EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)

Volume 41 (2022), Number 2

Learning from Shader Program Traces

Y. Yang[1] C. Barnes[2] and A. Finkelstein[1]

[1]Princeton University, USA
[2]Adobe Research, USA

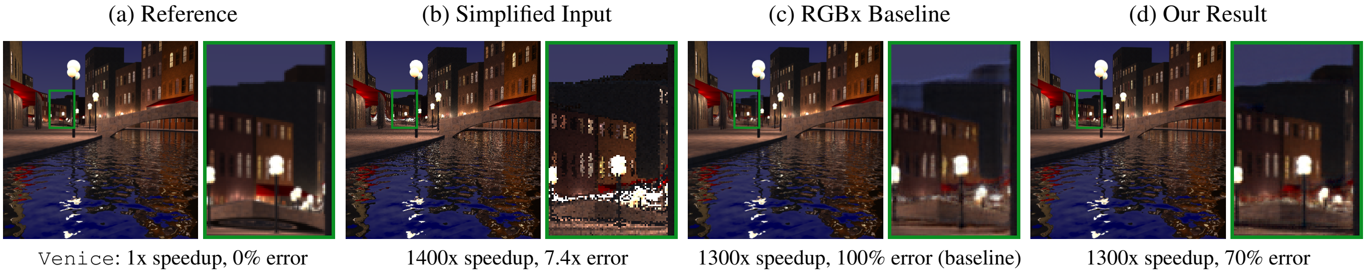

(a) Reference
(b) Simplified Input
(c) RGBx Baseline
(d) Our Result
`Venice`: 1x speedup, 0% error
1400x speedup, 7.4x error
1300x speedup, 100% error (baseline)
1300x speedup, 70% error

Figure 1: *Learning to extrapolate from the partial computation of a procedural shader called* `Venice`*. The reference solution (a) results from a full computation at 1000 samples per pixel (*SPP*). A simplified version of the shader provides an approximate solution (b) using only 1* SPP *and less computation per sample (72% of the original compute; 1400x speedup overall). Our RGBx baseline method (c) learns to approximate the reference well for much of the image, based on only the RGB output of the simplified shader as well as a few hand-picked auxiliary features – but exhibits artifacts in the distance (obvious in the zooms boxed in green). This paper shows that difficult learning tasks like this can benefit from relying on not just the RGBx features but also the* program trace *(a record of the intermediate values computed at every pixel, in this case that of the simplified shader) – producing a more faithful approximation of the reference. Percent (%) denotes mean perceptual error [ZIE*18] relative to that of the RGBx baseline, averaged over the test set.*

Abstract

Deep learning for image processing typically treats input imagery as pixels in some color space. This paper proposes instead to learn from program traces of procedural fragment shaders – programs that generate images. At each pixel, we collect the intermediate values computed at program execution, and these data form the input to the learned model. We investigate this learning task for a variety of applications: our model can learn to predict a low-noise output image from shader programs that exhibit sampling noise; this model can also learn from a simplified shader program that approximates the reference solution with less computation, as well as learn the output of postprocessing filters like defocus blur and edge-aware sharpening. Finally we show that the idea of learning from program traces can even be applied to non-imagery simulations of flocks of boids. Our experiments on a variety of shaders show quantitatively and qualitatively that models learned from program traces outperform baseline models learned from RGB color augmented with hand-picked shader-specific features like normals, depth, and diffuse and specular color. We also conduct a series of analyses that show certain features are important within the trace: these coincide with intuitively important aspects of the program. The important features can be helpful for selecting a good subset of trace features for learning, and even learning from a small subset of the trace already outperforms the baselines. We finally show that multiple shaders can be learned together with a shared denoising network and a lightweight shader-specific encoder.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Neural networks; Computer graphics;*** • ***Software and its engineering*** $\rightarrow$ ***Compilers;***

---

1. Introduction

Deep learning applications in graphics and vision typically work on images encoded as pixels in RGB color space. For images with 3D scenes, researchers have also explored augmenting the RGB data with hand-picked features like depth or surface normals [CKS*17, VRM*18]. These auxiliary features are picked based on domain expertise, and vary for different applications or programs.

This paper proposes augmenting the data from which a neural network learns with the *program trace*. In software engineering, a trace refers to the record of all states that a program visits during

© 2022 The Author(s)
Computer Graphics Forum © 2022 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

its execution [FH93, Lar93], including all instructions and data. We explore this idea in the context of procedural shader programs, like the one shown in Figure 1. The sequence of instructions tend to be similar from pixel to pixel, so we rely on just the intermediate values for learning, referring to these as the "program trace."

Shader programs can be used to flexibly construct complex and even fantastical appearances by combining sequences of mathematical operations to create texture patterns, produce lighting, perturb surface normals to produce effects such as bump mapping, apply noise functions, or determine ray intersections with procedurally generated geometry [AMHH08]. A range of example shaders may be seen throughout this paper; many more examples are available from websites such as www.shadertoy.com. Note that while the example shaders appearing here are simpler than those typical of production or games, they embody the key features that appear in production-level shaders.

Since the fragment shader program operates independently per pixel, we can consider the full program trace as a vector of values computed at each pixel – a generalization from simple RGB. Since there are many pixels (program traces) per image, and potentially many computed images, this provides a rich source of data from which to learn. Graphics and vision researchers have explored learning algorithms for input-output image pairs with a few auxiliary feature buffers, such as those of Vogels et al. [VRM*18] on removing sampling noise, and Xie et al. [XFCT18] on fluid super-resolution. Such features are *manually* identified by an expert on a per-shader basis. Moreover, the extent to which these auxiliary features helps learning depends on the choice of features, the particular shader, and the learning goal. We believe other shader-specific information useful to the learner remains hidden within the program execution, and that a learning process could *automatically* identify and leverage that information. Thus, we propose a learning-based approach that utilizes all of the information produced during the execution of a shader program. The learner could automatically identify which features are useful, obviating the need for manual feature selection in the midst of an experimental process.

Intuitively, learning tasks that extrapolate from a partial computation to predict the result of a full computation may benefit from learning from program traces. To illustrate its applicability, we introduce four applications. Three of them work from pixel data: learning to predict low-noise output, learning to reconstruct full computation from a program with partial computation, and learning the output of a postprocessing filter. The fourth application shows that the idea of learning from program traces can be applied to non-imagery data: it learns to simulate the position and velocity of a flock of "boids" [Rey87], which emulate flocking behavior similar to that of birds or fish. The performance of these applications is summarized in Figure 2. In most of our experiments, we train a separate model for each shader on each application. Scene specific learning is commonly used in recent work on novel view synthesis [STH*19, TZN19, TZT*20, MST*20]. Section 6.3 describes how a single network can be trained over *multiple* shaders.

The primary contribution of this paper is the idea that a shader program trace can be used as a feature vector for machine learning. Nevertheless, it is neither obvious how to use such a feature, nor that it would help in any particular application. Thus, a secondary contribution is to introduce a framework for learning from program traces, and demonstrate that it outperforms baseline methods in several applications. The third contribution is to investigate the relative importance of individual trace features, and how the input trace size across various trace subsampling strategies can affect the performance of the model. We include our code, data and models as a part of the supplemental material and will release them upon publication.

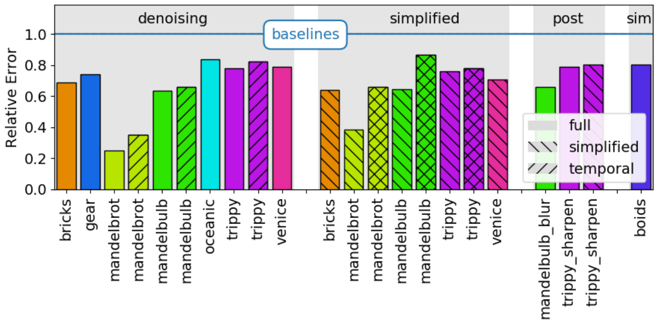


Figure 2: *Four-application summary: **denoising**, reconstruction from **simplified** shaders, learned **post**-processing effects and **sim**ulation. The vertical axis shows (in every example) improved perceptual error compared to each applications' strongest baseline: RGBx for denoising, simplified, and post; and I/O for simulation (Section 5). Hatching directions denote use of simplified shaders and/or temporally coherent models.*

2. Related Work

Program traces in machine learning. Program traces have proven helpful in malware detection [CSS18], program induction [RF16] and program synthesis [CLS19, JMKO20]. Researchers have also explored using partial execution or partial rendering to synthesize graphics programs [GKB*18] or infer parameters for procedural models [RTHG16]. Instead of developing specialized learning models for a particular application, we explore a generic architecture that can learn over a range of applications. Nevertheless, this work focuses on learning from program traces for shaders, which enjoy certain unique properties such as an emphasis on pixel outputs and an enormous degree of parallelism.

Features for deep learning on imagery data. Researchers have explored a variety of features beyond simple RGB as inputs to learned functions. Nalbach et al. [NAM*17] have made a comprehensive exploration of such features as part of a deferred shading pipeline. Xie et al. [XFCT18] consider auxiliary features including flow velocity and vorticity when learning density super-resolution for fluid simulation. Positional encoding [MST*20] augments the input by applying high frequency functions to coordinate features, but does not benefit our tasks (Appendix H). To our knowledge, our paper is the first to propose augmenting such features with the full program trace. The benefits are that the trace of a program that computes such manually picked features inherently includes them, as well as other potentially useful information; moreover the extent to which various features are useful for a particular application and shader are discovered automatically by the learning process.

Feature space reduction. In deep networks, an overly large feature space can exhaust memory, increase training time, or even make learning tasks harder. Researchers have explored methods to reduce the feature space by pruning whole convolutional filters [LKD*16, LWL17, MTK*16]. In our method, we focus mostly on reducing the input feature space because the dimension of the program trace can be large. We use a method similar to that of Molchanov et al. [MMT*19] to evaluate the importance of each trace input and show a trade-off between the runtime and visual fidelity as we change our feature reduction strategies. Our experimental results suggest that without prior execution or learning, we could find no subsampling strategy that consistently outperforms a simple uniform subsampling. However, if the shader is allowed the overhead of learning a model over the full program trace, we can select important trace features from the learned model.

Remove sampling noise. One of our applications addresses Monte Carlo noise reduction in low-budget rendering. One strategy for low-noise rendering involved carefully distributing the samples, see Zwicker et al. [ZJL*15] for a survey. Another method uses symbolic compilation techniques to analytically approximate the integral that produces the smoothed shader [DBLW15, YB18], but is hard to scale to complicated shaders such as the one shown in Figure 1. On the other hand, learning-based algorithms train regressors such as neural networks to predict the rendering. The input to networks are usually augmented with auxiliary features [CKS*17, VRM*18, GLA*19]. Unlike previous work, our approach gathers customized information per shader, and is orthogonal to learning-based denoising network design in the sense that it can be combined with an existing network.

Shader simplification. As the complexity of the shader program grows, it is common to apply lossy optimization to obtain programs that only approximate the original program but with better runtime performance [HFTF15, SaMWL11, WYY*14]. We show experiments that explore how the trace from the simplified programs can provide information that helps to recover the missing details in the target shader. Thies et al. [TZN19] identify a similar task where they learn novel view synthesis from a coarse proxy geometry. (The `Venice` example shown in Figure 1 is particularly reminiscent of that work.) Nevertheless, to our knowledge this is the first paper to propose the application of learning from a simplified shader program to restore details in the original program; and we show that using the program trace can help in this application.

Neural networks for image processing. Researchers have investigated a variety of learning-based methods for image processing tasks, such as image enhancement and filtering [GCB*17, LGA*18, LHAY17, WZZH18]. Our postprocessing application demonstrates that the proposed method is also helpful when learning these imagery operations as postprocessing filters.

Learning simulation programs. High quality simulation usually executes the program over many tiny time steps, which is expensive. Researchers have developed reinforcement learning based methods [HPG*19, KZP*20] to replace the program entirely, or execute the program at a lower spatial resolution and learn a super-resolution model [WXCT19]. Our approach instead learns from the program's execution trace on a larger time step, and corrects the output as if the program is executed for multiple smaller steps.

3. Compiler and Preprocessing

This section introduces a compiler that can collect traces from shader programs. It translates shader programs from a domain specific language to TensorFlow code that logs the trace (Section 3.1). To stay within the hardware memory budget, the compiler also restricts the trace length to an arbitrary size cap (Section 3.2). Collected program traces are further preprocessed before learning (Section 3.3). Section 4 describes the learning process in detail.

3.1. Compiler and Program Traces

Our compiler takes as input an arbitrary procedural shader program written in a domain specific language (DSL) and translates it to a TensorFlow (TF) program that outputs a rendered image as well as a collected program trace. We embed the DSL in Python, which allows us to use Pythonic features such as operator overloading. We also include common shader operations such as trigonometric functions, dot and cross products. For simplicity, we assume the shader program manipulates numerical scalars or vectors of known size. We handle branching by computing both branches of conditionals. Likewise, loops are unrolled to the maximum possible number of iterations: this limit is set by the programmer for each loop. These are not fundamental limitations of the approach, as we experimented emulating branching and variable-length loops by writing dummy values of zero to traces in the branch/iteration not executed, and this gives visually and quantitatively identical results to our current approaches (Appendix G). These policies permit us to express the trace of any shader as a fixed-length vector of the computed scalar values, regardless of the pixel location.

3.2. Feature Vector Reduction

Large program traces can produce unnecessarily large feature vectors from which learning becomes unwieldy, or worse, exhausts memory. Loop unrolling is a common contributor to large traces, because the program trace would be scaled by the number of iterations. The remainder of this section describes several strategies for reducing the size of the feature vector. All these strategies can be reused when targeting a different language, e.g. OpenGL Shading Language (GLSL) or CUDA.

Compiler optimizations. Since such features would be redundant in the learning network, the compiler omits constant values, duplicate nodes in the compute graph, and neighboring nodes that differ only by a constant addition or multiplication. The compiler also identifies common built-in functions and iterative improvement loops to eliminate trace features that are highly correlated.

Built-in functions (e.g., `sin`) should typically be treated as a black box. Our DSL provides widely used shader operations such as noise functions and a normal computation functor. The compiler logs only the return values of such built-in functions, not the intermediate values found when computing them. This is a natural choice since in principle one could trace down to a very low level such as including details about the microarchitecture, but we believe that learning will gain the most benefit if it occurs at a similar abstraction level as used by the programmer.

An iterative improvement loop repeatedly improves an approximate result to obtain a more accurate result [SDMHR11]. A commonly used iterative improvement pattern in shader prototyping is a ray marching loop that computes the distance from the camera to objects in the scene. Because each iteration computes a more accurate approximation than the previous iterations, the final iteration is the most informative. Therefore, the compiler will only log the trace from the final iteration of such loops. We automatically handle common cases of iterative improvement loops found in shaders by classifying loops based on a pattern matching: the output of the loop is either iterative additive or can be written as a parametric form of the iterative additive variable. Detailed classification rules appear in Appendix A. We also investigate several other strategies inspired by previous work on loop perforation [SDMHR11] and image perforation [LNLB16]. In our case, however, we always run the full computation, but simply select a subset of those computations as input to the learning task, as follows.

Uniform feature subsampling. The most straightforward strategy is to subsample the vector by some factor n, retaining only every n^{th} trace feature as ordered in a depth first traversal of the compute graph. This approach tends to work well in our experiments, and we speculate that it does so because nearby nodes in the compute graph tend to be related by simple computations and thus are redundant.

Other sampling schemes. We explored a variety of other schemes to reduce the feature vector length, including “clustering" based on statistical correlation, “loop subsampling” that logs features from every kth loop execution; “first or last” which only collects features from either the first or last iteration of a loop; and “mean and variance” summarize the statistics of a variable over all loop iterations. Yet none outperformed the above straightforward scheme consistently enough to justify their use in our subsequent experiments.

These options are combined as follows. We first apply compiler optimizations, then subsample the features with a subsampling rate that makes the trace length be most similar to a fixed target length. For all experiments, we target a length of 200, except where specifically noted such as in the simulation example. After compiling and executing the shader, we have for every pixel: a vector of dimension $\mathcal{N}$: the number of recorded intermediate values in the trace.

3.3. Whitening the Collected Trace

We preprocess the traces to rescale the data to a fixed range. Intermediate values in computed shader programs can vary over a large range: resulting in values such as 10^{30}, $\pm\infty$, or *not a number* (NaN), even when most values of this shader computation remain near zero. This can happen, for instance, near object silhouettes where textures have high frequency in image space. The extreme values could cause a standard whitening technique to fail entirely, due to say undefined mean or standard deviation where values such as $\pm\infty$ or NaN are present. Even if only finite trace values are observed at training time, standard whitening may focus too much on extreme values such as 10^{30}, resulting in meaningful data (e.g. between $[-1,1]$) being mapped to a very small range, and at test time, extreme values such as $\pm\infty$ or NaN can still produce non-finite floating point values that are problematic for inference.

We thus develop a whitening method for shader program traces. We first clamp the raw program traces by collecting the statistics for the intermediate values’ distribution at training time and decide the clamping threshold based on the lowest and highest pth percentile ($p = 5$ in practice). The clamped values are then rescaled to a fixed range [-1, 1]. More details are described in Appendix B.

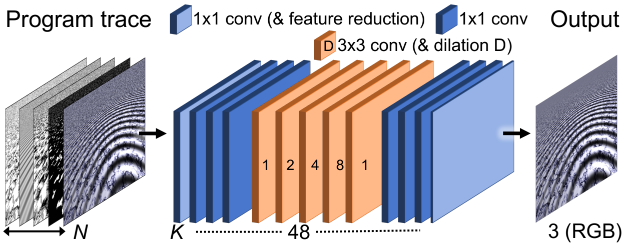


Figure 3: *Network architecture used in our experiments. The input from the program trace has $\mathcal{N}$ channels. The output layer has three channels for color images. The first feature reduction layer has $\mathcal{K}$ channels. We use $\mathcal{K} = 48$ in our method. When training the baseline method, $\mathcal{K}$ will be increased to a larger value to match the total number of trainable weights to be the same as training with the program trace at the maximum length. All other intermediate layers have 48 channels. The input feature maps are first analyzed by four 1x1 convolutional layers, followed by five 3x3 convolutional layers with dilation rates of $1,2,4,8,1$ respectively. Finally, four additional 1x1 convolutional layers are applied and output a three channel image. Note that the first and last convolutional blocks indicated in lighter blue each reduce the number of channels (from $\mathcal{N}$ to 48, and from 48 to 3, respectively).*

We evaluated the effectiveness of scaling and clamping on the denoising task (Section 5.1) with `Mandelbrot`. If trained without clamping, the model will diverge to NaN even before the first iteration finishes, while training without whitening results in 12x worse perceptual error compared to our full method. These results indicate that our data preprocessing is essential in our pipeline.

4. Network Architecture and Training Details

This section briefly summarizes the training details in our experiments. For all of our imagery applications we selected a basic architecture described in Section 4.1. Nevertheless, our method could be coupled with any deep learning architecture. This section thus serves as an example of how to select a network architecture and carry out training.

4.1. Network Architecture

Our experiments use a dilated convolutional neural network depicted in Figure 3, similar to that of Chen et al. [CXK17]. This network architecture is used directly in our denoising (Section 5.1) and post-processing (Section 5.3) applications, and also serves as a generator model in other applications and scenarios, each of which relies on a GAN model: conditional spatial GAN [IZZE17] for learning from a simplified shader (Section 5.2) and temporal GAN [WLZ*18] for learning temporally coherent sequences (Appendix F). We prefer to keep a single network architecture for consistency and ease of experimentation across all applications except boids, in order to demonstrate our core idea of learning from shader program traces is beneficial across many applications, although

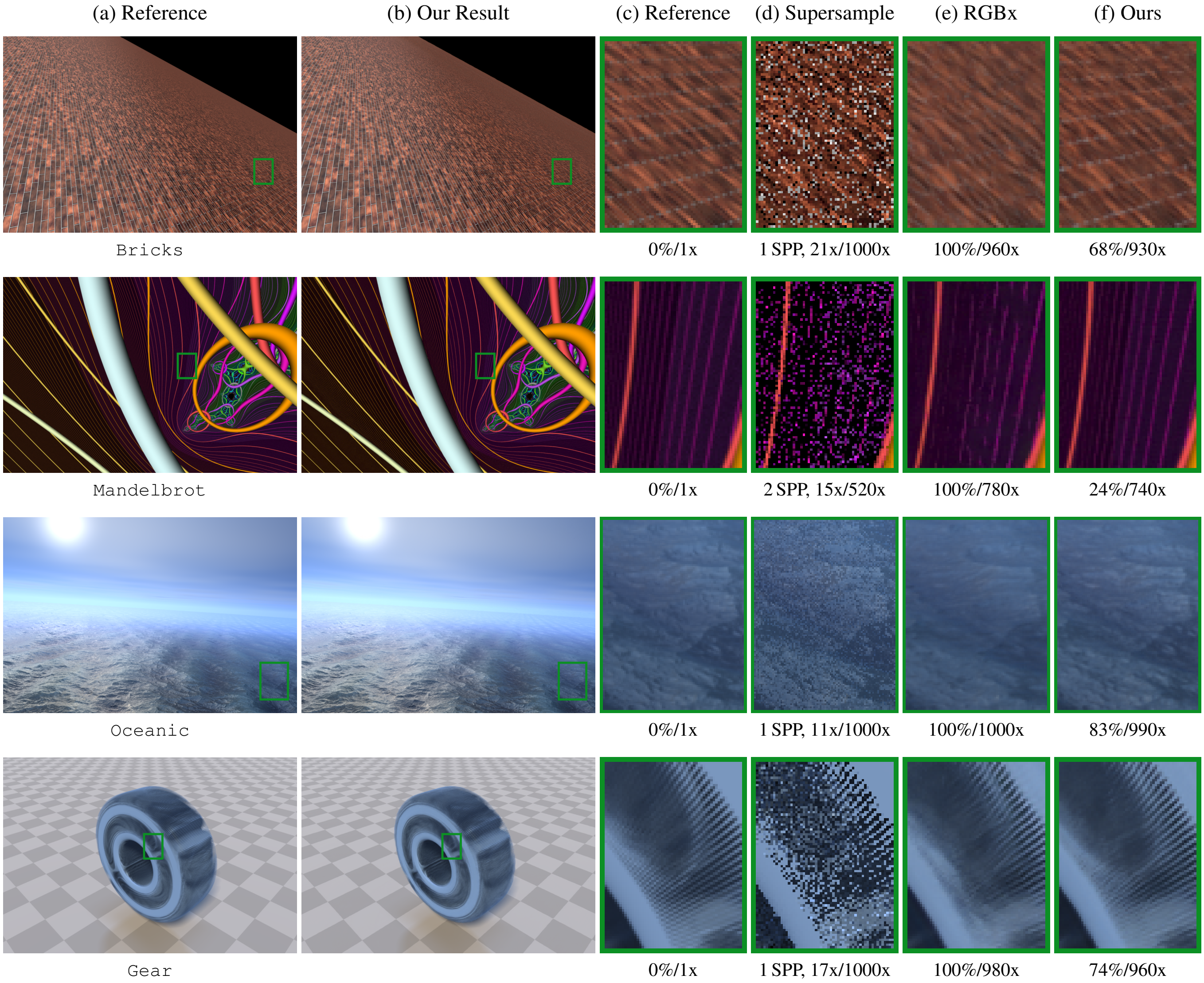

(a) Reference
(b) Our Result
(c) Reference
(d) Supersample
(e) RGBx
(f) Ours
Bricks
0%/1x
1 SPP, 21x/1000x
100%/960x
68%/930x
Mandelbrot
0%/1x
2 SPP, 15x/520x
100%/780x
24%/740x
Oceanic
0%/1x
1 SPP, 11x/1000x
100%/1000x
83%/990x
Gear
0%/1x
1 SPP, 17x/1000x
100%/980x
74%/960x

Figure 4: *Learning to reduce sampling noise in procedural shaders. The reference low-noise solution (a) relies on 1000 samples per pixel (SPP). Our method (b) approximates the reference well at only 1 SPP. Zooming into the region boxed in green (c, f) reveals approximation error, which compares favorably with the two baselines: (d) supersampling where the number of samples is chosen to have comparable run-time as ours, and (e) RGBx. Error and speedups are reported as in Figure 1. Our method better covers both the orientation and high-frequency detail than the baselines.* `Bricks` *and* `Mandelbrot` *are gamma corrected to emphasize visual differences when viewed on screens of various brightness. Full images for all experiments can be found in the supplemental material.*

more specialized architectures could be beneficial for certain applications like denoising (e.g. [GLA*19]). Details about the GAN models are discussed in Appendix D. The boids simulation relies on a fully-connected architecture described in Appendix E.

4.2. Loss Functions

We use a combination of pixel-wise color loss L_c and perceptual similarity loss L_p to encourage network output to be similar to the ground truth during training:

$$L_b = L_c + \alpha L_p \tag{1}$$

The parameter α is a weight that balances between the color and perceptual loss terms. We fix $\alpha = 0.04$ for all of our experiments. This value was chosen to roughly balance the magnitude of the gradients due to L_c and L_p during back-propagation.

The color term L_c is simply the standard L_2 loss on the RGB image. The other loss term L_p uses the learned image perceptual dissimilarity metric of Zhang et al. [ZIE*18]. This section describes the basic loss L_b used in training. Additional details about the GAN losses can be found in Appendix D, and the loss used in the boids simulation is described in Appendix E.

4.3. On the Fly Training

In training, we generate input program traces on the fly each time one is needed, rather than loading precomputed traces from disk. There are two benefits to this approach. First, precomputed traces are large, and it is typically faster to re-compute the trace, as opposed to loading from disk. Second, each time a trace is generated, we use a new randomly sampled sub-pixel location for evaluating the trace for any given pixel (a common strategy to reduce aliasing). Therefore, the input traces will generally have different values in each epoch even though we use the same ground truth solution. This approach helps the network avoid overfitting.

5. Evaluation

This section describes experiments evaluating our method for various applications and scenarios: denoising pixel shaders (Section 5.1), learning to reconstruct simplified shaders (Section 5.2), learning postprocessing effects (Section 5.3), and learning non-imagery simulation programs (Section 5.4). We also discuss learning temporal coherence in Appendix F. The architecture and training scheme in these applications includes fully connected networks, traditional CNNs and GANs, demonstrating our method's wide applicability to various deep learning models. We report LPIPS, SSIM and PSNR for all applications in Table 1, with a performance summary shown in Figure 2: in all cases our method outperforms the strongest baseline.

For the image processing applications, we choose a variety of challenging combinations of shaders and geometries. The `Bricks` shader relies on simplex noise [Per01]. The `Trippy Heart`, `Mandelbrot` and `Mandel-bulb` shaders rely on iterative fractals. The shaders `Mandel-bulb`, `Gear`, `Oceanic` and `Venice` construct complex 3D procedural geometry rendered by ray marching over a signed distance field [Won16]. The shaders `Bricks` and `Venice` extract contents from texture maps. We adapted shaders `Oceanic`, `Trippy Heart`, `Mandel-bulb` and `Venice` from shaders with the same names at the web site www.shadertoy.com, by the authors *Frankenburgh*, *Cras*, *EvilRyu*, and *reinder*, respectively, while `Gear` is adapted from the shader "primitives" by author *Iq*; and the boids and fluid simulations were adapted from "Simple Boids" by *Saduras* and "Chimera's Breath" by *nimitz*.

Our implementation is trained on a single GPU. For consistent timing in evaluation, we use a 4 core Intel Xeon E5-2620 v4 2.10 GHz CPU with a single Nvidia GForce RTX 2080 Ti GPU across all models. During training, we always train 400 epochs for models without a GAN and 800 epochs for models with a GAN. Timing results reported throughout appear as speedup relative to ground truth. The actual shader runtime ranges from 30ms to 21s with a median of 1.4s for full computation i.e. non-simplified shaders (Section 5.1 and 5.3), and from 20ms to 6s with a median of 80ms for partial computation (Section 5.2). Inference time ranges from 70ms to 0.2s with a median of 90ms. These shaders are relatively slow because they are implemented as computational graphs in TensorFlow. They could be greatly accelerated through engineering a GLSL or CUDA implementation. Note the shader's runtime is invariant to whether program traces are collected or not, therefore it is not a limitation to our proposed method. In all cases, we select the model at the epoch with the lowest validation loss. For imagery learning tasks (Section 5.1, 5.2, 5.3), the model trains on a dataset of 1200 tiles with 320 × 320 resolution, and 120 validation tiles in same resolution. Testing includes 30 full size images with resolution 640 × 960. Please refer to Appendix C for further details regarding our training. All experiments presented in this section are trained per shader. We also demonstrate in Section 6.3 that multiple shaders can be trained together with a shared network and a lightweight shader-specific encoder.

Table 1: *Error statistics for applications in Section 5 and 6.3. Errors reported as LPIPS [ZIE*18] / SSIM / PSNR.*

	Shader	RGBx	Ours
Denoising	`Bricks`	0.0141 / 0.981 / 36.68	**0.0097 / 0.987 / 38.29**
	`Gear`	0.0173 / 0.986 / 38.90	**0.0127 / 0.988 / 39.86**
	`Mandelbrot`	0.0235 / 0.973 / 36.07	**0.0059 / 0.986 / 38.55**
	`Mandel-bulb`	0.0185 / 0.962 / 32.14	**0.0118 / 0.975 / 34.18**
	`Oceanic`	0.0403 / 0.961 / 33.69	**0.0339 / 0.966 / 34.51**
	`Trippy Heart`	0.0696 / 0.856 / 26.30	**0.0543 / 0.886 / 27.27**
	`Venice`	0.0309 / 0.965 / 32.76	**0.0242 / 0.973 / 33.83**
Simplified	`Bricks`	0.0624 / 0.922 / 25.57	**0.0398 / 0.936 / 29.96**
	`Mandelbrot`	0.1111 / 0.826 / 27.04	**0.0430 / 0.949 / 31.15**
	`Mandel-bulb`	0.0932 / 0.812 / 25.22	**0.0600 / 0.856 / 26.85**
	`Trippy Heart`	0.2412 / 0.520 / 18.55	**0.1824 / 0.629 / 20.95**
	`Venice`	0.0404 / 0.957 / 31.50	**0.0285 / 0.965 / 32.72**
Post	blur	0.0126 / 0.978 / 35.81	**0.0082 / 0.985 / 37.62**
	Sharpen	0.0881 / 0.833 / 24.26	**0.0693 / 0.868 / 25.31**
	Simp Sharpen	0.2675 / 0.477 / 17.20	**0.2154 / 0.587 / 19.35**
Shared	`Gear`	0.0251 / 0.984 / 38.16	**0.0173 / 0.986 / 38.66**
	`Mandelbrot`	0.0546 / 0.801 / 28.02	**0.0165 / 0.933 / 32.60**
	`Mandel-bulb`	0.0423 / 0.861 / 28.02	**0.0298 / 0.906 / 30.19**
	`Trippy Heart`	0.1048 / 0.815 / 19.99	**0.0755 / 0.857 / 23.88**

Our strongest baseline is RGBx. It uses the same network and training as ours, but with the input features consisting of RGB color plus manually picked auxiliary features that are commonly used for learning with shader programs. We use normal, depth, diffuse and specular color whenever these terms are explicitly represented in the program. These corresponds to auxiliary features used in recent denoising papers [CKS*17, VRM*18]. Because the RGBx baseline generally has fewer input channels compared to our method, we increase the number of channels in the first convolutional layer of the baseline model such that the number of trainable weights matches that of our model. Unlike our *automatic* method, RGBx requires additional *manual* expertise to pick auxiliary features for every shader program. An automatic baseline that resembles ours would be RGB, which uses only RGB color without any auxiliary features. However, RGBx always outperforms RGB, so we only compare with RGBx.

5.1. Denoising Fragment Shaders

Here we describe the application of removing sampling noise. Our goal is to approximate a low noise reference image collected using 1000 samples per pixel (SPP). Our method is evaluated using 1 SPP, drawn from a Gaussian spatial distribution with a standard deviation of 0.3.

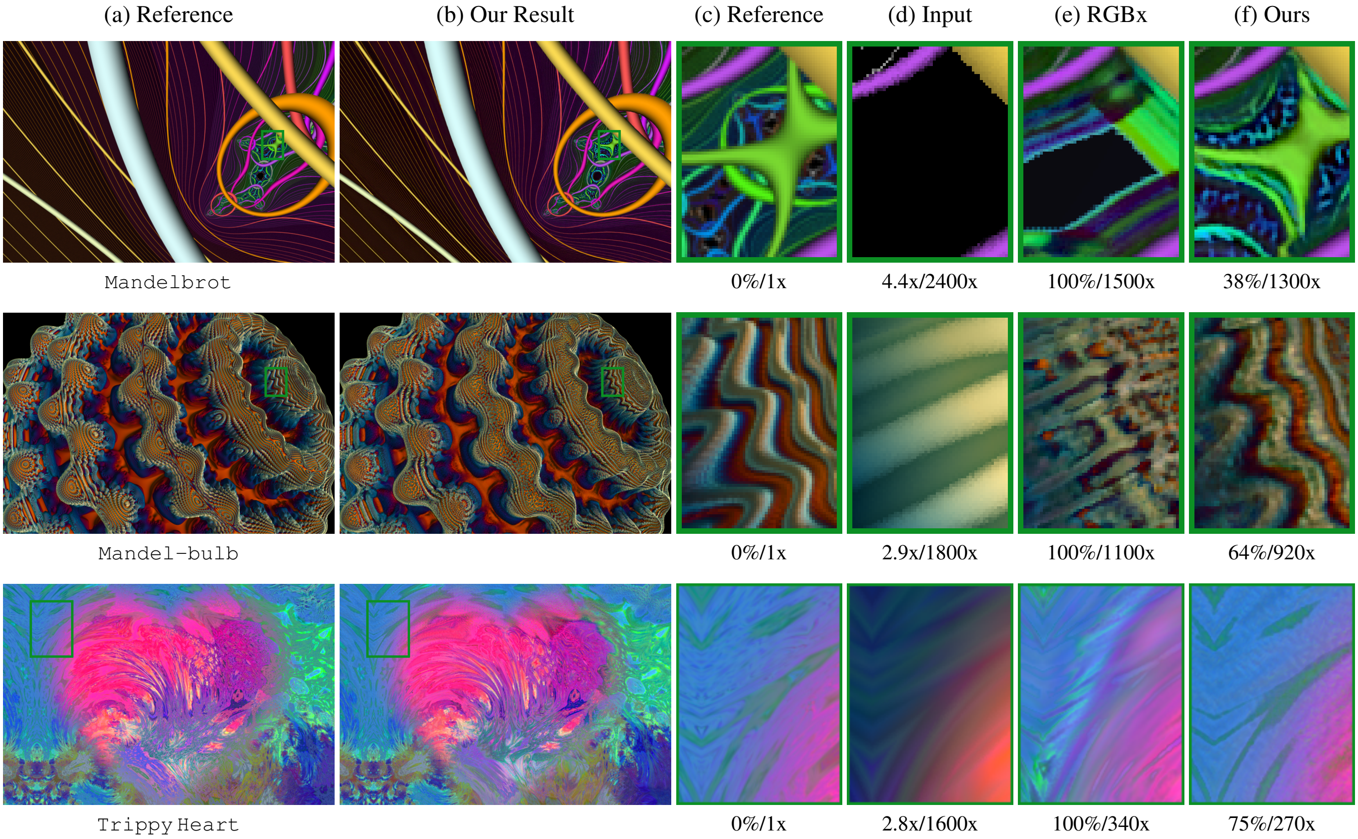


Figure 5: *Learning from simplified shaders* `Mandelbrot`, `Mandel-bulb` *and* `Trippy Heart`. *Errors and speedups are reported as in Figure 1. In* `Mandelbrot` *our method better reconstructs missing regions due to oversimplification in the input. In* `Mandel-bulb` *our method better recovers the orientation of the texture. In* `Trippy Heart` *ours correctly recovers the color.*

We evaluate our method and compare it against two baselines. The first baseline is RGBx described before. Our second baseline is supersampling. Supersampling draws a number of samples at each pixel, evaluates the shader to obtain RGB colors for each sample, and takes the mean of the colors. We supersample by choosing a constant sample budget per pixel to achieve approximately the same run time as ours, including the overhead for neural network inference.

Training for 400 epochs typically takes between 6 and 32 hours. However, the `Oceanic` shader is slower, and takes about 7 days to train. Note that all shaders are trained using the same process over an identical architecture with a similar number of input channels; therefore the great variation in training time derives primarily from the cost of sampling from shader programs, not from learning.

In terms of the arithmetic average over all shaders, our method has a relative perceptual error of 67% compared to the RGBx baseline. A different baseline, Supersampling, is consistently worse than RGBx, with relative perceptual error ranging from 3x to 21x compared to RGBx (Appendix Table 3). We believe the dramatic improvements in relative perceptual error of our method over the baselines corresponds with the qualitatively better reconstruction of high-frequency details that we observe in the renderings (Figure 4 c-f). Our supplemental video shows comparisons of several of these shaders rendered with a moving camera.

5.2. Reconstructing Simplified Shaders

We also explore a more challenging task: learning to reconstruct the appearance of a shader from its simplified variant. Shader simplification is commonly used as a lossy optimization that improves runtime while approximating the output of the original program. However, simplified programs often lose texture or geometry detail as compared with the original. For example, the simplified versions of `Mandelbrot` and `Mandel-bulb` shown in Figure 5d look obviously different from their original counterparts in Figure 5c. We therefore propose an application that learns to recover the denoised output of the original shader from the traces of the simplified shader program sampled at 1 SPP. To our knowledge, this paper is the first to propose this learning task.

We use two different techniques to simplify the shader programs: genetic programming simplification [SaMWL11] (on `Bricks`) and loop perforation [SDMHR11] (on all other shaders). Because the model needs to synthesize unseen texture, we use a spatial discriminator for this application, described in Appendix D. Training for 800 epochs takes between 10 and 60 hours. Similar to the denoising application, the great variation in training time mostly comes from generating input samples from the shader. Our method has on average 62% perceptual error compared to the RGBx baseline.

5.3. Postprocessing Filters

Our method can be useful for learning not only denoising, but also applying additional image-space postprocessing filters. We implement two postprocessing filters on the CPU: an edge aware sharpening filter [PHK11] and defocus blur [Rok93]. The network learns simultaneously to denoise and apply the postprocessing filter on the GPU. Figure 6 shows learning a defocus blur filter on `Mandelbulb`, and learning a sharpening filter on simplified `Trippy Heart`. Our approach reproduces the complex effect more faithfully, as compared to RGBx, and the average relative perceptual error for ours is 74% of that of RGBx.

5.4. Learning to Approximate Simulation

Departing from learning from procedural pixel shader programs, we also explore learning to predict the future for shader programs that perform simulations. This section describes simulation of flocking behavior, while Appendix I presents learning to approximate fluid simulations.

Our shader simulates a flock of “boids” [Rey87] which emulate the flocking behavior of birds or fish. Each boid has a 4-vector state representing 2D position and velocity. For a flock of K boids, the simulation program takes input of a $K \times 4$ tensor that represents each individual boid’s initial state, then updates the state based on repulsion and alignment forces. The updated state then becomes the input to the next simulation step, and so forth. The interaction between boids forms a complex flocking behavior that is difficult to predict. We run the ground truth simulation using a small δ step size: 2×10^6 steps with $\delta = \frac{1}{600}$ s, targeting 20δ per frame at 30fps. During training we further augment the data by randomly permuting boid indices. The learning task is to correct the simulation output from a larger time step $m \cdot \delta$ in order to approximate the boids’ states as if the simulation ran m times for step size δ. (We train with $m \in [20, 64]$.) We compare our method with two baselines: a naive baseline that directly takes the larger step simulation without any correction, and an input/output (I/O) baseline that uses the input and the output of the larger step simulation as the input to a neural network. The learning model is a combination of 1D convolution layers with 3 fully connected layers. For details please refer to Appendix E. For reported results, we simulate 40 individual boids and log every program trace from the boids program. We choose a larger program trace length than for the pixel shaders because the simulation considers all pairwise interactions between boids, and a larger program trace budget better captures these interactions.

In Figure 2 and 7, we show that ours always outperforms baselines numerically and visually, even when the step size is extrapolated outside the training range ($m \in [16, 128]$). The supplemental video shows that ours recovers individual boids’ interaction behaviors more faithfully with a step size of $m = 20$, while the I/O baseline mainly learns the average position and velocity for the entire flock but fails to recover a reasonable distance between the boids.

6. Trace Analysis

This section presents a series of analyses that help to understand how program traces are beneficial for learning. We start by analyzing which trace features are contributing the most to a learned model. Based on trace importance, we then investigate which subset of the trace can be used for learning. We empirically find that if one cannot afford to first execute and learn from the full shader trace, then the Uniform subsampling used throughout Section 5 always gives reasonable performance, and we were not able to find any strategy that consistently outperforms Uniform. However, if one is able to train an additional inital network that first uses the full program trace, then we can do better than Uniform, using a strategy that we call Oracle that selects important features. Finally, we show that multiple shaders can be trained together with a shared denoising network and a lightweight shader-specific encoder.

6.1. Which Trace Features Matter in a Learned Model?

We characterize the importance of the trace features by quantifying the change in training loss when removing each of the trace inputs. Inspired by Molchanov et al. [MMT*19, MTK*16], we used the first order Taylor expansion to approximate importance of each input trace feature. Specifically, for a model trained with loss L and trace length $\mathcal{T}$, the importance score Θ of the input trace feature $\mathbf{z}^l$ ($l = 1, ..., \mathcal{T}$) with image dimension $M \times N$ across K examples is:

$$\Theta(\mathbf{z}^l) = \frac{1}{K} \sum_{k=1}^{K} \left| \frac{1}{M \cdot N} \sum_{m=1}^{M} \sum_{n=1}^{N} \frac{\partial L}{\partial \mathbf{z}^l_{m,n}} \cdot \mathbf{z}^l_{m,n} \right| \qquad (2)$$

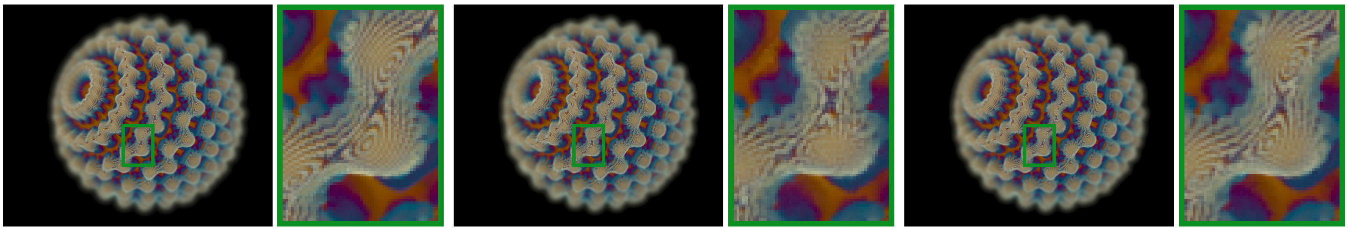
We evaluate Equation (2) on the denoising model for two shaders: `Bricks` and `Mandelbrot`. Only a small fraction of the trace results in a very high importance score. We manually inspect what the top 10% most important trace features represent and verified that the learned importance corresponds to human intuition. For example in `Bricks`, we found the most important traces include features that determines the distance to the nearest brick edges and the Boolean condition that decides whether the pixel is insde the mortar: this helps prevent edges from being broken. In `Mandelbrot`, we found the trace that controls the complex number computation for almost every iteration are among the most important features: a breakdown of such information at each level could help the model to better denoise between nearby structures.

6.2. Which Subset of the Trace to Use for Learning?

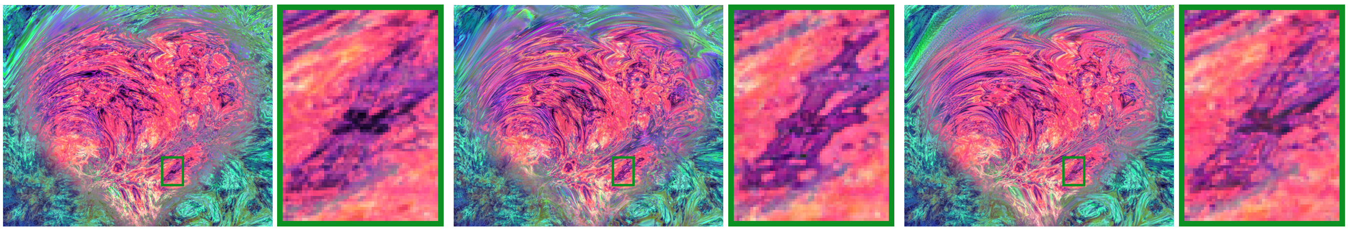
As discussed in Section 3.2, program traces can be arbitrarily long, and we could input only a subset of the trace for efficient learning and inference, such as Uniform subsampling used in Section 5. Therefore, a natural question to ask is: given a fixed input trace length budget, what subsets of the program trace are good for learning? The best way to answer this question is to enumerate all possible subsets of the program trace and train a separate model for each. However, for a shader program that has $\mathcal{T}$ traces before subsampling and a fixed input budget $\mathcal{N}$, this strategy will introduce combinatoric $\binom{\mathcal{T}}{\mathcal{N}}$ learning tasks, which is intractable.

To investigate how different subsets of the trace could affect learning in a practical fashion, we propose subsampling strategies we call Oracle and Opponent. Both the Oracle and Opponent strategies are based on the feature importance score (Section 6.1) from a Full Trace model trained with all of the program trace. Oracle always chooses the traces that have the highest importance scores, while Opponent always chooses the ones with the lowest scores.

(a) Reference (b) RGBx Baseline (c) Our Result



`Mandel-bulb` blur: 0%/1x 100%/700x (baseline) 65%/590x

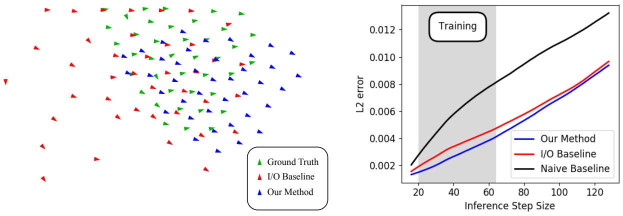


`Trippy Heart` simplified sharpen: 0%/1x 100%/340x (baseline) 80%/270x

Figure 6: *Learning postprocessing effects. The reference solution (a) shows the result of a postprocessing filter applied to a low-noise shader rendering sampled at 1000* SPP. *Both RGBx baseline (b) and our method (c) approximates the reference at 1* SPP. *Our method recovers more faithfully the thin structure in* `Mandel-bulb` *and the color pattern in* `Trippy Heart`. *We report relative perceptual error and speedup as in Figure 1.* `Mandel-bulb` *is gamma corrected so it can be viewed comfortably on darker displays.*

In an analogy to the lottery ticket hypothesis [FC18], we hypothesize that the Oracle exploits a winning “lottery ticket" found within the Full Trace model, and selects out the relevant trace subset: a “lottery ticket trace." The Opponent likewise selects losing tickets.

To better understand the trade-offs associated with the subsampled trace length, we experimented with varying trace lengths using Opponent, Uniform, and Oracle subsampling and compare them with the RGBx baseline, as shown in Figure 8. For each shader, the trace is subsampled by a relative sample budget compared to the full program trace length $\mathcal{T}$ (e.g. $\mathcal{N} = \mathcal{T}/2$, $\mathcal{T}/4$). Under a fixed budget $\mathcal{N}$, in most cases the inference error decreases in the ordering of Opponent, Uniform, Oracle. This corresponds to our intuition because Oracle selects traces that are beneficial to training based on prior knowledge from the Full Trace model, and similarly Opponent selects traces that are unimportant based on the same prior knowledge. Statistically, our hypotheses that Uniform outperforms Oracle, and Opponent outperforms Uniform each have p-values 7.2×10^{-4} and 5.9×10^{-3}, respectively. These are smaller than a threshold of 0.025 determined by correcting the traditional p threshold of 0.05 for the two comparisons, so we conclude that the ordering *Oracle outperforms Uniform outperforms Opponent* is significant. For details please see Appendix J.



(a) Visualization (b) Error Analysis

Figure 7: *(a) Visualization for flocks of boids. Both I/O baseline (red) and our method (blue) starts from the same initial state and have taken 80 inference steps with step size 20. Our mean position as well as flocking behavior is more faithful to the ground truth (green). (b) We plot average error as a function of step size, where training ranges from step size 20 to 64 (gray). Ours consistently outperforms both I/O and a more naive baseline, in the training range and beyond it.*

It is also worth noting that even when $\mathcal{N}$ is small (e.g. the left-most two data points in the plots corresponds to $\mathcal{N}$ below 50), the extra information from the program trace can still substantially reduce the relative perceptual error without significant extra cost in inference time. Because the x-axis in the plot is on a log scale, the actual performance gain would have a more drastic slope starting from RGBx to a small $\mathcal{N}$. Additionally, the current comparison is advantageous to RGBx as its learning capacity matches that of the Full Trace model as discussed in Section 5, which is more capacity than any of the subsampled models in Figure 8.

In practice, subsampling strategies can be chosen based on resources allowed for training and inference. If there is no limit at all, training a model with the Full Trace can always give the best performance. If $\mathcal{N}$ is only limited by inference time, but extra cost and memory can be permitted during training, one could use the Oracle strategy. However, when training also becomes a practical concern, our results suggest that without actually learning from the full trace in advance, there may not be a single subsampling strategy that could consistently outperform all others, as discussed in Section 3.2. Thus, Uniform subsampling provides an effective proxy

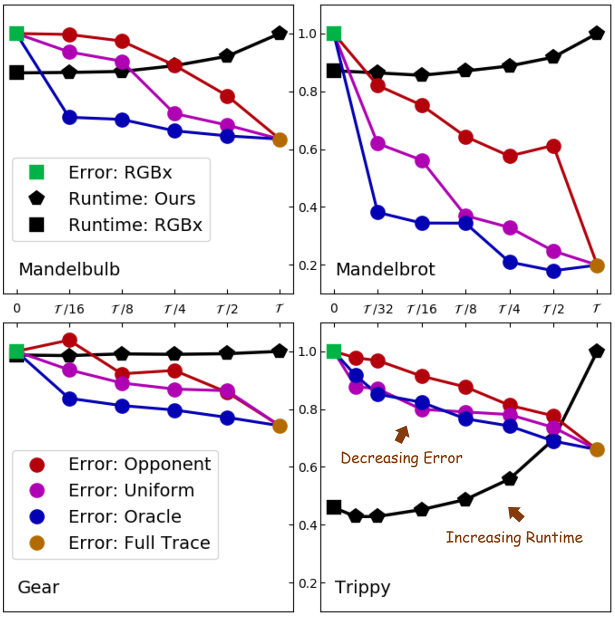


Figure 8: *Error vs. Time trade-off for Opponent, Uniform, and Oracle subsampling strategies with varying trace length. For each shader with $\mathcal{T}$ program traces, we subsample the trace such that the actual input trace length $\mathcal{N}$ is equal to $\mathcal{T}$/2, $\mathcal{T}$/4, $\mathcal{T}$/8, etc., and the x-axis shows each model's relative trace ratio compared to the full program trace. Circle shows perceptual error relative to the RGBx baseline (green square) for Opponent (red), Uniform (purple), Oracle (blue), and Full Trace (gold). Black pentagon shows relative inference time (including shader program runtime and network inference) for each $\mathcal{N}$ relative to the runtime of the Full Trace model. Similarly black square shows relative inference time for the RGBx baseline. Note that the x-axis is on a log scale in relative trace length compared to the Full Trace model, therefore although the relative error plot appears linear as $\mathcal{N}$ increases, the actual performance improvement is faster at the beginning of the plot: adding only a few traces quickly improves performance.*

that follows the performance of Oracle, and always outperforms the worst-case scenario Opponent.

6.3. Can Multiple Shaders be Learnt Together?

In this section, we explore whether part of the model can be shared across shaders with the same task. Because program traces are unique per shader, we propose to train a separate shallow encoder for each of the shaders, followed by a task-specific model shared across shaders. The setup is similar to the source-aware encoder by Vogels et al. [VRM*18].

Four shaders (`Mandelbrot`, `Mandel-bulb`, `Gear`, and `Trippy Heart`) are trained together for the denoising task. The encoder consists of four 1x1 convolutional layers, where the first layer outputs $\mathcal{K}$ channels and the rest output 48 channels. In our method, $\mathcal{K} = 48$ while in the RGBx baseline $\mathcal{K}$ varies similarly as in Section 5. The encoder is identical to the four 1x1 convolutions that analyze the input program trace in Figure 3. The 48-dimensional encoding then inputs to a shared denoising network, whose architecture is identical to Figure 3 excluding the four initial 1x1 convolutions. All four shaders use Uniform subsampling to bring their $\mathcal{N}$ to be closest to 200. Training alternates between the 4 shaders after each epoch, and each shader is trained for 400 epochs.

We report the error statistics for the shared model in Table 1. Ours has on average 60% perceptual error compared to the RGBx baseline. Although one might expect this experiment to benefit the RGBx baseline as the RGBx features are more similar, in fact, ours outperforms RGBx in all cases.

7. Conclusion, Limitations & Future Work

This paper proposes the idea of learning from shader program traces. It evaluates this idea in a range of learning contexts: denoising, simplified shaders, postprocessing filters and simulation. We describe a compiler that can produce program traces suitable for learning, as well as practical considerations like how to handle large traces and how to process the trace data to make it amenable to learning. Our method is agnostic to the learning architecture, loss function and training process; however, we also discuss a particular set of these that worked well in our experiments. We evaluate our method on a range of shaders, over which it compares favorably with baselines. We also analyze which features are important in the trace, and explain how one can select subsets of the trace for learning.

Our method has several limitations, which offer potential avenues for future work. First, as with many neural network based approaches, the inference time is not negligible. For example, for the denoising task, simple, fast shaders can draw sufficiently many samples via supersampling so as to outperform inference. Likewise for the simplified shader tasks, one could use the time budget for network inference to instead compute multiple samples of the original more expensive shader. Future research might address this by developing specialized networks that are more efficient for inference, along similar lines as the method of Gharbi et al. [GCB*17]. Another limitation of our TensorFlow implementation is that the operation of collecting program traces and concatenating them into one single tensor is not particularly efficient, and is a major cause of the inference time increasing with the trace length (Figure 8). Additionally, TensorFlow is efficient for deep learning models, but less efficient for shader computations. For example, for shaders with complex BRDFs, branching generated by varying number of ray bounces per pixel may become a major bottleneck in TF. We believe further engineering efforts could alleviate these bottlenecks, for example by using compiled GLSL as a frontend renderer prior to the inference process.

The experiments described this paper were performed using computer graphics shaders. Future work could explore how well the ideas introduced herein generalize to other kinds of programs that can rely on (and tolerate) approximate solutions, for example those relying on stochastic algorithms or Markov-like decision processes.

References

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: *Real-time rendering*. CRC Press, 2008. 2

[BA83] Burt P., Adelson E.: The laplacian pyramid as a compact image code. *IEEE Transactions on communications 31*, 4 (1983), 532–540. 13

[BJB*19] Bylinskii Z., Judd T., Borji A., Itti L., Durand F., Oliva A., Torralba A.: Mit saliency benchmark. http://saliency.mit.edu/, 2019. 13

[CBSC16] Cornia M., Baraldi L., Serra G., Cucchiara R.: A deep multi-level network for saliency prediction. In *2016 23rd International Conference on Pattern Recognition (ICPR)* (2016), pp. 3488–3493. 13

[CKS*17] Chaitanya C. R. A., Kaplanyan A. S., Schied C., Salvi M., Lefohn A., Nowrouzezahrai D., Aila T.: Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph. 36*, 4 (July 2017), 98:1–98:12. 1, 3, 6

[CLS19] Chen X., Liu C., Song D.: Execution-guided neural program synthesis. In *International Conference on Learning Representations* (2019). URL: `https://openreview.net/forum?id=H1gfOiAqYm`. 2

[CSS18] Chen L., Sultana S., Sahita R.: Henet: A deep learning approach on intel® processor trace for effective exploit detection. *CoRR abs/1801.02318* (2018). URL: `http://arxiv.org/abs/1801.02318`, `arXiv:1801.02318`. 2

[CXK17] Chen Q., Xu J., Koltun V.: Fast image processing with fully-convolutional networks. In *The IEEE International Conference on Computer Vision (ICCV)* (Oct 2017). 4

[DBLW15] Dorn J., Barnes C., Lawrence J., Weimer W.: Towards automatic band-limited procedural shaders. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 77–87. 3

[FC18] Frankle J., Carbin M.: The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018). 9

[FH93] Feiler P. H., Humphrey W. S.: Software process development and enactment: Concepts and definitions. In *Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the* (1993), IEEE, pp. 28–40. 2

[GCB*17] Gharbi M., Chen J., Barron J. T., Hasinoff S. W., Durand F.: Deep bilateral learning for real-time image enhancement. *ACM Transactions on Graphics (TOG) 36*, 4 (2017), 118. 3, 10

[GKB*18] Ganin Y., Kulkarni T., Babuschkin I., Eslami S. M. A., Vinyals O.: Synthesizing programs for images using reinforced adversarial learning. *CoRR abs/1804.01118* (2018). URL: `http://arxiv.org/abs/1804.01118`, `arXiv:1804.01118`. 2

[GLA*19] Gharbi M., Li T.-M., Aittala M., Lehtinen J., Durand F.: Sample-based monte carlo denoising using a kernel-splatting network. *ACM Transactions on Graphics (TOG) 38*, 4 (2019), 1–12. 3, 5

[Goo17] Goodfellow I.: NIPS 2016 tutorial: Generative adversarial networks. *CoRR abs/1701.00160* (2017). `arXiv:1701.00160`. 14

[HFTF15] He Y., Foley T., Tatarchuk N., Fatahalian K.: A system for rapid, automatic shader level-of-detail. *ACM Transactions on Graphics (TOG) 34*, 6 (2015), 1–12. 3

[HPG*19] Hahn C., Phan T., Gabor T., Belzner L., Linnhoff-Popien C.: Emergent escape-based flocking behavior using multi-agent reinforcement learning. *CoRR abs/1905.04077* (2019). URL: `http://arxiv.org/abs/1905.04077`, `arXiv:1905.04077`. 3

[IZZE17] Isola P., Zhu J.-Y., Zhou T., Efros A. A.: Image-to-image translation with conditional adversarial networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on* (2017), IEEE, pp. 5967–5976. 4, 14

[JMKO20] Jeppu N. Y., Melham T., Kroening D., O'Leary J.: Learning concise models from long execution traces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), pp. 1–6. `doi:10.1109/DAC18072.2020.9218613`. 2

[KWGB17] Kummerer M., Wallis T. S., Gatys L. A., Bethge M.: Understanding low-and high-level contributions to fixation prediction. In *Proceedings of the IEEE International Conference on Computer Vision* (2017), pp. 4789–4798. 13

[KZP*20] Kim S. W., Zhou Y., Philion J., Torralba A., Fidler S.: Learning to Simulate Dynamic Environments with GameGAN. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun. 2020). 3

[Lar93] Larus J. R.: Efficient program tracing. *Computer 26*, 5 (1993), 52–61. 2

[LGA*18] Li T.-M., Gharbi M., Adams A., Durand F., Ragan-Kelley J.: Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 139. 3

[LHAY17] Li Y., Huang J.-B., Ahuja N., Yang M.-H.: Joint image filtering with deep convolutional networks. *arXiv preprint arXiv:1710.04200* (2017). 3

[LKD*16] Li H., Kadav A., Durdanovic I., Samet H., Graf H. P.: Pruning filters for efficient convnets. *CoRR abs/1608.08710* (2016). `arXiv:1608.08710`. 3

[LNLB16] Lou L., Nguyen P., Lawrence J., Barnes C.: Image perforation: Automatically accelerating image pipelines by intelligently skipping samples. *ACM Transactions on Graphics (TOG) 35*, 5 (2016), 153. 4

[LWL17] Luo J.-H., Wu J., Lin W.: Thinet: A filter level pruning method for deep neural network compression. In *The IEEE International Conference on Computer Vision (ICCV)* (Oct 2017). 3

[MMT*19] Molchanov P., Mallya A., Tyree S., Frosio I., Kautz J.: Importance estimation for neural network pruning. *CoRR abs/1906.10771* (2019). URL: `http://arxiv.org/abs/1906.10771`, `arXiv:1906.10771`. 3, 8

[MST*20] Mildenhall B., Srinivasan P. P., Tancik M., Barron J. T., Ramamoorthi R., Ng R.: Nerf: Representing scenes as neural radiance fields for view synthesis. *arXiv preprint arXiv:2003.08934* (2020). 2, 15, 16

[MTK*16] Molchanov P., Tyree S., Karras T., Aila T., Kautz J.: Pruning convolutional neural networks for resource efficient transfer learning. *CoRR abs/1611.06440* (2016). URL: `http://arxiv.org/abs/1611.06440`, `arXiv:1611.06440`. 3, 8

[NAM*17] Nalbach O., Arabadzhiyska E., Mehta D., Seidel H.-P., Ritschel T.: Deep shading: Convolutional neural networks for screen-space shading. 2

[Per01] Perlin K.: Noise hardware. *Real-Time Shading SIGGRAPH Course Notes* (2001). 6

[PHK11] Paris S., Hasinoff S. W., Kautz J.: Local laplacian filters: Edge-aware image processing with a laplacian pyramid. In *ACM SIGGRAPH 2011 Papers* (2011), SIGGRAPH '11, pp. 68:1–68:12. 8

[Rey87] Reynolds C. W.: Flocks, herds, and schools: A distributed behavioral model. *SIGGRAPH Computer Graphics 21*, 4 (July 1987), 25–34. 2, 8

[RF16] Reed S., Freitas N. D.: Neural programmer-interpreters. *CoRR abs/1511.06279* (2016). 2

[Rok93] Rokita P.: Fast generation of depth of field effects in computer graphics. *Computers & Graphics 17*, 5 (1993), 593–595. 8

[RTHG16] Ritchie D., Thomas A., Hanrahan P., Goodman N.: Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances in Neural Information Processing Systems* (2016), Lee D., Sugiyama M., Luxburg U., Guyon I., Garnett

R., (Eds.), vol. 29, Curran Associates, Inc., pp. 622–630. URL: https://proceedings.neurips.cc/paper/2016/file/40008b9a5380fcacce3976bf7c08af5b-Paper.pdf. 2

[SaMWL11] SITTHI-AMORN P., MODLY N., WEIMER W., LAWRENCE J.: Genetic programming for shader simplification. *ACM Trans. Graph. 30* (12 2011), 152. 3, 7

[SDMHR11] SIDIROGLOU-DOUSKOS S., MISAILOVIC S., HOFFMANN H., RINARD M.: Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), ACM, pp. 124–134. 3, 4, 7

[STH*19] SITZMANN V., THIES J., HEIDE F., NIESSNER M., WETZSTEIN G., ZOLLHÖFER M.: Deepvoxels: Learning persistent 3d feature embeddings. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE* (2019). 2

[TZN19] THIES J., ZOLLHÖFER M., NIESSNER M.: Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics 2019 (TOG)* (2019). 2, 3

[TZT*20] THIES J., ZOLLHÖFER M., THEOBALT C., STAMMINGER M., NIESSNER M.: Image-guided neural object rendering. In *International Conference on Learning Representations* (2020). URL: https://openreview.net/forum?id=Hyg9anEFPS. 2

[VRM*18] VOGELS T., ROUSSELLE F., MCWILLIAMS B., RÖTHLIN G., HARVILL A., ADLER D., MEYER M., NOVÁK J.: Denoising with kernel prediction and asymmetric loss functions. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 124. 1, 2, 3, 6, 10

[WLZ*18] WANG T.-C., LIU M.-Y., ZHU J.-Y., LIU G., TAO A., KAUTZ J., CATANZARO B.: Video-to-video synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)* (2018). 4, 14

[Won16] WONG J.: Ray marching and signed distance functions. http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/, 2016. Accessed: 2019-01-12. 6

[WXCT19] WERHAHN M., XIE Y., CHU M., THUEREY N.: A multi-pass GAN for fluid flow super-resolution. *CoRR abs/1906.01689* (2019). URL: http://arxiv.org/abs/1906.01689, arXiv:1906.01689. 3

[WYY*14] WANG R., YANG X., YUAN Y., CHEN W., BALA K., BAO H.: Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics (TOG) 33*, 6 (2014), 1–11. 3

[WZZH18] WU H., ZHENG S., ZHANG J., HUANG K.: Fast end-to-end trainable guided filter. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 1838–1847. 3

[XFCT18] XIE Y., FRANZ E., CHU M., THUEREY N.: tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 95. 2

[YB18] YANG Y., BARNES C.: Approximate program smoothing using mean-variance statistics, with application to procedural shader bandlimiting. *Comput. Graph. Forum 37*, 2 (2018), 443–454. 3

[ZIE*18] ZHANG R., ISOLA P., EFROS A. A., SHECHTMAN E., WANG O.: The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR* (2018). 1, 5, 6, 16

[ZJL*15] ZWICKER M., JAROSZ W., LEHTINEN J., MOON B., RAMAMOORTHI R., ROUSSELLE F., SEN P., SOLER C., YOON S.-E.: Recent advances in adaptive sampling and reconstruction for monte carlo rendering. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 667–681. 3

Supplemental Material

Learning from Shader Program Traces

(in review for Eurographics 2022)

Note to reviewers: Here we include as appendices some details we removed from the main body of the paper to reduce overall length. Any or all them could be included or omitted as needed.

Appendix A: Classifying Iterative Improvement Loops

Here we present our detailed classification rules for iterative improvement loops. For a loop variable X at iteration n, we will denote its value as X_n.

Definition 1 A loop variable X is iterative additive if it matches the following pattern or its equivalent forms:

$$X_n = X_{n-1} + Z \tag{3}$$

Here Z can be any arbitrary variable.

Definition 2 A variable Y is dependent on an iterative additive variable X if it matches the following pattern or its equivalent forms:

$$Y_n = \text{select}(cond, Y_{n-1}, f(X_n, X_{n-1}, C)) \tag{4}$$

Here, $cond$ is an arbitrary Boolean variable, f is an arbitrary function, and C is a variable computed outside the loop, i.e. C can be viewed as constant inside the loop.

Definition 3 A loop variable X is an output variable if for any iteration n, its value X_n is used outside the loop.

Definition 4 A loop is classified as an iterative improvement loop if all of its output variables are either iterative additive or are dependent on an iterative additive variable.

Appendix B: Details for Whitening the Collected Trace

For each intermediate value in the program trace, we clamp extreme values and rescale others by the following process.

We clamp extreme values by collecting the statistics for the intermediate values' distribution at training time. For each intermediate value, we first decide whether its distribution merits clamping. If we detect that the distribution has only a small number of finite, discrete values (10 or fewer), we do not apply clamping to the corresponding intermediate value. For the rest of the intermediate values, we first discard infinite values and then find from their distributions the lowest and the highest pth percentiles, denoted P_0 and P_1, and use these to compute clamping thresholds. Next we clamp all values to the range $[P_0 - \gamma(P_1 - P_0), \quad P_1 + \gamma(P_1 - P_0)]$. We also set NaN values to the low end of this range. Empirically, we found in our experiments that $p = 5$ and $\gamma = 2$ work well, and we use these values for all results. Finally, for each intermediate feature, we rescale the clamped values to the fixed range [-1,1], and record the corresponding scale and bias used. In both training and testing, the collected program traces are used directly by applying the same precomputed scale and bias, but the values will be clamped to range [-2, 2] to allow data extrapolation.

Appendix C: Generating the Dataset

Our experiments generates the dataset from 800 images for training, 80 images for validation and 30 images for testing (each 960×640). Although this training set size is small relative to typical deep learning tasks, we address this concern in Section 4.3. The training images are generated with random camera poses, while testing images are divided into two groups: 20 *similar* distance images with camera pose sampled from the same distribution as the training set, as well as 10 *different* distance images that are closer or further than the training set. For some shaders, (`Trippy Heart`, `Mandelbrot`, `Mandel-bulb`, `Venice` and `Oceanic`), a periodic time parameter also changes the shader appearance, which is sampled from the same distribution for both training and testing datasets.

We find it beneficial to further divide the training and validation set into tiles. One advantage is that certain features in the shader may be visually salient to humans, so we can emphasize such features to ensure they are learned well. In principle this could be accomplished with automatic saliency models (e.g. [KWGB17, BJB*19, CBSC16]). However, off-the-shelf saliency models are trained for natural imagery whereas our shaders are non-photorealistic, and therefore we combine both a saliency model [CBSC16] and a traditional Laplacian pyramid representation to robustly and automatically select salient tiles. Another benefit of tiled training is that it reduces memory, and it also accelerates convergence, because we can use larger mini-batches with more varied content within the same GPU memory to obtain a gradient estimator with lower mean squared error.

We sample training and validation tiles as follows. We first generate saliency maps for each of our 800 training images and 80 validation images using Cornia et al. [CBSC16]. Saliency models usually incorporate a center bias that tends to give lower saliency scores to pixels closer to image boundaries. This behavior is not ideal for our framework because our training images are generated from randomly sampled camera poses so that salient content could appear anywhere in the image. Therefore, we run the saliency model on images with an extended field of view (each 1280×960) where the center patches of size 960×640 are our original training images. This allows every pixel in the original training dataset to be away from image boundaries to avoid center bias in the resulting saliency maps.

We then subdivide each of the training and validation images into six 320×320 tiles. For each tile, we estimate its intensity on low, middle and high frequencies by taking the average over its first, third, and fifth level of the Laplacian pyramid [BA83]. Together with the average saliency score, these four metrics can be combined to robustly sample salient and interesting tiles for learning.

Next, we use identical sampling rules to sample one-quarter of the sampling budget from each of the four metrics. For each metric, we rank the tiles according to their associated score and only sample from the tiles whose score is within the top 25% nonzero scores. The score of the qualified tiles will further be normalized to [0, 1], and each tile will be sampled with a probability proportional to the normalized score.

Apart from the rules described above, we find it helpful to also include a small portion of constant color tiles in the training dataset, e.g. the black background in `Bricks` Figure 4. These uninformative and constant color tiles can be easily selected from a low color variance threshold. Although some salient tiles already contain both informative and uninformative regions, they are usually close to object silhouettes and could still pose challenges when extrapolating to uninformative regions far away from the object.

We sample a total of 1200 tiles for training and 120 tiles for validation. If the shader does not contain constant color tiles, all of the sampling budget will be used to equally sample from the 4 saliency metrics described above. Otherwise, only 95% of the sampling budget will be sampled from saliency, and another 5% will be sampled from low color variance tiles. Testing still relies on 30 full images.

Appendix D: Details for GAN Models

Our spatial GAN model is a conditional GAN, where the conditional labels are the RGB channels of the 1 SPP rendering from the shader program, denoted as c_x. Because c_x is already part of the program trace, we directly use the model from Figure 3 as our generator and the generator's output is naturally conditioned on c_x. We then train the model to match the ground truth denoted as c_y. Additionally, we used a patchGAN architecture similar to that of Isola et al. [IZZE17] with receptive field 34×34 as our discriminator D.

Our temporal GAN model uses a similar architecture as the spatial GAN with modifications following [WLZ*18]. The generator is conditioned on imagery from three consecutive frames: the current predicted frame and the two previous ones. This involves five 3-channel images as conditional labels: shader RGB output from all three frames plus the generator's output from the two previous frames. Because neither the shader output nor the generator output from the previous two frames is part of the program trace for the current frame, we modified the generator architecture in Figure 3 to concatenate the additional four conditional label images *after* the feature reduction layer. The rest of the architecture remains the unchanged. We use the same discriminator architecture as for our spatial GAN, but it takes an input of sequences of frames and their corresponding conditional labels.

We now introduce the variation on the basic loss function (equation (1)) that incorporates the GAN loss. We use a modified cross entropy loss [Goo17] for both spatial and temporal GAN models. Our spatial GAN model is conditioned on the RGB channels of the shader program c_x to approximate the distribution of the ground truth c_y, while our temporal GAN loss is applied to sequences $\widetilde{c_x}$ and $\widetilde{c_y}$. The training objective (that we minimize) for generator L_G and loss for spatial discriminator L_{D_S} can be expressed as:

$$\begin{aligned} L_G = {} & L_b - \beta\, \mathbb{E}_{c_x} \log(D_S(G(c_x), c_x)) \\ L_{D_S} = {} & -\mathbb{E}_{c_x, c_y} \log(D_S(c_y, c_x)) \\ & - \mathbb{E}_{c_x} \log(1 - D_S(G(c_x), c_x)) \end{aligned} \tag{5}$$

Similarly, the training objective on temporal sequences for generator L_G and temporal discriminator L_{D_T} can be expressed as:

$$\begin{aligned} L_G = {} & L_b - \beta\, \mathbb{E}_{\widetilde{c_x}} \log(D_T(G(\widetilde{c_x}), \widetilde{c_x})) \\ L_{D_T} = {} & -\mathbb{E}_{\widetilde{c_x}, \widetilde{c_y}} \log(D_T(\widetilde{c_y}, \widetilde{c_x})) \\ & - \mathbb{E}_{\widetilde{c_x}} \log(1 - D_T(G(\widetilde{c_x}), \widetilde{c_x})) \end{aligned} \tag{6}$$

The parameter β is a weight that balances between the GAN loss and the regular color and perceptual loss in equation (1). In all our experiments with GAN loss, we fix $\beta = 0.05$ to roughly balance the magnitude of gradients from all loss terms. Note in equation (6) we did not include spatial discriminators for simplicity. But it is possible to combine both equation (5) and equation (6). For example, in Appendix F, we trained on both discriminators to produce a temporally coherent model for simplified shaders.

We also skip the back-propagation on the GAN loss for any minibatch with constant color to avoid training instability.

Appendix E: Details for the Boids Simulation

The boids simulation Section 5.4 works with non-imagery data, and we therefore uses a combination of 1D convolution and fully connected layers for that learning task. The input to the network has size $B \times \mathcal{N}$ where B represents the number of boids (40 in our experiments) and $\mathcal{N}$ represents either the length of the program trace in our method or 8 for the I/O baseline. We first reduce the dimensionality of the trace to $\mathcal{K}$ using a 1D convolution with kernel size one, followed by 3 additional 1D convolutions with kernel size one and 48 output channels. This is an analogy to the 2D feature reduction layer and 1x1 convolutions described in Figure 3, where $\mathcal{K} = 48$ for our method and $\mathcal{K} = 1173$ for the I/O baseline to match the number of trainable weights in both models. We then flatten the $B \times 48$ tensor as an input to a 3 layer fully connected network, where each layer has 256 hidden neurons, and an output fully connected layer with the number of neurons being $B \cdot 4$, representing the output state for each boid.

We learn a four-channel state (2D position and velocity) for each boid, rather than RGB. Therefore we use only L_2 loss on these coordinates after separately normalizing position and velocity over the training set.

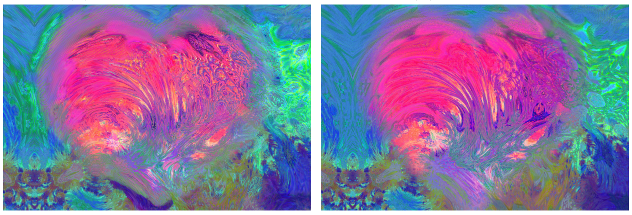
Appendix F: Training Temporally Coherent Sequences

Temporal coherence in a graphics or vision context refers to there being a strong correlation between each frame and the next. Training only on individual images can introduce temporal incoherence for rendered video. One straightforward fix would be to apply a temporal filter to the output sequences to blur out the noise. Alternatively, we implemented a temporal discriminator to directly train temporally coherent sequences using a training scheme similar to that of Wang et al. [WLZ*18]. Each frame in a sequence is synthesized conditioned on two previous frames. In training, frames are synthesized in groups of six consecutive frames, relying on eight-frame ground truth sequences to be able to bootstrap the initial frame. We train temporally coherent sequences both for the task of denoising and learning from simplified programs, and compare with an RGBx baseline as in Sections 5.1 & 5.2. A summary of quantitative error is shown in Table 2. In all cases ours outperforms

Table 2: *Error statistic for learning temporally coherence sequences. Metrics reported are similar as in Table 1. The temporal application is trained both on shaders with full computation and simplified shaders with partial computation (simp). For each experiment, we generate a 30 frames sequence and compute the error with respect to ground truth using the last frame. The reported numbers are averaged across 30 different sequences.*

Shader	RGBx	Ours
`Mandelbrot`	0.0104 / 0.980 / 37.16	**0.0037 / 0.989 / 39.75**
`Mandelbrot` simp	0.1049 / 0.898 / 27.51	**0.0693 / 0.929 / 28.93**
`Mandel-bulb`	0.0213 / 0.959 / 32.06	**0.0140 / 0.971 / 33.56**
`Mandel-bulb` simp	0.1194 / 0.780 / 23.49	**0.1035 / 0.788 / 24.25**
`Trippy Heart`	0.0665 / 0.864 / 26.61	**0.0546 / 0.884 / 27.17**
`Trippy Heart` simp	0.2295 / 0.563 / 19.10	**0.1788 / 0.637 / 21.10**

the RGBx baseline, and produces a more temporally coherent sequence than their non-temporal counterparts (Sections 5.1 & 5.2) while retaining similar visual quality in still images. We additionally verify that the temporal models generate more temporally stable sequences by computing the perceptual loss of 2 adjacent frames. For each of the 30 test sequences, we use the last two frames of the length 30 sequence and average the score across ten renders with different random seeds. We then average the score across the test dataset and compare between our temporal and our non-temporal models. In all cases, the temporal model has a lower error between adjacent frames. The temporal models have 94% perceptual error relative to the non-temporal models on average and 80% in the best case. Our supplementary video does not present temporally coherent animation as a separate application, but rather shows this training scheme in the denoising and simplification applications. Figure 9 shows an example where our method generalizes better to longer sequences than the RGBx baseline. Our result correctly learns both temporal coherence as well as the complicated structure in each individual frame, whereas the RGBx baseline introduces additional color artifacts in the output. The video shows even longer sequences (180 frames).


(a) RGBx Baseline (b) Our result

`Trippy Heart`: 100% 77%

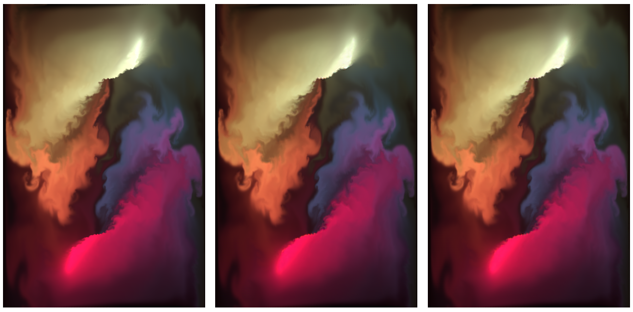
Figure 9: *Learning temporally coherent sequences for* `Trippy Heart` *with the same ground truth as in Figure 5. We report relative perceptual error compared to the RGBx baseline. Both RGBx (a) and ours (b) are the 90th frame of a synthesized temporally coherent sequence. Note how our method generalizes well to long sequences whereas the RGBx baseline presents obvious artifacts such as color residual from previous frames near the silhouette of the heart.*

Appendix G: Branching and Loop Emulation

As discussed in Section 3.1, our compiler currently handles conditional execution by simply evaluating both branches and unrolls loops to its maximum possible iteration. Variable-length loops are handled using a user-given compile-time pragma specifying a ceiling on the possible number of loop iterations: it is common to have such ceilings on iteration counts in shader programs because of the need to maintain consistent shading performance. Values from unused iterations are replaced with the values from the final computed iteration. We made these choices because they are much easier to implement in TensorFlow. However, in a practical application, shaders would typically be compiled to code that takes either branch or exits the loop early based on a termination condition. Therefore, we did an experiment to determine what would be have been the effect of handling branches and loops the traditional way. For branching, we simply wrote dummy values of zero to traces in the branch not taken. We applied such branch emulation to a shader called `Texture Maps` which—similar to aspects of `Venice` in Figure 1—uses a conditional statement to select a texture based on whether a ray has hit a plane. For loops, we wrote zero values to traces after the loop termination condition is met, and applied the emulation to `Mandelbrot`. In both cases we found that the emulation gives results that are visually and quantitatively identical to our compiler's implementation.

Appendix H: Comparison with Positional Encoding

Positional encoding [MST*20] can be viewed as a general method to augment input to learning that is agnostic to the input data's generation process. It applies high-frequency functions to positional features such as 3D coordinates. Because many shaders involve


(a) Reference (b) I/O Baseline (c) Our Method

0% 100% 96%

Figure 10: *An example of fluid simulation where our method (c) gives a very similar result as the I/O baseline (b). This indicates our method may not be advantageous for simple learning tasks where the baseline is already good enough to reconstruct the reference (a). The relative perceptual error compared to the I/O baseline is reported below each image.*

Table 3: *Error statistics for denoising (Section 5.1). We report LPIPS perceptual error [ZIE*18], SSIM and PSNR. The numbers reported are averaged across the entire test dataset. The error metric in the first two columns (RGBx and ours) are already reported in the main paper Table 1, we include them here for a clear comparison with Supersampling. For LPIPS, we report the absolute error for the RGBx baseline, and for other methods their errors relative to that of the RGBx baseline (% or ×).*

Shader	RGBx	Ours	Supersampling
`Bricks`	0.0141 / 0.981 / 36.68	**0.0097(68%) / 0.987 / 38.29**	0.2985(21×) / 0.839 / 24.14
`Gear`	0.0173 / 0.986 / 38.90	**0.0127(73%) / 0.988 / 39.86**	0.2935(17×) / 0.880 / 24.71
`Mandelbrot`	0.0235 / 0.973 / 36.07	**0.0059(24%) / 0.986 / 38.55**	0.3502(15×) / 0.746 / 23.46
`Mandel-bulb`	0.0185 / 0.962 / 32.14	**0.0118(63%) / 0.975 / 34.18**	0.1580(8.5×) / 0.895 / 23.98
`Oceanic`	0.0403 / 0.961 / 33.69	**0.0339(84%) / 0.966 / 34.51**	0.4314(11×) / 0.780 / 23.81
`Trippy Heart`	0.0696 / 0.856 / 26.30	**0.0543(77%) / 0.886 / 27.27**	0.2178(3.1×) / 0.767 / 22.42
`Venice`	0.0309 / 0.965 / 32.76	**0.0242(78%) / 0.973 / 33.83**	0.2893(9.4×) / 0.853 / 23.73

computing intermediate values that vary spatially in ways that cannot easily be captured via positional encoding, and some of them will be important to the learner, we believe our method offers an improvement over positional encoding for most shaders and most applications. To evaluate this hypothesis, we tested two applications (denoising in Section 5.1 and simplification in Section 5.2) × two shaders (`Mandelbrot` and `Trippy Heart`), adding explicit positional encoding features as described by [MST*20] to both the RGBx baseline and our method. On average across the four cases, we found that the addition of positional encoding features did not measurably change PSNR values. Moreover, the addition of positional encoding features increased the perceptual error of both RGBx and ours by 4% on average. Therefore, we verified our hypothesis that in the context of our applications and shaders, learning does not benefit from positional encoding.

Appendix I: Fluid Simulation

Although our method is beneficial in all the previously described experiments, we also find a null result for our second simulation example: a 2D fluid simulation. The state of the simulation on a 2D grid can be viewed as a 7D feature: 3D for RGB color of the fluid and 4D for internal states: velocity, density and vorticity. The simulation takes input of the 7 channel fluid state, solves the Navier-Stokes equation with a hard-coded external force to compute the new internal state, then applies color advection on image space to output the new 7D state. The color advection step controls the trade-off between how fast the fluid propagates and how accurate the simulation is. We ran the simulation with step size δ as ground truth. The learning task is to run the simulation at a coarser step 10δ, and predict the intermediate states in between the 10 steps as if they were run at the fine scale simulation with step size δ.

We use the same architecture as in Section 5.1 for this task and compare our method with an I/O baseline that takes the initial and output fluid states as learning features. While our method is marginally numerically better than the baseline (ours has 92% L2 error and 96% perceptual error compared to the baseline), the visual quality of the two methods is almost identical. We hypothesize that this learning task is not suitable for our method because it is relatively simple and lacks complicated hidden state: the neural network can easily approximate solving the Navier-Stocks equation given initial and output states. Additionally, because the fluid states change slowly even after 10 simulation steps, the network can easily hallucinate a reasonable correction using the initial state as a good starting point, therefore, the baseline features already suffice. In Figure 10 we show both the baseline and our method can reasonably approximate the reference with almost identical results.

Appendix J: Statistical Evidence for Subsampling Strategies

In this section, we provide statistical evidence for our findings when investigating trade-offs between different subsampling strategies and subsampling budgets described in Section 6.2. Our first null hypothesis makes the following assumption on the performance between Uniform and Oracle subsampling: the ratio of relative error between Uniform and Oracle (μ_0) is less than or equal to 1. This hypothesis has a p-value of $p_0 = 7.2 \times 10^{-4}$. Similarly, we propose another null hypothesis regarding the performance between Opponent and Uniform subsampling: the ratio of relative error between Opponent and Uniform (μ_1) is smaller than or equal to 1, which has a p-value $p_1 = 5.9 \times 10^{-3}$. If we choose a significance level of 0.05 and apply Bonferroni correction over the 2 hypotheses, we have both $p_0 < 0.025$ and $p_1 < 0.025$, indicating significant evidence that Oracle outperforms Uniform ($\mu_0 > 1$) and Uniform outperforms Opponent ($\mu_1 > 1$). These statistics are computed using all possible $\mathcal{N}$ available for all 4 shaders: $\mathcal{N} \in [\mathcal{T}/2, \mathcal{T}/4, \mathcal{T}/8, \mathcal{T}/16]$.