).$$

Although this expression is exact, it contains an infinite sum that cannot be evaluated in practice. However, the terms in this sum are scaled by $e^{-2w^2\pi^2 n^2}$. Thus, the absolute value of the later terms rapidly approach zero. Therefore, one approximation method is to simply truncate this sum after a fixed number of terms or once an error bound is reached. However, we found that this approach can still yield long running times and noticeable ringing artifacts, especially when used in conjunction with other functions (Figure 3b).

We also investigated approximation strategies based on the method of repeated integration [Hec86]. Repeated integration uses the observation that the convolution of f and g can be determined by convolving the n-th integral of f with the n-th derivative of g. Specifically, we explore approximating the Gaussian kernel as a box function and a tent function, whose first and second derivatives, respectively, are impulse trains. Thus, the convolutions are obtained by repeatedly integrating $fract(x)$ and point-sampling. Table 1 lists these convolutions as $fract_2$ and $fract_3$.

3.4. Example: Band-Limited Checkerboard 1

As a simple example of the preceding results and to compare the different approximation strategies for $fract(x)$, let us consider a simple black-and-white checkerboard shader, starting with the code in Listing 1. First, recall that $\lfloor x \rfloor$ may be written in terms of $fract(x)$ (Table 1). Second, observe that `ss` depends only on the s coordinate while `tt` depends only on the t coordinate. Thus, their product is multiplicatively separable by definition, as is the product `(1−ss)*(1−tt)`.

```
float3 checker1(float2 p) {
  float ss = floor(p.s + 0.5) − floor(p.s);
  float tt = floor(p.t + 0.5) − floor(p.t);
  return (float3)(ss*tt + (1−ss)*(1−tt));
}
```

Listing 1: A black-and-white checkerboard shader implemented using the $floor(x)$ function. Note that `p.s` and `p.t` access the s and t surface texture coordinates, respectively.

$f(x)$	$\hat{f}(x,w)$
x	x
x^2	x^2+w^2
$fract_1(x)$	$\frac{1}{2}-\sum_{n=1}^{\infty}\frac{\sin(2\pi nx)}{\pi n}e^{-2w^2\pi^2n^2}$
$fract_2(x)$	$\frac{1}{2w}\left(fract^2\left(x+\frac{w}{2}\right)+\left\lfloor x+\frac{w}{2}\right\rfloor-fract^2\left(x-\frac{w}{2}\right)-\left\lfloor x-\frac{w}{2}\right\rfloor\right)$
$fract_3(x)$	$\frac{1}{12w^2}\left(f'(x-w)+f'(x+w)-2f'(x)\right)$ where $f'(t)=3t^2+2fract^3(t)-3fract^2(t)+fract(t)-t$
$\lvert x\rvert$	$x\,\mathrm{erf}\,\frac{x}{w\sqrt{2}}+w\sqrt{\frac{2}{\pi}}e^{-\frac{x^2}{2w^2}}$
$\lfloor x\rfloor$	$x-\widehat{fract}(x,w)$
$\lceil x\rceil$	$\widehat{floor}(x,w)+1$
$\cos x$	$\cos x\,e^{-\frac{w^2}{2}}$
$saturate(x)$	$\frac{1}{2}\left(x\,\mathrm{erf}\,\frac{x}{w\sqrt{2}}-(x-1)\,\mathrm{erf}\,\frac{x-1}{w\sqrt{2}}+w\sqrt{\frac{2}{\pi}}\left(e^{-\frac{x^2}{2w^2}}-e^{-\frac{(x-1)^2}{2w^2}}\right)+1\right)$
$\sin x$	$\sin x\,e^{-\frac{w^2}{2}}$
$step(a,x)$	$\frac{1}{2}\left(1+\mathrm{erf}\,\frac{x-a}{w\sqrt{2}}\right)$
$trunc(x)$	$\widehat{floor}(x,w)-\widehat{step}(x,w)+1$

Table 1: Band-limited versions of several common one-dimensional primitive shader functions. The band-limiting kernel used to derive the second column is the Gaussian function with a standard deviation equal to the sample spacing w. The *fract* function, used as the basis of $\lfloor x\rfloor$, $\lceil x\rceil$, and $trunc(x)$, is defined: $fract(x)=x-\lfloor x\rfloor$. The different versions of the *fract* function correspond to the different approximation strategies described in the paper (Section 3.3). The *trunc* function truncates its argument to the nearest integer in the direction of zero. The Gauss error function is denoted by erf.

To construct the band-limited shader, we first determine the projections of the screen-space x and y vectors. For example, in the the OpenGL Shading Language these are available as `dFdx(p)` and `dFdy(p)` [RLK09]. Given these two vectors, we then use the axis-aligned approximation (see Section 3.2) to compute the sample spacing in the surface coordinate system.

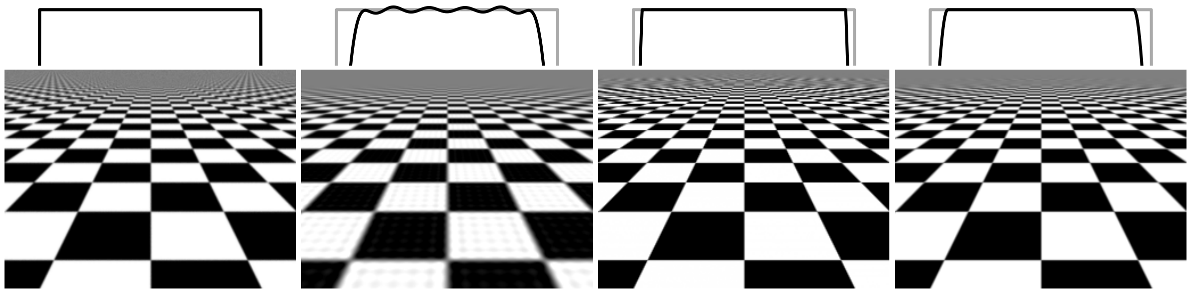
We compose the body of the band-limited shader in a bottom-up fashion. Table 1 provides the implementation of a properly band-limited floor function. We simply replace calls to `floor` with calls to this band-limited function, passing the computed approximate sample spacing.

Band-limiting the remainder of the function is trivial. Observe that `ss` and `tt` are linear combinations of functions for which we have band-limited expressions. As described in Section 3, their band-limited values are simply the linear combination of the band-limited subexpressions. For the same reason, `1−ss` and `1−tt` are band-limited expressions as well. According to the result of Section 3.1, since the `ss` and `tt` are already band-limited and their product is multiplicatively separable, the product itself is band-limited and similarly for `(1−ss)*(1−tt)`. Finally, we note that the linear combination of the two products is properly band-limited.

Figure 3 shows the checkerboard shader applied to an infinite plane. Note that the target image required 2048 shader calls per pixel to converge while the band-limited images required only one call per pixel.

4. Approximate Band-Limiting through Partial Substitution

Not all shaders are conveniently linear combinations of terms of mathematically separable functions, however. In this section we consider an automated search strategy for approximating band-limited shaders in such situations. We mo-



(a) Target (b) *fract*$_1$ (c) *fract*$_2$ (d) *fract*$_3$

Figure 3: Renderings with the checkerboard shader developed in Section 3.4, showing the effect of the different approximations to band-limiting the *fract*(x) function from Table 1. The shapes of the corresponding approximations of a square pulse are given above each rendered image. The target image (3a) was rendered using the original shader with 2048 Gaussian-distributed samples per pixel. The remaining images were rendered using a single sample per pixel. The faint grid of gray dots and blurred edges of the foreground checkers in (3b) is due to truncating the infinite series in the band-limited expression for *fract*. For this image, we chose to truncate so as to achieve a run time four times slower than *fract*$_3$ (3d). Truncating the series later reduces the visual effect at the cost of increased runtime.

tivate our approach with observations on two simple shaders: an alternate formulation of the checkerboard and a field of tiled circles.

Consider the checkerboard shader in Listing 2. It produces an image that is essentially identical to that produced by the shader in Listing 1. Note that this alternate implementation employs function composition, to which the techniques of the previous section do not directly apply. However, if we replace the calls to `fract` and `step` with the band-limited expressions in Table 1, despite the function composition, we produce the reasonable image in Figure 4c. This suggests that, even though we do not have an exact solution for situations involving function composition, it is sensible to consider the effect of composing band-limited subexpressions.

Our second example to motivate approximation, the circles shader in Listing 3, is nearly as simple as the checkerboard shaders. It employs function composition of the same functions (`step` and `fract`) to the same depth. This time, however, as shown in Figure 5b, naively replacing each subexpression with band-limited expressions introduces unacceptable artifacts. The circles appear to be inscribed within a grid of lines and the distant portion of the image is black instead of a uniform gray. These artifacts are due to squaring the result of the band-limited `fract` and to applying `step` to the result of band-limited squares, respectively. Crucially, however, we observe that replacing only the call to `step` produces the much more appealing image in Figure 5c.

Taken together, the checkerboard and circles examples suggest that a band-limited shader can be approximated in the presence of function composition by substituting only a subset of the relevant functions.

4.1. Approximation Algorithm

To handle such shaders, we use search-based software engineering techniques [HMZ12], exploring the space of similar shaders to find an implementation that approximates the band-limited output of the initial shader.

We define the search space with respect to the abstract syntax tree [ASU86] (AST) of the shader programs. To produce a new shader from an existing shader, we select a non-band-limited node of the AST and replace it with the corresponding band-limited subtree. Each replacement subtree takes the same inputs (plus a sample spacing parameter) and returns the same output type as the node it replaces, ensuring that the resulting program remains valid.

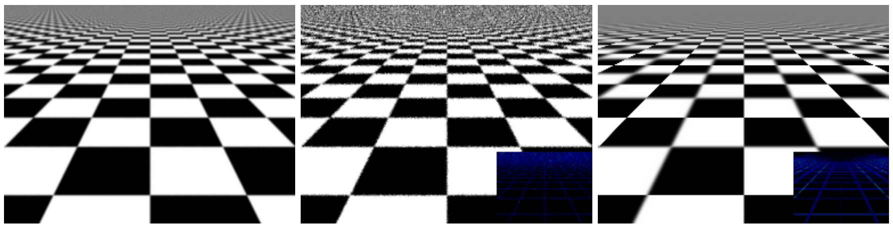
As with the circles shader above, this transformation may not result in the correct band-limited shader. We therefore define the following measurement of the quality or fitness of

```
float3 checker2(float2 p) {
  float ss = step(fract(p.s), 0.5);
  float tt = step(fract(p.t), 0.5);
  return (float3)(ss*tt + (1−ss)*(1−tt));
}
```

Listing 2: A black-and-white checkerboard shader implemented using the *step*(x) and *fract*(x) functions. Compare this program with the code listing in Listing 1.

```
float3 circles(float2 p) {
  float ss = fract(p.s*10) − 0.5;
  float tt = fract(p.t*10) − 0.5;
  return (float3) step(ss*ss + tt*tt, 0.2);
}
```

Listing 3: A shader for an infinite grid of circles. Unlike the shader in Listing 2, replacing each subexpression with its band-limited version produces an unacceptable result.



(a) Target (2048 samples) (b) Original (16 samples) (c) Band-limited (1 sample)

Figure 4: Renderings with the checkerboard shader in Listing 2. The band-limited image was generated using the $fract_3$ approximation to `fract`, given in Table 1. The other two images were generated using a Gaussian distribution for consistency with the band-limiting kernel we use in this paper. The false-color insets indicate the L^2 image error relative to the target image (4a). Rendering 4c was an order of magnitude faster than rendering 4b with only 16% more visual error.

the new shader. We consider the similarity between the images the shader produces for a set of representative scenes and the corresponding target images. Our technique is independent of the particular similarity metric used. For our experiments, we consider the average per-pixel L^2 distance in RGB.

Our search space consists of all programs reachable via a finite sequence of these node replacements. We represent programs in the space as a bit vector with one bit per node, with 1 or 0 indicating the node was or was not replaced, respectively. For a shader program with N nodes in its AST, the search space consists of $\mathcal{O}(2^N)$ programs (since each of the N nodes can either be replaced or not). For real-world programs it is infeasible to evaluate every such shader.

Instead, we propose to use genetic search to to guide exploration of this search space. Genetic search has previously been shown to apply well to repairing faulty programs [LNFW12], generating new analytic BRDF models [BLPW14], and reducing shader run time [SAMWL11, WYY*14]. Bitvector genetic algorithms are particularly well-studied [Mit98]. This provides initial confidence that it may apply well in this domain. However, to the best of our knowledge, such searches have not been applied to the specific problem of band-limiting shaders.

4.2. Determining the Sampling Rate

To achieve the correct degree of band-limiting, the kernels and the band-limited expressions we derive from them must incorporate the sample spacing as a parameter. This sample spacing is necessarily specifically associated with the spatially varying quantity in the original expression. For example, consider a modification of the checkerboard shader in Listing 2 that replaces `fract(p.s)` with `fract(p.s*2)` to produce checks that are taller than they are wide. This doubles the effective spacing between samples of the band-limited expression; the sample spacing parameter must be doubled to reflect this.

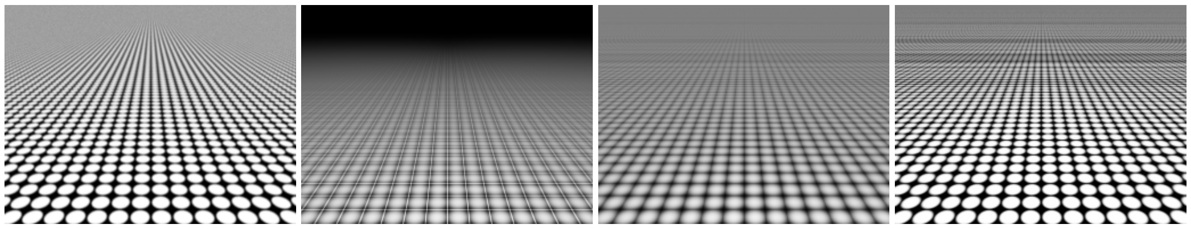
In general, the spacing between samples of the result of an expression may be different from the spacing between samples of its inputs. Note that we cannot simply compute the sample spacing based on the partial derivative with respect to the input parameters. For example, the derivative of $\lfloor x \rfloor$ with respect to x is 0 almost everywhere, yet this does not imply that no function of $\lfloor x \rfloor$ can ever alias. Nor can we use finite differencing primitives to compute the local sample spacing. For example, using `dFdx(fract(x)−1)`, which falls in the range $[-1,1]$, as the sample spacing for band-limiting `step(fract(x)−1)` could result in something other than the constant value 0.

For our experiments, we apply an approximating dataflow analysis to compute sample spacing statically as a function of the axis-aligned projection of the pixel onto the surface (see Section 3.2). We make the simplifying assumption of modeling the sample spacing as a linear combination of the axis-aligned sample spacings in each dimension. The spacing of the result of addition or subtraction is the sum of the spacings of the terms. The spacing of the result of multiplication or division is the product or quotient, respectively, of the spacings of the factors. In all other cases, including function calls, the modeled spacing of the result of an expression is the average of the non-zero spacings of the inputs.

Both the use of a linear model and our averaging approximation may result in a sampling rate for a subexpression that is either too high or too low. We therefore learn coefficients to refine this approximation. Again comparing against a set of representative target images, we use the Nelder-Mead simplex search [NM65] to learn multiplicative factors for approximated sampling rates. In the cases where our averaging approximation produces results that are too large, for example, the simplex search may learn that a smaller constant causes better agreement with the target.

5. Results

We evaluate the effectiveness of our technique for band-limiting the shaders listed in Table 2, drawn from the shaders used in previous work on antialiasing [YNS*09]. Several of these shaders (e.g., brick and wood) sample a random texture or employ a procedural noise function as a source of randomness. We treat these in the same way as any of the



(a) Target (b) Naive Replacement (c) Genetic Search Result (d) Genetic + Simplex Result

Figure 5: Renderings with the circles shader showing the results of using search to determine which expressions to replace and which coefficients to use to determine sample spacing. The target image (5a) was rendered using the original shader with 2048 samples per pixel. Naively replacing every expression produces artifacts around each the circle (5b). Using the genetic search (5c) to determine which expressions to replace produces a result that, while still not perfect, more closely matches the desired image. Combining the results of the genetic and simplex search (5d) produces an even closer approximation of the target.

			L^2 error		Runtime (ms)		
Shader	Lines	Exprs	Ours	MSAA	Ours	MSAA	Description
step	1	1	0.000	0.001	0.6	17	Black and white plane
ridges	1	1	0.011	0.056	1.3	17	$fract(x)$
pulse	2	2	0.021	0.089	2.2	18	Black and white stripes
noise1	26	3	0.042	0.044	39	241	Super-imposed noise
checker	3	4	0.072	0.125	4.1	18	Checkerboard
circles1	3	5	0.268	0.308	0.9	18	Tiled circles
wood	51	18	0.141	0.049	268	1337	Wood grain
brick	40	26	0.048	0.118	116	683	Brick wall
noise2	66	28	0.222	0.218	55	319	Color mapped noise
circles2	71	74	0.157	0.066	58	1180	Overlapping circles
perlin	79	244	0.146	0.068	4.0	71	Improved Perlin noise

Table 2: Shaders used for evaluation. "Lines" indicates the number of non-comment, non-blank lines; "exprs" lists the number of expressions that are candidates for replacement. "L^2 error" and "Runtime (ms)" indicate the performance of our antialiased shaders versus 16x multi-sample antialiasing. Our shaders are often an order of magnitude faster than 16x multi-sampling while maintaining comparable or better image quality.

functions in Table 1, that is, as nodes that the search may or may not replace with a corresponding band-limited node. Specifically, our implementation uses summed area tables and Gabor noise, for which an anisotropic band-limited formulation is known [LLDD09].

We measured the quality of the shaders by computing the average per-pixel L^2 error between a set of images rendered with one sample per pixel and the corresponding target images. We generated the target images using the original shader programs and 2048 Gaussian-distributed samples per pixel. For these experiments we chose to render an infinite plane with the texture applied, from five different rotations around an axis perpendicular to the plane. All images were generated 256x256 pixels and rendered on an NVIDIA GPU.

For each shader we ran 10 random restarts of the genetic search with a population of 200 candidate shaders for 20 generations. The number of possible shaders is given by raising 2 to the number of expressions that may be band-limited. Thus, for roughly half of our shaders our search exhaustively evaluated every variant shader that our technique generates, completing the search in under a minute. On the other hand, for the brick shader, the search evaluated less than one out of every 16,000 possible variants. For these shaders, the genetic search required roughly 8 hours for each random restart.

To reduce the time taken by our experiments, we only consider the sample spacing expressions generated by our dataflow analysis during the genetic search. However, as discussed above, in some cases this may incorrectly estimate the sample spacing. To mitigate this, we apply simplex search to the result of the genetic search as a post-processing step to find coefficients to replace those generated by the dataflow analysis, considering only the expressions band-limited by the genetic search. We ran the simplex search using the same error metric and same set of representative scenes as the genetic search, with 100 random restarts. The results presented in Table 2 reflect the best coefficients from this search.

We present images generated using the best shaders found by the genetic search in Figure 6. For comparison, we also present the images produced by the original shader with one sample per pixel and 16 Gaussian-distributed samples per pixel.

The L^2 error and run time of the shaders found by the search are listed in Table 2. Note that the run times of our shaders are significantly better than from 16x multi-sampling despite our search considering only image quality. Even though the band-limited subexpressions are typically more computationally intensive than their non-band-limited counterparts, the benefit of taking a single sample per pixel outweighs the added complexity.

6. Discussion

Our technique consistently produces shaders that are several times faster than 16x multi-sampling. In many cases, these shaders also produce comparable or less error than multi-sampling. However, in some cases such as the wood and circles2 shaders in our evaluation, we found that although our technique was more accurate than using only a single sample per pixel, it did not outperform multi-sampling in terms of visual quality. Furthermore, for the perlin shader, our technique failed to achieve any significant improvement. This prompted us to analyze the circles2 and perlin shaders to determine what aspects of those programs made the search more difficult. We summarize our findings and suggest directions for future research in this section.

Loops. Both shaders are structured around an outer loop. The circles2 shader divides the texture space into cells and processes adjacent cells in a loop. The perlin shader accumulates several octaves of noise in its outer loop, with an inner loop that sums the contributions from the vertices of the cell. The outer loop of the former shader and the inner loop of the latter have a number of iterations fixed by the algorithm and so may be easily unrolled completely. The number of octaves of Perlin noise to accumulate is generally a user-controlled parameter. For our experiments, we fixed the number of octaves at 4 to allow completely unrolling the loop.

Significantly, the bounds of the loops we investigated do not depend on spatially varying parameters. However, we note that one common approach to manually band-limiting Perlin noise is to exclude higher-frequency octaves when the sampling rate grows sufficiently large. Previous work on automatic shader simplification has demonstrated the capability to unconditionally remove high-frequency octaves [SAMWL11], but these techniques do not incorporate sampling rate in their transformations. This suggests a new band-limiting transformation to investigate, namely introducing spatially-varying bounds to existing loops. Since arbitrary loops necessarily introduce high-frequency effects to represent the jump between an integer number of iterations, further research into band-limiting them is required.

Conditionals. Both the circles2 and perlin shaders include if-statements, to determine the top-most circle and to identify the correct cell, respectively. Similarly to Velázquez-Armendáriz et al. [VAZH*09], we always compute both branches and merge their values with a ϕ-function [CFR*91] for which we know the corresponding band-limited expression. For the experiments in this paper, we used a simple function based on *step*, replacing `if (a<=b) c else d` with `step(a,b)*c + (1−step(a,b))*d`. Interestingly, the majority of nodes in circles2 replaced by the genetic search were these *step* nodes. This suggests future research could investigate the effects of different ϕ-function implementations on the success of the genetic search.

Lookup Tables. Perlin noise is often implemented using nested array accesses, where one array contains indices into a second array. Simply band-limiting the arrays in the way one might band-limit a texture would have the result of averaging the indices in the first array. This is unlikely to produce reasonable results. Recently, researchers have begun to investigate formal analysis and modeling of such nested array accesses [NKWF14]. Further research is required to discover automated transformations to handle these cases or to identify data structures and coding styles that handle them without requiring further transformation.

7. Conclusion and Future Work

This paper has explored the problem of automatically band-limiting procedural shaders. In general, this is a hard problem involving finding a solution to the convolution of an input shader function and a band-limiting kernel parameterized by the local distribution of the sample spacing. We showed that in certain cases an analytic solution can be achieved and provide those results for a number of built-in functions that are common in modern procedural shader languages (Table 1). We also demonstrated that exact solutions can be obtained for any linear combination or separable product of these functions. Finally, we described a new approximation strategy for the many cases when an exact solution is not possible. Our approach integrates a meta-heuristic genetic search over possible subexpression replacements along with a non-linear simplex search over sample spacing parameters. We showed that in some cases this approximation strategy is able to find visually pleasing results that require far less computational effort than MSAA. In a few cases, our search failed to find a satisfying result largely due to the difficulty of this search problem. Indeed, more work on this topic is needed.

We see a number of interesting directions for future work. One idea is to study alternative methods of parameterizing the space of code transformations that are considered during the search. Our genetic algorithm considers the set of shader functions reachable by replacing a subset of the subexpressions with their band-limited counterparts. Perhaps an approach that considers more generic function transformations would lead to better results.

Another direction is to develop closed-form expressions for higher-order functions and thus expand the set of subexpressions that can be exactly band-limited. For example, this

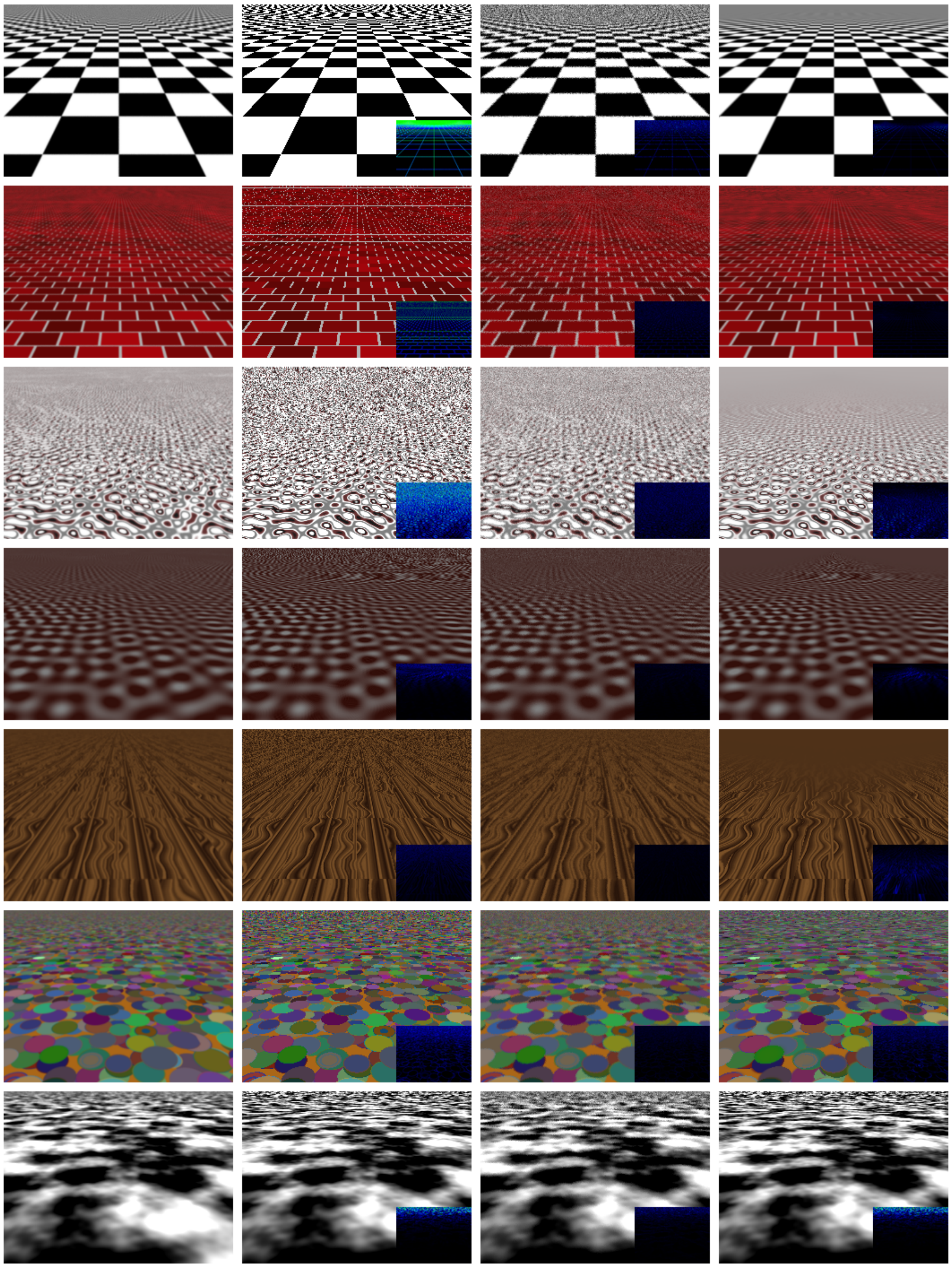


Figure 6: Comparison between (from left to right) target (2048 supersampling), no antialiasing, 16x multi-sampling, and our approach.

would avoid the requirement that the band-limiting kernel must be expressible as the product of two one-dimensional functions. A similar direction mentioned above is to investigate techniques for band-limiting loops that are bounded by spatially varying quantities.

Finally, it would be interesting to study how the design of the language may assist with this challenging task. As was demonstrated by the checker1 and checker2 shaders, there are typically many different mathematical functions that produce the same visual effect. Furthermore, it is frequently the case that one mathematical expression is preferable in terms of its suitability for this type of analysis (e.g., checker1 permits a better solution than checker2 with our method). It would be interesting to develop languages or language constructs that force a developer to produce shaders that permit analytic band-limited versions.

8. Acknowledgments

This work was partially supported by NSF grant CCF 116289, CCF 0954024, and CCF 0905373 as well as AFOSR grant FA8750-15-2-0075 and DARPA grant FA8650-10-C-7089.

References

[AG99] Apodaca A. A., Gritz L.: *Advanced RenderMan: Creating CGI for Motion Picture*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 2

[AMHH08] Akenine-Moller T., Haines E., Hoffman N.: *Real-Time Rendering*, 3rd ed. AK Peters / CRC Press, 2008. 1

[ASU86] Aho A., Sethi R., Ullman J.: *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986. 6

[BLPW14] Brady A., Lawrence J., Peers P., Weimer W.: genBRDF: Discovering new analytic brdfs with genetic programming. *ACM Transactions on Graphics 33*, 4 (Jul 2014). 7

[Bra86] Bracewell R.: *The Fourier Transform and Its Applications*, 2nd ed. McGraw-Hill series in electrical engineering. McGraw-Hill, 1986. 4

[BWG03] Bala K., Walter B., Greenberg D. P.: Combining edges and points for interactive high-quality rendering. *ACM Transactions on Graphics 22*, 3 (Jul 2003), 631–640. 2

[CFR*91] Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systtems 13*, 4 (Oct. 1991), 451–490. 9

[CML11] Chajdas M. G., McGuire M., Luebke D.: Subpixel reconstruction antialiasing for deferred shading. In *Symposium on Interactive 3D Graphics and Games* (2011), I3D '11, pp. 15–22. 2

[Cro77] Crow F. C.: The aliasing problem in computer-generated shaded images. *Communications of the ACM 20*, 11 (Nov 1977), 799–805. 1

[Cro84] Crow F. C.: Summed-area tables for texture mapping. *SIGGRAPH Computer Graphics 18*, 3 (Jan 1984), 207–212. 1, 2

[Hec86] Heckbert P. S.: Filtering by repeated integration. *SIGGRAPH Computer Graphics 20*, 4 (Aug 1986). 4

[HMZ12] Harman M., Mansouri S. A., Zhang Y.: Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv. 45*, 1 (2012), 11. URL: http://doi.acm.org/10.1145/2379776.2379787, doi:10.1145/2379776.2379787. 6

[HNPN13] Heitz E., Nowrouzezahrai D., Poulin P., Neyret F.: Filtering color mapped textures and surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2013), pp. 129–136. 2

[HSS98] Heidrich W., Slusallek P., Seidel H.-P.: Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics 17*, 3 (July 1998), 158–176. 3

[LLDD09] Lagae A., Lefebvre S., Drettakis G., Dutré P.: Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics 28*, 3 (Jul 2009), 54:1–54:10. 1, 8

[LNFW12] Le Goues C., Nguyen T., Forrest S., Weimer W.: Genprog: A generic method for automatic software repair. *Transactions on Software Engineering 38*, 1 (Jan 2012), 54–72. 7

[Mit98] Mitchell M.: *An introduction to genetic algorithms*. MIT press, 1998. 7

[NKWF14] Nguyen T., Kapur D., Weimer W., Forrest S.: Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology 23*, 4 (Sept. 2014). 9

[NM65] Nelder J. A., Mead R.: A simplex method for function minimization. *The computer journal 7*, 4 (1965), 308–313. 7

[NRS82] Norton A., Rockwood A. P., Skolmoski P. T.: Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *SIGGRAPH Computer Graphics 16*, 3 (Jul 1982). 1, 2, 4

[OKS03] Olano M., Kuehne B., Simmons M.: Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2003), HWWS '03, pp. 7–14. 2

[Pel05] Pellacini F.: User-configurable automatic shader simplification. *ACM Transactions on Graphics 24*, 3 (Jul 2005), 445–452. 2

[Res09] Reshetov A.: Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, pp. 109–116. 2

[RLK09] Rost R. J., Licea-Kane B.: *OpenGL Shading Language*, 3rd ed. Addison-Wesley Professional, 2009. 5

[SAMWL11] Sitthi-Amorn P., Modly N., Weimer W., Lawrence J.: Genetic programming for shader simplification. *ACM Transactions on Graphics 30*, 6 (Dec 2011). 2, 7, 9

[VAZH*09] Velázquez-Armendáriz E., Zhao S., Hašan M., Walter B., Bala K.: Automatic bounding of programmable shaders for efficient global illumination. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 142:1–142:9. 3, 9

[Wil83] Williams L.: Pyramidal parametrics. *SIGGRAPH Computer Graphics 17*, 3 (Jul 1983). 1, 2

[WYY*14] Wang R., Yang X., Yuan Y., Chen W., Bala K., Bao H.: Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics 33*, 6 (Nov 2014). 2, 7

[YNS*09] Yang L., Nehab D., Sander P. V., Sitthi-amorn P., Lawrence J., Hoppe H.: Amortized supersampling. *ACM Transactions on Graphics 28*, 5 (Dec 2009). 7