Genetic Programming for Shader Simplification

Pitchaya Sitthi-amorn Nicholas Modly Westley Weimer Jason Lawrence
University of Virginia

Abstract

We present a framework based on Genetic Programming (GP) for automatically simplifying procedural shaders. Our approach computes a series of increasingly simplified shaders that expose the inherent trade-off between speed and accuracy. Compared to existing automatic methods for pixel shader simplification [Olano et al. 2003; Pellacini 2005], our approach considers a wider space of code transformations and produces faster and more faithful results. We further demonstrate how our cost function can be rapidly evaluated using graphics hardware, which allows tens of thousands of shader variants to be considered during the optimization process. Our approach is also applicable to multi-pass shaders and perceptual-based error metrics.

Keywords: procedural texturing, pixel shaders, code simplification, genetic programming

Links: DL PDF WEB VIDEO

1 Introduction

The complexity of procedural shaders [Cook 1984; Perlin 1985] has continued to grow alongside the steady increase in performance and programmability of graphics hardware. Modern interactive rendering systems often contain hundreds of pixel shaders, each of which may perform thousands of arithmetic operations and texture fetches to generate a single frame.

Although this rise in complexity has brought considerable improvements to the realism of interactive 3D content, there is a growing need for automated tools to *optimize* procedural shaders to meet a computational budget or set of hardware constraints. For example, the popular virtual world Second Life allows users to supply custom models and textures, but not custom shaders: the performance of a potentially-complex custom shader cannot be guaranteed on all client hardware. Automatic optimization algorithms could adapt such shaders to multiple platform capabilities while retaining the intent of the original.

As with traditional computer programs, pixel shaders can be executed faster through a variety of semantics-preserving transformations like dead code elimination or constant folding [Muchnick 1997]. Unlike traditional programs, however, shaders also admit *lossy* optimizations [Olano et al. 2003]. A user will likely tolerate a shader that is incorrect in a minority of cases or that deviates from its ideal value by a small percentage in exchange for a significant performance boost.

One common way to achieve this type of optimization is through code simplification. The methods proposed by Olano et al. [2003] and Pellacini [2005] automatically generate a sequence of progressively simplified versions of an input pixel shader. These may be used in place of the original to improve performance at an acceptable reduction in detail. However, these methods have a number of important disadvantages. The system proposed by Olano et al. [2003] only considers code transformations that replace a texture with its average color. This overlooks many possible opportunities involving source-level modifications. The system proposed by Pellacini [2005] does consider source-level simplifications. However, that approach is limited to a small number of code transformations and uses a brute-force optimization strategy that can easily miss profitable areas of the objective function. Furthermore, both approaches were demonstrated only on shaders that require a single rendering pass. Modern shaders often involve multiple interdependent passes. Finally, in both of these systems, only a single error metric is used to evaluate the fidelity of a modified shader. It would be desirable for a simplification algorithm to support a range of error metrics, including those that are designed to predict perceptual differences [Wang et al. 2004].

Intuitively, we hypothesize that a shader contains the seeds of its own optimization: relevant functions such as `sin` and `cos`, operations such as `*` or `+`, or constants such as `1.0` are already present in the shader source code. We propose to produce optimized shader variants by copying, reordering and deleting the statements and expressions already available. We also hypothesize that the landscape of possible shader variants is sufficiently complex that a simple hill-climbing search will not suffice to avoid being trapped in local optima. We thus present a novel framework for simplifying shaders that is based on Genetic Programming (GP). GP is a computational method inspired by biological evolution which evolves computer programs tailored to a particular task [Koza 1992]. GP maintains a population of program variants, each of which is evaluated for suitability using a task-specific fitness function. High-fitness variants are selected for continued evolution. Computational analogs of biological crossover and mutation help to locate global optima by combining partial solutions and variations of the high-fitness programs; the process repeats until a fit program is located.

Our approach is related to recent software engineering work that applies GP to automatic program repair [Weimer et al. 2009, 2010]. Our approach is novel not only in the domain considered (continuous shader output vs. discrete software test cases) but also in the techniques used: to take advantage of the special structure of shader software, we apply new mutation operations, new approaches to select a diverse population of variants, new handling for multiple competing objectives, and optimizations to rapidly approximate the performance of a shader variant.

Our approach offers a number of benefits over existing methods for shader simplification. In particular, it explores optimizations beyond just texture lookups (cf. Olano et al. [2003]), is not limited to an *a priori* set of simplifying transformations (cf. Pellacini [2005]), does not require the user to specify continuous domains for shader input parameters (cf. Pellacini [2005]), is demonstrably applicable to multi-pass shaders and perceptual-based error metrics, and outperforms previous work in a direct comparison.

2 Related Work

Prior work on shader optimization has generally taken the form of either pattern-matching code simplification or data reuse. Previous work on GP-based program repair has focused on fixing bugs in off-the-shelf, legacy software, and not on optimizing programs.

Code Simplification: The original system for simplifying procedural shaders was developed by Olano et al. [2003]. It focused on converting texture fetches into less expensive operations. A more recent technique, and one more similar to our own, was proposed by Pellacini [2005]. Pellacini's algorithm generates a sequence of simplified shaders automatically, based on the error analysis of a fixed set of three simplification rules (e.g., replacing an expression with its average value over the input domain). It only uses error estimation to guide its search [Pellacini 2005, Sec 3.2]. By contrast, our approach considers arbitrary statement- and expression-level modifications, and uses both error and rendering time to guide a multi-objective optimization. In Section 4, we directly compare our algorithm to Pellacini's.

Data Reprojection: An alternative strategy for optimizing a procedural shader is to reuse expensive calculations over consecutive frames [Nehab et al. 2007]. In many cases, this can reduce rendering time at an acceptable reduction in quality. Temporal reprojection was used by Scherzer et al. [2007] to improve shadow generation and by Hasselgren and Akenine-Moller [2006] to accelerate rendering in multi-view architectures. Sitthi-amorn et al. [2008] demonstrated a related system for automatically identifying the subexpressions in a pixel shader that are most suitable for reuse.

Code simplification in general has a number of advantages over data reuse. First, it does not incur additional memory requirements in the form of off-screen buffers to support a cache. Second, it can be used with semi-transparent shaders, which are not supported by existing data reprojection methods. Finally, a series of progressively simplified shaders can be accessed at run-time to achieve appropriate level of detail while rendering [Olano et al. 2003].

GP-based Code Repair: Weimer et al. [2009, 2010] use genetic programming to automatically repair bugs in unannotated software. Their technique optimizes only a single discrete value: the number of passed test cases. In contrast, our approach makes no use of negative test cases to limit the search and must simultaneously optimize real-valued error and performance objectives, for which we introduce a rapid approximation. We also use new mutation operators and new approaches to select a diverse set of variants that are specific to the domain of shader software.

3 Shader Simplification with GP

Our system for shader simplification is inspired by the program repair techniques of Weimer et al. [2009, 2010] and NSGA-II multi-objective optimization [Deb et al. 2002], but incorporates shader-specific insights. The user provides as input the original shader source code and an indicative rendering sequence (e.g., samples of game play). An iterative genetic algorithm then maintains and evaluates a diverse population of shader variants, returning those representing the best optimization tradeoffs.

We represent each shader variant as its abstract syntax tree (AST). For example, "`3*x`" is represented as a *BinOp* expression node with three children: `3`, `*`, and `x`. Similarly, "`x=y`" is represented as a *Set* statement instruction node with two children: `x` and `y`. See Necula et al. [2002] for a formal description of the AST nodes used. The steps of our algorithm are as follows:

Step 1: Population Initialization. Each element of the population is initialized by mutating the original shader. A variant is mutated by considering each of its expressions (i.e., AST nodes) in turn. We may delete that expression, insert it as the child of a randomly-chosen expression, or swap it with a random expression. Unlike work in previous software repair, we may also replace the expression by its estimated average value. Each of these four outcomes is equally likely. Deletion is only applicable to statement-level nodes, and is implemented by replacing the statement with an empty block.

Step 2: Fitness Evaluation. The fitness of each variant is computed by transforming the AST back to shader source code that is then compiled and executed (see Sections 3.2 and 3.3). As an optimization, the fitness calculations can be memoized based on the shader source code or assembly code. We write $f(v) = \langle t, e\rangle$ for the fitness of a variant v given its rendering time t and error e.

Step 3: Non-Dominated Sort. Unlike previous test-centric work in software repair, since time and error have incomparable dimensions, we introduce a partial ordering and say that $\langle t_1, e_1\rangle$ *dominates* (i.e., is preferred to) $\langle t_2, e_2\rangle$ (written $\langle t_1, e_1\rangle \prec \langle t_2, e_2\rangle$) if either $t_1 < t_2 \wedge e_1 \leq e_2$ or $t_1 \leq t_2 \wedge e_1 < e_2$. That is, one variant dominates another if it improves in one dimension and is at least as good in the other. In a set of variants V, the *Pareto frontier* of V is $P(V) = \{v \in V : \forall v' \in V.\ v' \not\prec v\}$. That is, the variants in the Pareto frontier are preferred (but incomparable) and the variants not in the frontier are dominated. If $v \in P(V)$ we say $rank(v) = 1$. It is also possible to consider the variants that would be in the Pareto frontier if only the current frontier were removed: $P(V - P(V))$. We say such variants v have have $rank(v) = 2$, and so on, as in Deb et al. [2002].

Step 4: Diversity. We wish to retain low-$rank$ed variants into the next iteration. However, if there is limited space in the next iteration, we also prefer to break ties by retaining a diverse set of variants. In the particular domain of graphics shaders, variants with distinct source code often have equivalent fitness values; we thus cannot use the standard *crowding heuristics* of previous work [Deb et al. 2002]. In addition, crowding heuristics typically assume that the objectives have compatible units which is not true of time and error. Instead, we partition those variants into equivalence classes by fitness value, and select the next iteration of variants uniformly at random, without replacement, from those classes.

Step 5: Mating Selection. We choose the fittest variants using a process known as *tournament selection* [Miller and Goldberg 1996] with a tournament size of 16. This is done by selecting 16 individuals at random and returning the preferred variant, where v_1 is preferred to v_2 if $rank(v_1) < rank(v_2)$ (with ties broken by fitness diversity). We apply selection repeatedly to obtain $2 \times |V|$ (possibly repeated) individuals, treating them pairwise as *parents*. Previous work in program repair and most genetic algorithm approaches use a tournament size of two [Mitchell et al. 1993]; however, our experiments with different tournament sizes (2, 4, 8, 16, 32) found that tournament size of 16 gives converges faster than the alternative. Our larger tournament size means that more variants are considered when making a decision. This focuses our search less on *exploration* and more on *exploitation* of high-fitness substructures that have already been located. This is desirable because we start with a fully-functional shader, rather than a blank slate or random program. Compared to a classic optimization approach such as Newton's Method, we have reason to believe that our starting point is relatively close to our desired point, and thus we have less reason to jump wildly from it. In addition, because shader software is unlike discrete systems software, small changes to the source are likely to result in small changes to the render time and output fidelity. Finally, this high tournament size implies a high *selection pressure*, which would normally cause a loss of diversity, but is balanced by Step 4, which favors greater diversity during the selection function.

Step 6: Crossover and Mutation. GP uses crossover to combine partial solutions from high-fitness variants; along with mutation it helps GP to avoid being stuck in local optima. Each pair of parents produces two offspring via *one-point crossover*. Recall that each variant is represented by its AST. We impose a serial ordering on AST nodes by depth-first traversal, anchored at the statement level, producing a sequence of statement nodes for each variant. We select a crossover point along that sequence. A child is formed by copying the first part of one parent's sequence and the second part of the other's and then reforming the AST. For example, if the parents have sequences $[P_1, P_2, P_3, P_4]$ and $[Q_1, Q_2, Q_3, Q_4]$, with crossover point 2, the child variants are $[P_1, P_2, Q_3, Q_4]$ and $[Q_1, Q_2, P_3, P_4]$. Each child variant is then mutated once (see Step 1), and its fitness is computed (see Step 2).

Step 7: Selection for Next Iteration. The incoming population V and all of the produced and mutated children are considered together as a single set of size $3 \times |V|$. This set is sorted and its crowding distances are computed as in Steps 3 and 4. We then select $|V|$ preferred variants to form the incoming population for the next iteration. This is done by adding all variants with *rank* equal to 1, then all variants with *rank* equal to 2, and so on, until $|V|$ have been added. The final *rank* may exceed the number of remaining slots, at which point diversity is used to break ties (Step 4).

At the conclusion of the final iteration, all variants ever produced and evaluated are used to compute a single unified Pareto frontier. The output of the algorithm is the set of variants along that frontier: a sequence of progressively simplified shaders that trade off error for rendering time. Because it is taken from the Pareto frontier, the sequence is ordered by increasing error and decreasing rendering time simultaneously.

The heart of our algorithm — the fitness evaluation (see next section), diversity heuristic, high-population tournament selection, and mutation operators — are all novel in this design space compared to previous work [Deb et al. 2002; Weimer et al. 2009, 2010] and are specialized to the structure of shader software. For example, we have found empirically that using our higher-than-normal tournament selection size of 16 halves total optimization time while producing higher quality results. The genetic algorithm is parameterized by a population size, mutation rate, and desired number of iterations. For the experiments in this paper we set them to default values of 300, 0.015, and at most 100.

3.1 Example

It is perhaps remarkable that random reorderings of shader source code could produce desirable optimizations. In this section, we illustrate our approach on a simple example. Consider the following function, which computes the Fresnel reflectance of a dielectric with index of refraction `n` at incident angle `th`:

```
1   float Fresnel (float th, float n) {
2     float cosi = cos (th);
3     float R = 1.0f;
4     float n12 = 1.0f / n;
5     float sint = n12 * sqrt (1 - (cosi * cosi));
6     if (sint < 1.0f) {
7       float cost = sqrt (1.0 - (sint * sint));
8       float r_ortho = (cosi - n * cost)
9                     / (cosi + n * cost);
10      float r_par = (cost - n * cosi)
11                    / (cost + n * cosi);
12      R = (r_ortho * r_ortho + r_par * r_par) / 2;
13    }
14    return R;
15  }
```

For this example, the index of refraction `n` is set to 1.25 while `th` is allowed to vary over the range $[0°, 90°]$. The shader variants produced by our system are visualized in Figure 1(left). The first point on the Pareto frontier corresponds to the original shader. Note that

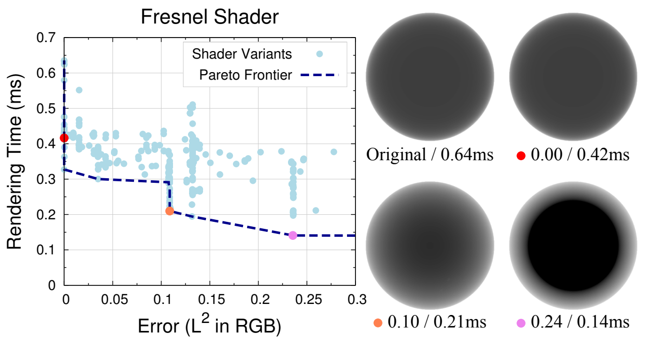

Figure 1: *Illustration of our simplification algorithm applied to a simple Fresnel shader.* ***Left:*** *Each shader variant produced during the optimization process is visualized by a light blue dot showing its error and performance; the dashed line marks the Pareto frontier. Error is measured as the average per-pixel L^2 norm in RGB space.* ***Right:*** *A few variants that lie along the frontier. Each image shows the shader applied to a sphere illuminated by a point light located at the camera. The captions report the error and render time.*

the check for `sint < 1.0f` always returns true since `n > 1.0`, and can therefore be removed. This corresponds to the point on the Pareto frontier that is marked with a red dot in Figure 1, and code below:

```
1   float Fresnel (float th, float n) {
2     float cosi = cos (th);
3     float R = 1.0f;
4     float n12 = 1.0f / n;
5     float sint = n12 * sqrt (1 - (cosi * cosi));
6     {
7       float cost = sqrt (1.0 - (sint * sint));
8       float r_ortho = (cosi - n * cost)
9                     / (cosi + n * cost);
10      float r_par = (cost - n * cosi)
11                    / (cost + n * cosi);
12      R = (r_ortho * r_ortho + r_par * r_par) / 2;
13    }
14    return R;
15  }
```

Our technique also considers changes in operands and expressions. For example, one of the variants produced by our technique swaps $\star$ and + operators on Line 11:

```
10      float r_par = (cost - n * cosi)
11                    / (cost * n + cosi);
```

Note that because `cost * n` has been previously computed in Lines 8–9, that value can be reused, and this variant produces faster and more compact shader assembly code (e.g., this reduces the shader Cg assembly from 80 to 78 instructions). Note that while the algorithm of Pellacini [2005] can produce the `if`-removing optimization, it cannot produce the operand-swapping one. Larger tradeoffs are possible, however: the variant marked with an orange dot in Figure 1 removes the `r_par` term entirely, setting it to `0.0f`. While these changes are simple in isolation, together they speed rendering time by 3x while incurring only a very slight error (0.10 L^2 RGB). The final highlighted variant, represented by the purple dot in Figure 1, significantly sacrifices visual fidelity for rendering speed:

```
8     float r_par = (cost - n * cosi)
9                   / (cost + n * cosi);
10    R = r_par / 2;
11    return (R);
```

In effect, it removes the `r_ortho` term entirely, as well as the squaring of `r_par`.

3.2 Error Model

The quality of any optimization system depends on the error metric used to measure the difference between the original shader and the variants. For comparison with previous systems, we use the average per-pixel L^2 distance in RGB by default, although our system allows for other error metrics (see Section 4.7).

For each shader variant, we compute the mean L^2 RGB error over the frames in a representative rendering sequence provided by the user. Only shaded pixels are considered; background pixels are not considered as part of the average. Because our system may generate tens of thousands of shader variants, the cost of directly evaluating the error for thousands of frames can be high. In the case of single-pass shaders, we address this problem by randomly sub-sampling a small fraction of patches from the representative sequence and computing the average error using only these patches. This can be done quickly using deferred shading [Deering et al. 1988], as described below. In the case of multiple-pass shaders, we use the entire representative sequence to evaluate the error.

In a preprocessing step, we populate a geometry buffer (G-buffer) with patches of input values taken from the representative sequence. Specifically, we use a 2048×2048 G-buffer composed of patches uniformly sampled from each frame in the representative sequence. To achieve some degree of stratification, we collect a fixed number of patches from each frame and discard those that overlap the background. We evaluate each shader variant over this G-buffer and compute the difference between this output and the original. This subsampling procedure reduces the error evaluation time by over an order of magnitude for the representative sequences we considered. Additionally, note that by sampling patches, we can support other error metrics such as the Structural Similarity Index (SSIM), which operate on groups of nearby pixels to measure local contrast (Section 4.7).

Figure 2(left) evaluates the correlation between the error values computed using the representative sequence ("brute-force") and our subsampling approach. The evaluation uses roughly 100 shader variants of the Marble shader (Section 4.2) generated by our simplification system. The cross-correlation is 0.84 which indicates a high degree of success in predicting the actual error from this small set of samples.

3.3 Performance Model

We also accelerate evaluating running time using a technique similar to the subsampling method for evaluating error. Note that using the same exact deferred shading approach described above to estimate rendering time would involve modifying the pixel shader code to fetch its inputs from the G-buffer, which may change its performance profile. We therefore use a grid-based method that allows a fast evaluation while still capturing the expected performance characteristics of a shader.

We draw a matrix of quadrilaterals that each occupy an 8×8 pixel window and together cover a $2,048 \times 2,048$ render target. Each vertex in each quad fetches the inputs stored in the G-buffer and then passes these to the pixel shader. This avoids modifying the pixel shader code and reduces the performance evaluation time from minutes to 1–2 seconds per shader.

Another important benefit of using coherent 8×8 pixel patches is that this avoids introducing differences in the performance profile due to dynamic flow control. Modern graphics hardware maximizes throughput by executing nearby pixel shaders in "lock-step". Assemble the G-buffer by sub-sampling single pixels from the representative sequence potentially introduces many discontinuities in the execution paths of neighboring pixels (i.e., the shaders at neighboring pixels may do very different things since their inputs were constructed artificially). By sampling patches, we maintain the same degree of spatial coherency found in the representative sequence and thus achieve a more accurate approximation of the expected rendering time. We determined experimentally that 8×8 pixel windows produce a G-buffer that matches the lock-step execution boundaries in modern NVidia and AMD hardware.

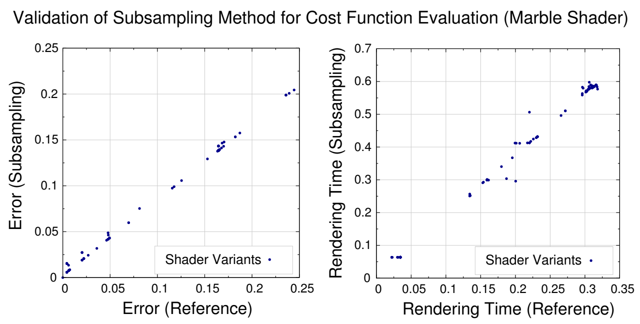

Figure 2: *Validation of our subsampling approach for computing the error and performance of each shader variant. Each point in these graphs represents a single variant of the Marble Shader produced by our GP optimization.* ***Left:*** *A comparison between the average L^2 RGB error computed using the entire representative sequence (x-axis) and the approximation by our subsampling approach (y-axis). The correlation coefficient is 0.84.* ***Right:*** *A comparison between the rendering time measured using the representative sequence (x-axis) and the rendering time approximated by our subsampling approach (y-axis). The correlation coefficient is 0.95.*

Figure 2(right) compares the estimated rendering time using our grid-based subsampling method to the rendering time measured for the entire sequence. Note that the absolute values are different, because the geometry processing and fixed setup costs differ between these approaches. However, the cross-correlation value is 0.95, indicating that our acceleration technique can reliably predict the relative differences between two variants, which is all that is required for our optimization strategy.

4 Experimental Results

In this section, we present results of our prototype shader optimization system. Our primary evaluation architecture is an AMD Phenom X6 equipped with an NVidia 285GTX. Our implementation accepts either HLSL (DX10) or OpenGL Cg source code. It can also operate on shader assembly, treating the assembly file as an AST block and each assembly instruction as an atomic AST node. We use the `gpu_time` hardware counters provided by the NVidia PerfSDK to measure rendering times when possible.

The shaders we used to evaluate our system are detailed in Table 1. The setup time for the sampling and deferred shading approximations is on the order of three minutes, which is dwarfed by the total optimization time. In general, there is no direct relationship between the number of variants considered (e.g., every light blue dot in Figure 1) and the number of dominating variants on the Pareto frontier (e.g., the points on the dashed blue line in Figure 1): the former represent the search space considered by the optimization and the latter the best discovered answers.

Table 2 shows the distributions of the different mutation operators that were used during our GP search to obtain the set of results along the Pareto frontier. Notably, these figures show that replacing one expression with another is the most useful operation in this context, followed by inserting expressions and crossing over to share source code with other variants. Removals were less likely to be involved in the construction of a shader output by our system, which supports the claim that genetic evolution is necessary and that our method is not simply removing or pruning shader code.

	Lines of Code		Time (hours)			Shader Variants			Speedup @ ≤ 0.075 L^2 error	
Shader	Source	Asm	Compiling	Testing	Total	Generated	Unique	On Frontier	Pellacini	Our Approach
Fresnel	28	40	0.7	0.3	1	14,218	743	94	1.67x	2.11x
Marble	120	1,023	0.2	0.7	1	7,417	1,725	294	1.43x	4.44x
Trashcan	127	363	2.0	3	5	4,862	1,461	119	1.24x	3.60x
Human Head	962	522	1.0	10	11	18,682	2,960	88	n/a	2.13x
S.S. Amb. Occl.	612	4,266	4.8	1.5	7	1,289	676	54	n/a	4.71x

Table 1: *Shaders used in our experiments. "Lines of code" counts non-comment, non-blank lines. "Compiling time" is the total time taken by the shader compiler over all variants. "Testing time" reports the time spent evaluating the error and performance of variants (Section 3). "Total time" includes the total time spent performing GP bookkeeping, compiling time and testing time. "Generated Variants" counts all distinct shader source codes produced; "Unique Variants" counts the number of unique assembly produced. "Variants on Frontier" gives the number of shaders on the Pareto frontier: the size of the final set of error-performance tradeoffs produced by our algorithm. The "Speedup @ ≤ 0.075 L^2 error" columns give the best speedup (original rendering time divided by optimized rendering time) of a variant with L^2 RGB error at most* 0.075 *for both our approach and that of Pellacini [2005].*

In what follows, we provide a more detailed analysis of these results and then compare to prior work. The supplemental material includes results for one additional example: a multiple-pass variance shadow map shader.

4.1 Fresnel Shader

The Fresnel shader was discussed in Section 3.1. While it is less complicated than our other examples, it is indicative of the inner loops of many shaders. Figure 1 shows the Pareto frontier computed by our algorithm and variants along this frontier rendered as a sphere under simple lighting. We used this simple scene to evaluate the performance and error values of the variants during the optimization of this shader.

Table 1 shows measurements related to the optimization process. Our approach produced a total of 94 distinct variants representing non-dominated performance-error tradeoffs. Each such variant corresponds to a point on the Pareto frontier in Figure 1: the right side of the Figure highlights three interesting variants. Finding the 94 final variants involved the generation of 14,218 unique shader source programs, which compiled to 743 unique shader assembly programs (i.e., programs with different source text may produce the same shader assembly when passed through the NVidia optimizing `cgc` shader compiler). The unique shader assembly programs are marked by light blue dots in the upper right of Figure 1; they represent the search space explored by our optimization technique. The entire process took one hour, with 70% of the time spent compiling shader programs and 30% of the time spent evaluating them (see Sections 3.2 and 3.3). Bookkeeping costs related to the genetic algorithm (e.g., performing the non-dominated sort, see Section 3) were dwarfed by the costs of compiling and evaluating the shaders.

4.2 Marble Shader

The Marble shader combines four octaves of a standard procedural 3D noise function [Perlin 1985] with a Blinn-Phong specular layer [Blinn 1977]. Figure 3 shows the error and rendering time of each shader variant produced. The representative rendering sequence consisted of 256 frames showing the model at many different orientations and illuminated from many different directions. We have included a video of the representative sequences for all of our test scenes as supplemental material. Our system successfully produces shader variants that reduce the rendering time, while introducing only a small amount of error. Variants with an L^2 error exceeding 0.2 were not considered. The average render time of the original shader is 0.31ms.

The first highlighted shader, marked with a red dot in Figure 3, removes the highest frequency bands in the noise calculation. This results in a shader that renders each frame in 70% of the time required by the original while introducing only 0.015 average per-pixel L^2 RGB error. The small inset images show the per-pixel L^2 error linearly scaled by 100 and mapped to a standard colormap (i.e., fully saturated red corresponds to an error value of 0.01). The same colormap is used throughout the paper.

Shader	Add	Replace	Swap	Remove	Cross Over
Marble	235	477	68	194	941
Trashcan	202	331	100	111	393
Human Head	3,691	4,269	1,961	1,546	2,933
S.S. Amb. Occl.	333	612	186	239	393

Table 2: *Distributions of mutation operators use to obtain the set of results along the Pareto frontier for each non-trivial test shader.*

The next highlighted shader, marked with an orange dot, removes two of the highest frequency bands in the noise calculation, reducing rendering time to 45% of the original. The final highlighted shader, marked with a purple dot, represents an aggressive optimization that removes the three highest frequency bands as well as the specular highlight, which contributes to only a small region of the scene. The rendering time of this last shader is 23% of the original. In total, our technique produced 96 shaders representing distinct performance-error tradeoffs (see Table 1) in one hour.

4.3 Trashcan Shader

The Trashcan shader is from ATI's "Toyshop" demo.[1] This shader reconstructs 25 samples of an environment map and combines them with weights given by a Gaussian kernel. These samples are evaluated along a 5×5 grid of normal directions generated from a normal map. The representative rendering sequence for this shader consisted of 340 frames that show the trashcan being rotated and enlarged.

Figure 4 illustrates the optimized shaders found by our system. At the red point, our system removes 4 of the 25 environment map samples from the pixel value and correctly changes the total weight by which the remainder are combined, producing an output that is almost indistinguishable from the original and takes 89% of the time to render as the original shader.

The shader variant indicated by the orange dot computes 12 environment map samples, reuses some of the computation, and adjusts the normalization values. This reduces the rendering time to 75% of the original. The resulting error can be seen most clearly along the trashcan's lid and body (see inset Figure 4). The final example, corresponding to the purple dot, uses only the highest-weighted of the original 25 environment map samples. This gives a significant performance improvement (rendering time reduced to 30% of the original) at the cost of some aliasing artifacts. The entire optimization process took five hours to produce the 120 shaders on the

[1]`http://developer.amd.com`

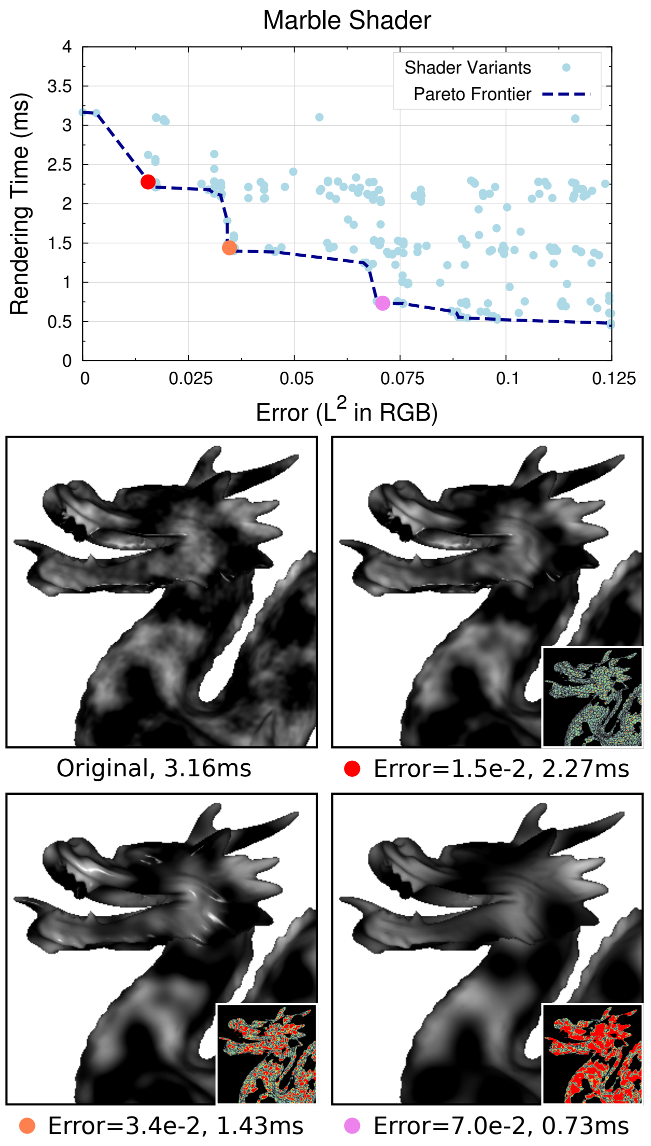

Figure 3: Top: *Scatter plot of error and performance of different Marble shader variants generated by our system. The line marks the Pareto frontier.* **Bottom:** *Selected rendering results as indicated in the graph above. The inset contains a visualization of the per-pixel error.*

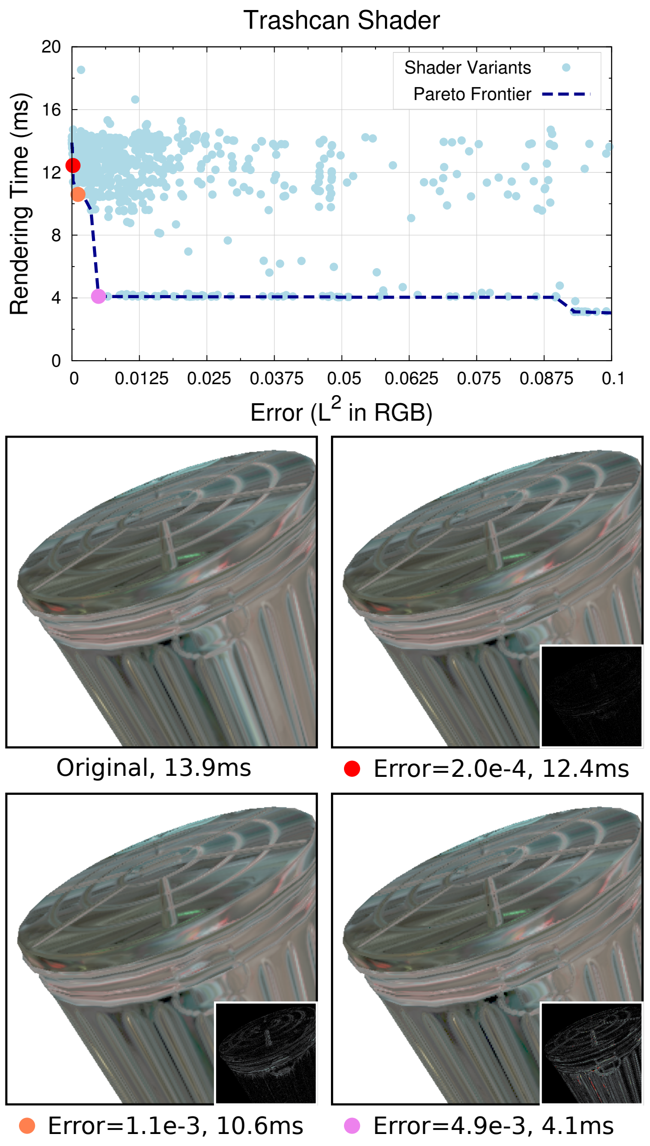

Figure 4: Top: *Scatter plot of error and performance of different Trashcan shader variants generated by our system. The line marks the Pareto frontier.* **Bottom:** *Selected rendering results as indicated in the graph above. The inset constrains a visualization of the per-pixel error.*

Pareto frontier.

4.4 Human Head Shader (Multiple Pass)

The NVidia Human Head demo was developed by d'Eon et al. [2007]. It uses texture-space diffusion to simulate subsurface scattering in human skin in multiple rendering passes. In the first pass, the surface irradiance over the model due to a small number of point light sources is computed. In subsequent rendering passes, this irradiance function is repeatedly blurred using a cascading set of Gaussian filters to simulate subsurface scattering at different wavelengths. Each Gaussian filter uses eight samples. In a final rendering pass, this estimate of subsurface scattering is combined with a surface reflection layer (Cook-Torrance BRDF [Cook and Torrance 1981]), an environment map, and standard tone mapping and bloom filters. Altogether, this shader involves fifteen passes.

Figure 5 shows the set of shaders our system discovered. At the first highlighted shader, indicated by the red dot, our system reduces the calculation involving the environment map and bump maps. This produces almost-equivalent output (L^2 RGB error of 0.001) but reduces the rendering time to 91% of the original.

More aggressive tradeoffs are possible. The shader at the orange dot simplifies the specular highlight produced by one of the light sources. In addition, it simplifies the Gaussian filter from 8 to 6 samples along the x-axis and from 8 to 7 along the y-axis. This variant reduces render time to 76% of the original. The most noticeable differences are in the specular highlight on the nose and the overall increasing in surface roughness.

The final example, marked by a purple dot, removes the Gaussian blur filter entirely, and removes several calculations in the final gather step associated with those passes. Without the repeated blurring of the irradiance function, the simulation of subsurface scattering is much less faithful, and the shadows on the face are clearly sharper. This variant renders in 46% of the time of the original.

The human head shader is significantly more complicated than the previous single-pass shaders (see Table 1). In addition to being

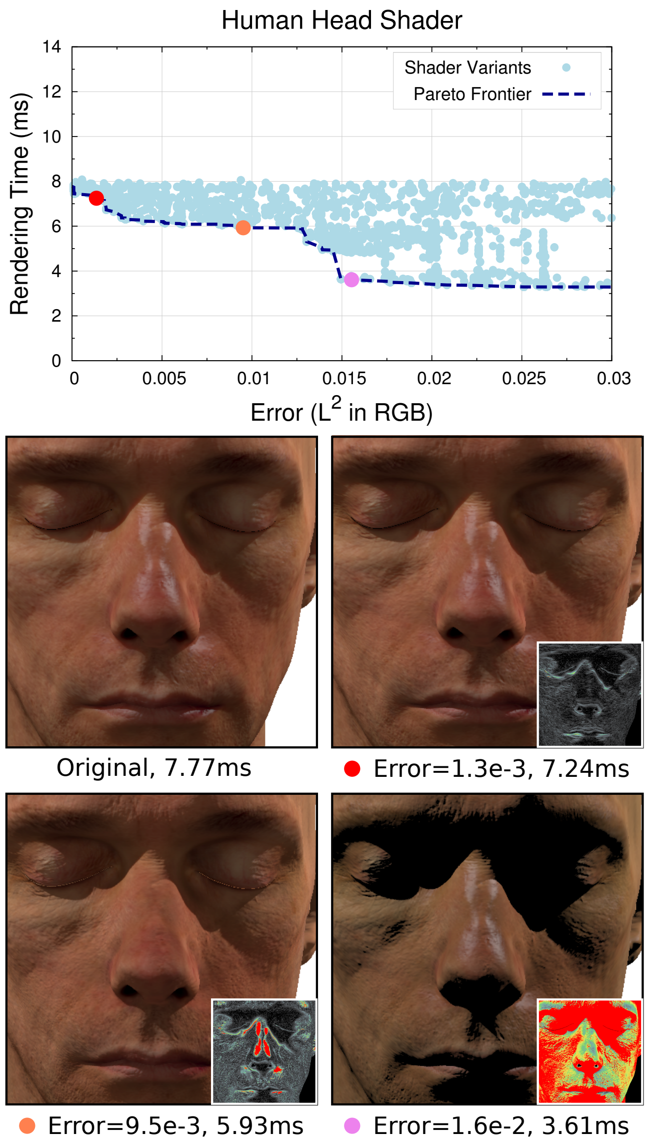

Figure 5: **Top:** *Scatter plot of error and performance of different shader variants generated by our system on the Human Head shader.* **Bottom:** *Selected rendering results as indicated in the graph above. The inset contains a visualization of the per-pixel error.*

eight times larger, it is also significantly more expensive to evaluate, as our subsampling optimization applies only to single-pass shaders (see Sections 3.2 and 3.3). Over 90% of the optimization time was thus spent evaluating error and performance.

In addition, this shader's complexity creates a much larger search space of possible optimizations. An order of magnitude more unique shader variants were considered than for the single-pass shaders. This can be seen in the dense covering of dots in Figure 5: while the smaller shaders are sparse and have a discontinuous fitness landscape, the complexity of the human head shader means that more feasible error-performance tradeoffs can be investigated. The high coverage demonstrates that our technique is strong enough to produce shaders with fine error distinctions. If we merely deleted statements or replaced variables with their averages, the distribution of discovered variants would display far more discontinuous jumps. In total, 88 optimized shaders were produced in 11 hours.

4.5 SSAO (Multiple Pass)

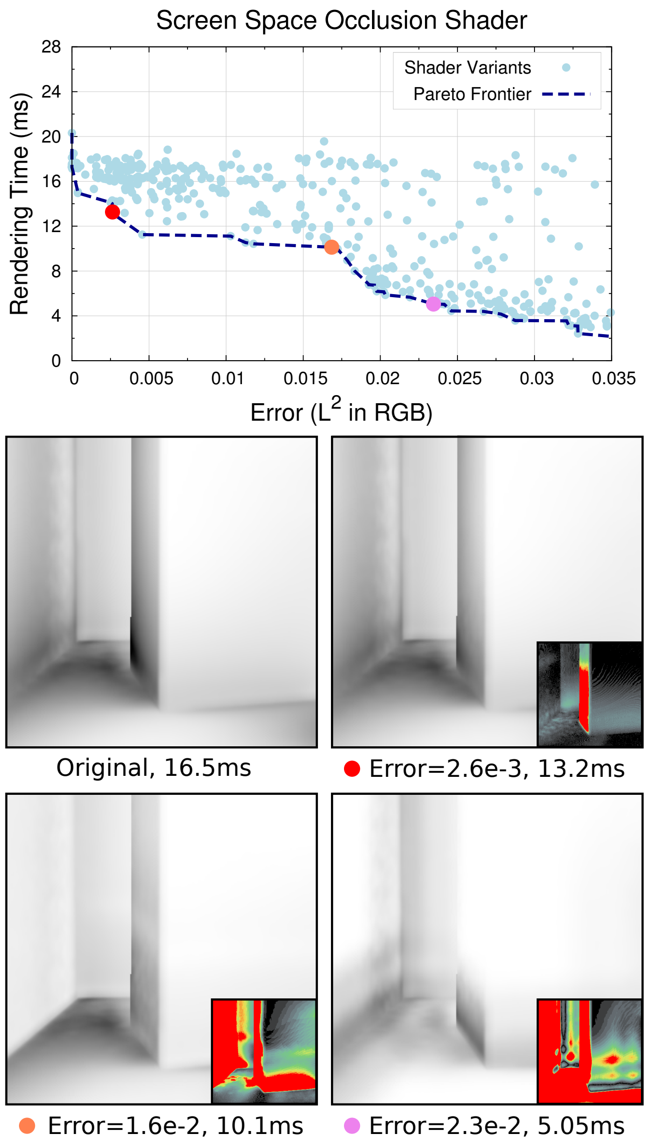

Figure 6: **Top:** *Scatter plot of error and performance of different shader variants generated by our system on the Screen Space Ambient Occlusion shader.* **Bottom:** *Selected rendering results as indicated in the graph above. The inset contains a visualization of the per-pixel error.*

The Screen Space Ambient Occlusion (SSAO) shader is taken from the NVidia DirectX SDK 10 demo suite and implements the horizon-based algorithm of Bavoil et al. [2008]. This approximates the portion of the hemisphere that is occluded by scene geometry within a fixed radius by summing together an estimate of the visibility along 16 directions. Eight discrete positions along each of the 16 directions are checked to detect occluded lines of sight.

Figure 6 visualizes the optimized shaders located by our system. At the red point, our system has reduced the number of tracing steps per direction by one and reduced the number of directions considered at each pixel from 16 to 15. This reduces the rendering time to 80% of the original.

The shader variant indicated by the orange dot reduces the number of positions along each ray from 8 to 6. It also simplifies the code that produces the sampling directions. This reduces the rendering time to 60% of the original but increases the overall brightness of the scene. The most noticeable differences are near the corner of the box. The final variant at the purple dot represents a more significant reduction in the number of directions that are considered. This reduces the rendering time to 31% of the original, but introduces clear visual differences. Our system took seven hours to produce the 54 shaders on the Pareto frontier.

4.6 Comparison to Prior Work

The technique most similar to our own was developed by Pellacini [2005]. Pellacini's algorithm generates a set of shaders according the three simplification rules. The first rule removes constants from binary operations (e.g., $e \times 3 \mapsto e$); the second rule simplifies loops by removing iterations; the third rule replaces an expression by its average value. The average values of expressions are computed by simulating the shader $1,000$ times on random points in its input domain as specified by the user. Each simplification rule is applied at all relevant locations; the application of a single rule to a single location produces a new candidate variant. The error of each candidate variant is calculated by evaluating it on one sample of the input domain. Each variant is normalized so that its average output equals the average output of the original. The variant with the lowest error is selected for the next iteration (note that rendering time is not considered and thus is not guaranteed to decrease) and the process repeats.

Figure 7 compares our system to that of Pellacini on the Trashcan shader. Only the Pareto frontier of the final shader sequences are shown (our technique in solid blue, unchanged from earlier; Pellacini's technique in dashed green). To compare the two techniques in context, we consider a scenario involving a fixed rendering time budget per frame. The two red points answer the question: "What is the best visual fidelity that can be achieved using at most 11ms?" In that setting, the best available shader produced by our technique has 20 times less error than the best available shader produced by Pellacini's technique. The orange point shows the best shaders available with a rendering time of at most 5ms. Here our technique yields 35 times less error: the Pellacini shader is almost a single solid color.

This trend continues for most of the fitness space: when the blue line is below and to the left of the green dashed line, our technique produces better error at fixed time and also better rendering times at fixed error. At even higher levels of error (above 0.09), Pellacini's technique is more efficient than ours (it reduces to returning the average RGB color of the original shader; the lines cross in Figure 7). However, since both shaders are effectively returning solid colors at that point, the space is uninteresting and likely represents an unacceptable level of error. Before that point, all variants produced by our technique dominate those produced by previous work, typically with an order-of-magnitude less error for the same time budget.

Figure 8 repeats this comparison on the Marble shader. Once again, our technique produces faster, more faithful shaders for the majority of the fitness space. When the rendering budget is fixed under 2.5ms (the red dots), our technique introduces less than half as much error (note the shading near the neck, just below the jaw, on Pellacini shader). When the rendering budget is doubled to 1.5ms, our approach produces three times less error (orange dots; note blocky visual artifacts on Pellacini shader).

A large part of this performance disparity results from Pellacini's technique using the error metric alone for guidance (i.e., it is not a classical multi-objective search): if one variant increases the error by ϵ but does not reduce the rendering time and another increases the error by 2ϵ but reduces rendering time to 20% of the original, that technique will select the former. Unfortunately, the former may rule out the latter: because only a single edit is considered at a time, an attractive local step may preclude a richer optimization later. Consider the original Fresnel source code shown in Section 3.1: replacing the `r_ortho = (cos - n * cost) / ...` calculation

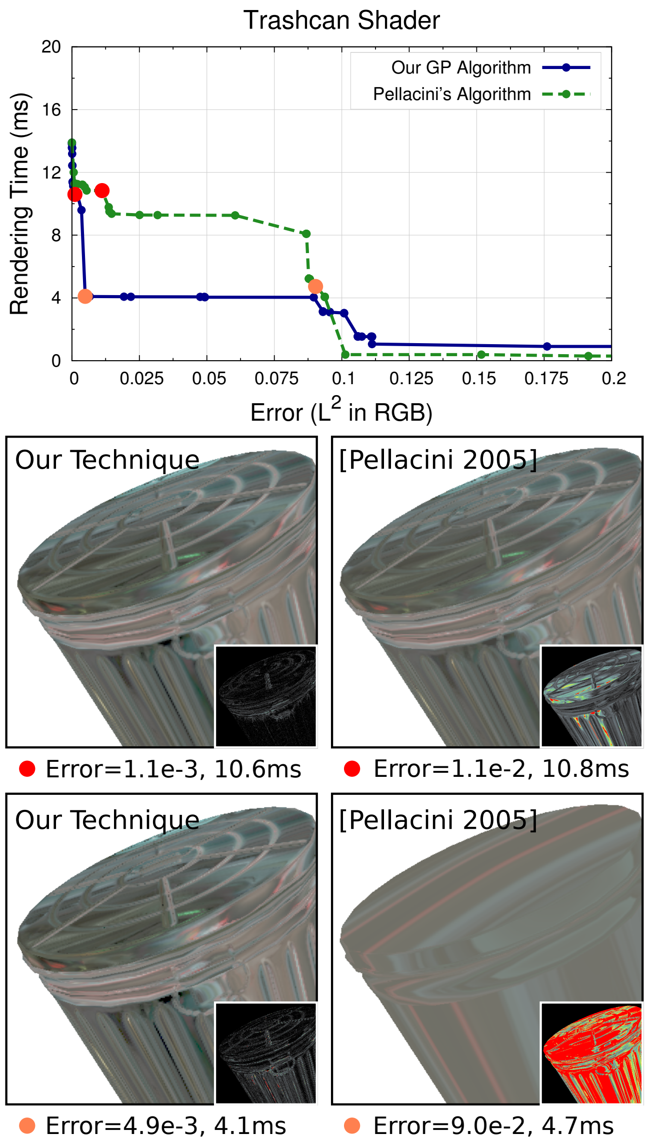

Figure 7: **Top:** *Comparison between our system and Pellacini's.* **Bottom:** *Side-by-side comparisons of the best shaders each method produces for a fixed rendering budget of 11ms (red dots) and 5ms (orange dots).*

on Lines 8–9 with its average value may be locally effective, but it rules out the `r_par` rewriting optimization our algorithm finds because `n * cost` is no longer a subexpression available for reuse by the compiler. Thus the technique may converge to poor local minima.

By contrast, our multi-objective approach explicitly minimizes both error and performance, which in practice results in a faster rendering time given the same amount of error. The GP approach we use keeps many shader instances in each generation; this high population combined with crossover and mutation explicitly helps to avoid being stuck in local optima (see Step 6, Section 3). In addition, our stronger mutation operator includes arbitrary sub-expression and statement insertion and deletion, allowing the shader to contribute to its own optimization, rather than relying on a small number of fixed simplifications.

Another important difference between our method and Pellacini's is that we can gracefully handle shaders that require multiple passes (e.g., Human Head, SSAO, and the shadow map shader included in supplemental material). Pellacini's requires average values of

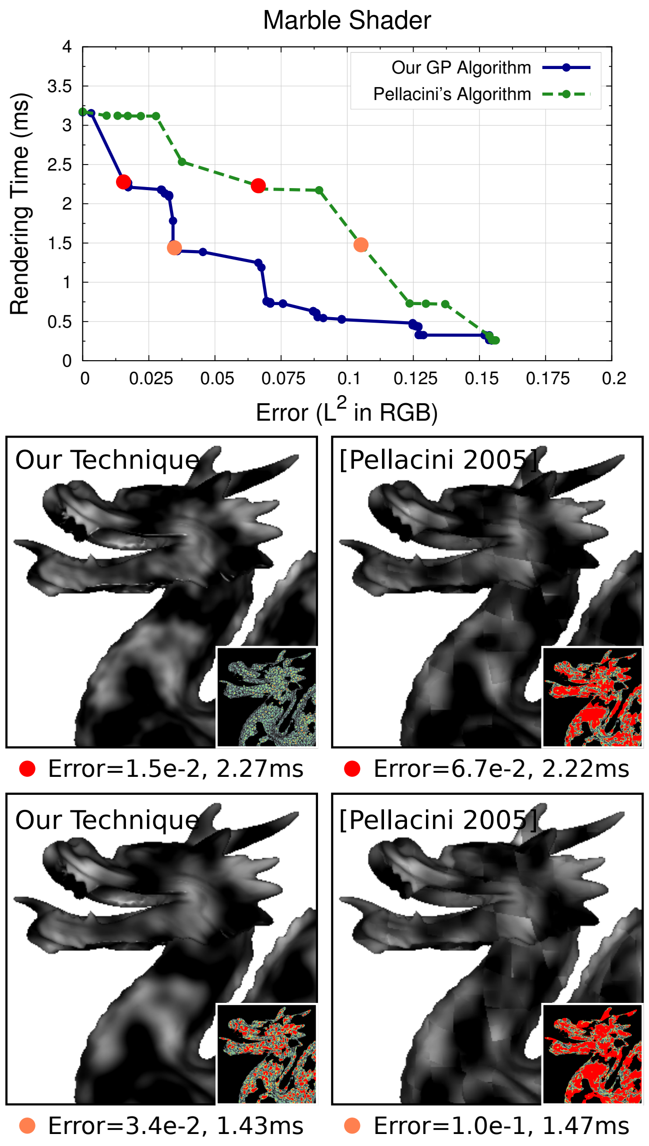

Figure 8: Top: *Comparison of the output of our system to that of Pellacini's.* **Bottom:** *Side-by-side comparisons of the best shaders each method produces for a fixed rendering budget under 2.5ms (red dots) and 1.5ms (orange dots).*

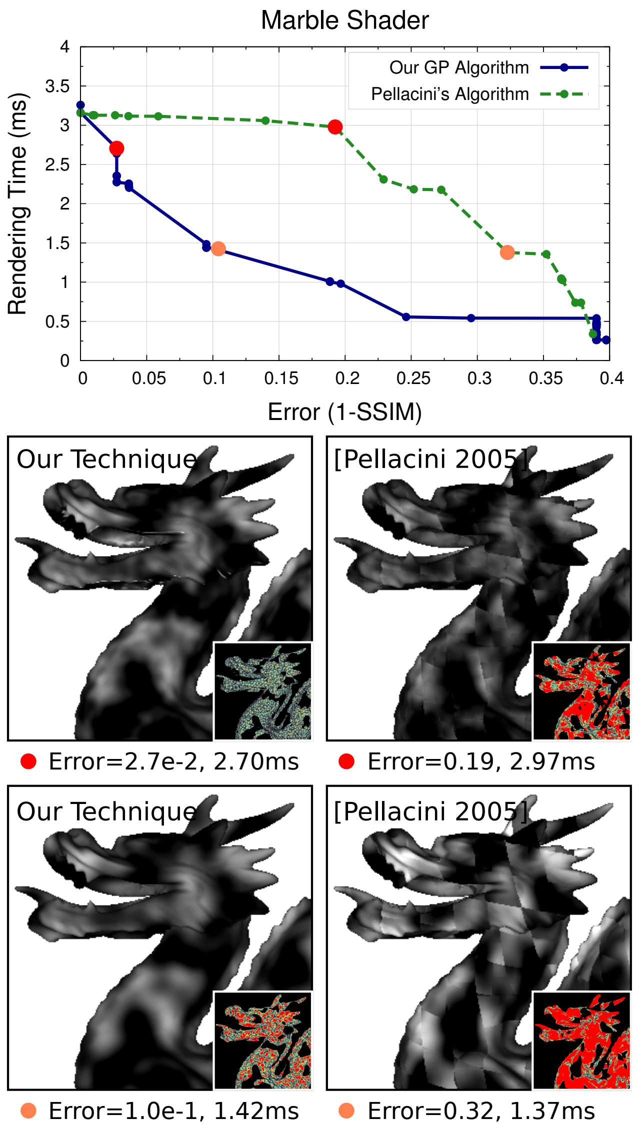

Figure 9: Top: *Comparison between our system and Pellacini's using the SSIM error metric.* **Bottom:** *Side-by-side comparisons of the best shaders each method produces for a fixed rendering budget of 0.30ms (red dots) and 0.15ms (orange dots).*

every nodes in the AST. Computing an average value of a node in one pass may require rendering results of all prior passes, which is impractical for a long rendering sequence with many passes.

In addition, our technique converges rapidly after considering far fewer unique variants than Pellacini's. For example, we generate over 3,000 total unique variants for the Marble and Trashcan shaders, while Pellacini's technique generates over 25,000. As a result, Pellacini's approach takes more than twice as long: 16 total hours for those two shaders compared to 6 hours for our technique.

4.7 Alternate Perceptual Error Metric

While measuring image fidelity using the L^2 error in RGB space is simple to understand and facilitates comparison with previous work, many alternative perceptually-motivated distance metrics have been proposed. To demonstrate our technique's direct applicability to other error metrics, we consider one such metric, the Structural Similarity Index (SSIM), which operates on patches of pixels [Wang et al. 2004]. This metric has been demonstrated to be more consistent with human perception than L^2-distance for grayscale images.

Figure 9 shows the results of applying our technique, and that of Pellacini, to the (grayscale) Marble shader with error evaluated for both using SSIM (8×8 non-overlapping window size). Since SSIM is a similarity metric, $1-$SSIM increases with error; variants above 0.4 error were not considered. As in Figure 8, we consider the same rendering budgets of 0.30ms (red dots) and 0.15ms (orange dots). At 0.30ms, our shader removes the highest frequency of the Perlin noise calculation. By contrast, the best shader produced by Pellacini's technique introduces an order-of-magnitude more error (note blocky visual artifacts, which do not show up until later in Pellacini's sequence when using RGB in Figure 8). At 0.15ms, our shader renders only the highest and the lowest frequency of the noise calculation, producing less than half the error of the corresponding shader from Pellacini. Note that this is a different optimization than what our technique found when optimizing for RGB L^2 error (i.e., retaining only the lowest-frequency noise bands).

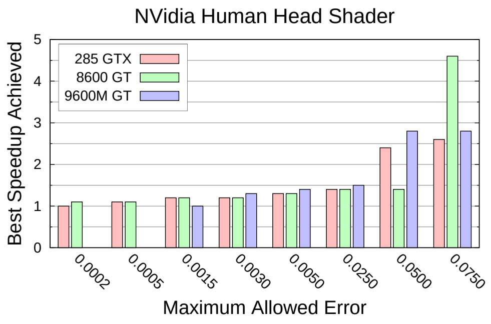

Figure 10: *Best speedup achieved by our optimization technique when applied to different hardware over a range of allowed error budgets. Our technique was run separately for each hardware configuration. Note that at the lowest allowed error, no acceptable optimizations are produced for the 9600M GT mobile hardware. Note also that at the two highest error budgets shown, the 285 GTX and 8600 GT show alternate optimization efficacy (see text).*

Because our technique treats image error as one more opaque objective to be minimized, any such metric can be used. While previous techniques can simply "plug in" other error metrics as well, this is not always well founded. For example, the approach of Pellacini implicitly assumes L^2 RGB error in two steps: by biasing produced shaders to the average RGB output color of the original shader, and by assuming that minimizing delta error will necessarily produce a faster shader. This distinction can be seen in Figure 9, where the separation between the two lines, and thus the relative improvement provided our approach, is greater than in the RGB case in Figure 8.

4.8 Alternate Hardware Targets

Not all tradeoffs are equally good choices on different hardware. To demonstrate that our technique can produce different hardware-specific optimizations, we produced simplified sequences of the NVidia Human Head shader for three different graphics cards. The cards used were the NVidia 285 GTX, 8600 GT and 9600M GT. The 285 GTX, the card used in all of our other experiments, is a high end desktop card with 240 CUDA cores running at 648 MHz; its average rendering time for Human Head is 6.42ms. The 8600 GT is an older desktop model with 32 cores running at 540 MHz; its average rendering time is an order of magnitude slower at 61.4ms. Finally, the 9600M GT is a low end card for mobile laptops with 32 cores running at 120 MHz; its average rendering time was slowest at 66.2ms. To reduce the required running time, we used a smaller set of indicative frames for error and performance evaluation.

The results are shown in Figure 10. The pink bars only roughly correspond to the Figure 5 (e.g., 2.2x improvement at 0.05 error). We first note that at the most stringent error bounds (0.002 and 0.005), no acceptable variant was produced on the mobile card that was faster than the original. By contrast, the other two cards were able to simplify small amounts of the lighting calculation in the final pass, but the mobile card's hardware was insufficient to take advantage. The next point of interest is at 0.05 error where the 285 GTX and 9600M GT were both able to remove some of the Gaussian samples, an optimization that was not equally profitable on the 8600 GT. However, at the very relaxed error bound of 0.075 (a weaker bound than that shown in Figure 5), the 8600 GT, which has a high clock speed but a comparatively small number of cores and small amount of memory, profited from removing all of the Gaussian samples, substantially increasing the memory bandwidth in the final pass. Note that all speedups presented are relative, so even though the 8600 GT is able to improve upon its baseline performance by 4.5 at the most relaxed error bound, it is still much slower than the 285 GTX in absolute terms.

This demonstrate that our approach produces qualitatively and

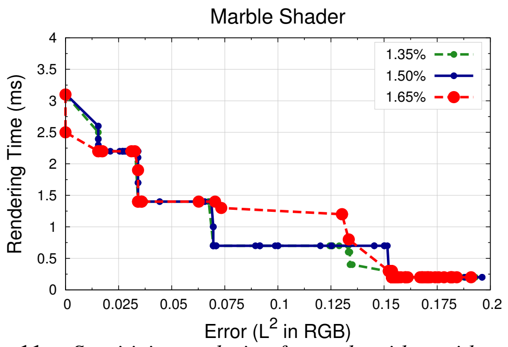

Figure 11: *Sensitivity analysis of our algorithm with respect to key parameter mutation rate. Our default mutation rate is 0.015; the two other lines represent* $\pm 10\%$.

quantitatively different optimization tradeoffs for different hardware profiles. Among other benefits, this allows for scenarios in which suites of shader variants are produced in advance at development time and the best sequence is selected locally at install time. The supplemental material further highlights our system's ability to target different architectures by presenting results for NVidia's mobile Tegra 2 chipset.

4.9 Parameter Discussion and Sensitivity

A key concern in any metaheuristic or search optimization technique is the sensitivity of the approach to the choice of internal algorithmic parameters. Two major parameters for our technique are the number of generations (i.e., the number of variants considered or the amount of work put into the search) and the mutation rate (i.e., how aggressively the search considers possible changes). In this sort of genetic algorithm, the number of generations is less important as long as it exceeds a minimum threshold [Weimer et al. 2009, 2010]. In this domain, after a certain number of generations, the Pareto frontier converges to a series of points from which improvements are made only very rarely. In genetic algorithm terms, this is related to the use of *elitism* [Koza 1992], in which the best variants in one generation may be carried over to the next (see Step 7 of Section 3, in which the initial population is considered in the final selection). In our experiments, we found that the single pass shaders settled in under twenty generations and the multi-pass shader settled in under eighty; adding additional generations beyond that did not notably improve output quality.

Experiments suggest that the algorithm is also stable with respect to the mutation rate. Figure 11 shows the result of applying our technique to the Marble shader with the mutation rate increased by 10% and then also decreased by 10%. The area under the Pareto frontier changes by an average of 7.5%, suggesting stability with respect to this parameter.

5 Limitations and Future Work

Our method requires a representative rendering sequence, typically a few hundred frames, to estimate expected performance and error of each shader variant. While this is both potentially easier and more accurate than the assumption of uniformly distributed input ranges [Pellacini 2005], it does require the user to capture and record an indicative series of inputs. As with all profile-guided optimizations [Muchnick 1997], if conditions in the field are very different from the input sequence, the simplified shaders may decrease performance while increasing error. However, we view the task of constructing an indicative workload as orthogonal to the task of optimizing a shader, just as constructing a comprehensive test suite is orthogonal to the task of automated program repair [Weimer et al. 2009, 2010].

Our system does not explicitly handle textures (cf. Olano et al. [2003]). While we might change or remove an expression that includes a texture lookup (as in the Trashcan shader), a dedicated texture optimizer would be able to consider some situations more rapidly than our method. However, the two techniques are almost orthogonal, and could be applied together.

Although we have focused on pixel shaders, our technique could apply equally well to vertex and geometry shaders. Future research might include developing tools that consider all of these components together, possibly in addition to the scene geometry. Additionally, perhaps rather than optimizing for overall performance, we might optimize for subsystem-specific performance (e.g., minimize memory bandwidth used or memory bank conflicts encountered while leaving everything else as untouched as possible).

6 Conclusion

We have presented a system for simplifying procedural shaders that builds on and extends prior work in Genetic Programming (GP) and software engineering. The key insight is that a shader's source code contains the seeds of its own optimization. We therefore consider transformations to an input shader that take the form of copying, reordering, and deleting statements and expressions already available. GP provides an optimization framework for efficiently exploring the space of shader variants induced by this set of transformations, with a goal of identifying those that provide a superior time/quality trade-off. Our system achieves this in part via a GPU-based method for rapidly evaluating candidate shaders. The output of our system is a sequence of increasingly faster shaders that can be used in place of the original. Our approach is powerful, considering a wide space of transformations and producing shaders that are, on average, three times as fast as those of previous work at equal fidelity. Our approach is also general: unlike previous work, it can be used to optimize multiple-pass shaders and can use perceptually-based error metrics in a well-founded manner.

Acknowledgements

We wish to thank Stephanie Forrest along with the CAVGRAPH and SIGGRAPH reviewers for their helpful and constructive feedback. We'd also like to thank NVidia and XYZRGB Inc. for providing access to the "human head" shader and model. This research was partially supported by NSF CAREER Awards CCF-0747220 and CCF-0954024, NSF grant CCF 0905373, AFOSR grants FA8750-11-2-0039 and FA9550-07-1-0532, and DARPA grant FA8650-10-C-7089.

References

BAVOIL, L., SAINZ, M., and DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, New York, NY, USA. ACM, pages 22:1–22:1. ISBN 978-1-60558-343-3. URL `http://doi.acm.org/10.1145/1401032.1401061`.

BLINN, J. F. 1977. Models of light reflection for computer synthesized pictures. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 192–198.

COOK, R. L. 1984. Shade trees. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*.

COOK, R. L. and TORRANCE, K. E. 1981. A reflectance model for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH)*, pages 7–24.

DEB, K., PRATAP, A., AGARWAL, S., and MEYARIVAN, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., and HUNT, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pages 21–30.

D'EON, E., LUEBKE, D., and ENDERTON, E. 2007. Efficient rendering of human skin. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*.

HASSELGREN, J. and AKENINE-MOLLER, T. 2006. An efficient multi-view rasterization architecture. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*.

KOZA, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

MILLER, B. L. and GOLDBERG, D. E. 1996. Genetic algorithms, selection schemes, and the varying effects of noise. *Journ. of Evolutionary Computation*, 4(2):113–131.

MITCHELL, M., HOLLAND, J. H., and FORREST, S. 1993. When will a genetic algorithm outperform hill climbing. In *Advances in Neural Information Processing Systems*, pages 51–58.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann. ISBN 1-55860-320-4.

NECULA, G. C., MCPEAK, S., RAHUL, S. P., and WEIMER, W. 2002. Cil: An infrastructure for C program analysis and transformation. In *International Conference on Compiler Construction*, pages 213–228.

NEHAB, D., SANDER, P. V., LAWRENCE, J., TATARCHUK, N., and ISIDORO, J. R. 2007. Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware*.

OLANO, M., KUEHNE, B., and SIMMONS, M. 2003. Automatic shader level of detail. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*.

PELLACINI, F. 2005. User-configurable automatic shader simplification. *ACM Transacations on Graphics (Proc. SIGGRAPH)*.

PERLIN, K. 1985. An image synthesizer. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*.

SCHERZER, D., JESCHKE, S., and WIMMER, M. 2007. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*.

SITTHI-AMORN, P., LAWRENCE, J., YANG, L., SANDER, P., NEHAB, D., and XI, J. 2008. Automated reprojection-based pixel shader optimization. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 27(5):127.

WANG, Z., BOVIK, A. C., SHEIKH, H. R., and SIMONCELLI, E. P. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4).

WEIMER, W., FORREST, S., LE GOUES, C., and NGUYEN, T. 2010. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116.

WEIMER, W., NGUYEN, T. V., LE GOUES, C., and FORREST, S. 2009. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 364–374.