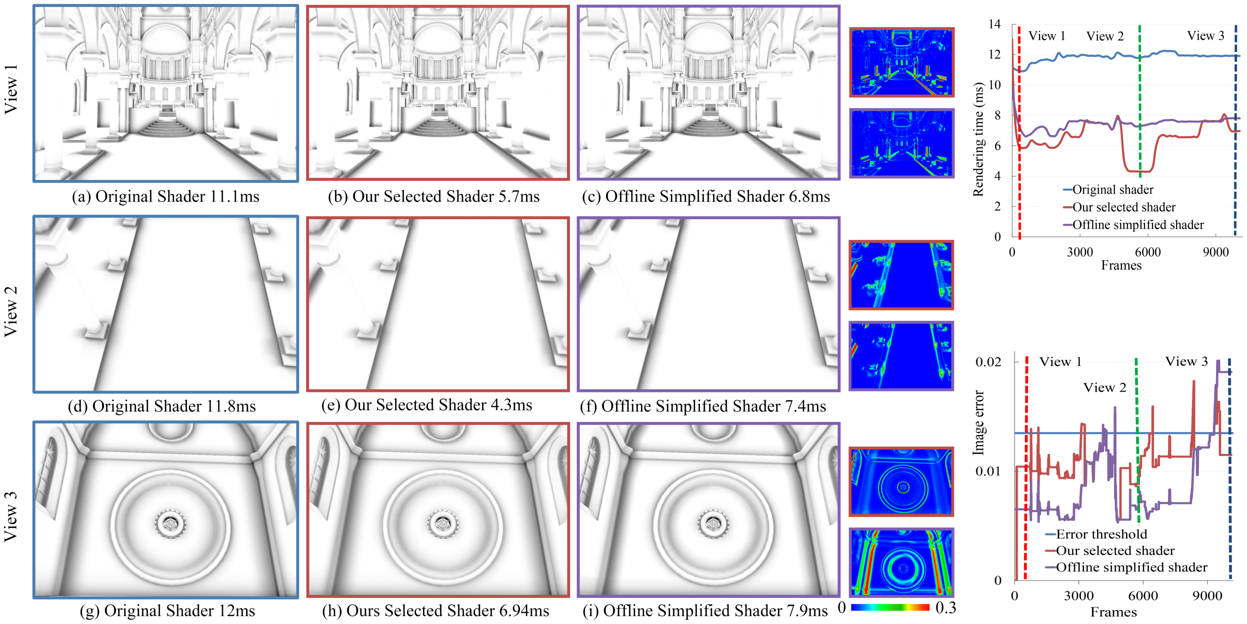
Eurographics Symposium on Rendering 2018
T. Hachisuka and W. Jakob
(Guest Editors)

Volume 37 (2018), Number 4

Runtime Shader Simplification via Instant Search in Reduced Optimization Space

Yazhen Yuan, Rui Wang[†], Tianlei Hu, Hujun Bao

State Key Lab of CAD&CG, Zhejiang University
†Corresponding author: rwang@cad.zju.edu.cn



(a) Original Shader 11.1ms (b) Our Selected Shader 5.7ms (c) Offline Simplified Shader 6.8ms

(d) Original Shader 11.8ms (e) Our Selected Shader 4.3ms (f) Offline Simplified Shader 7.4ms

(g) Original Shader 12ms (h) Ours Selected Shader 6.94ms (i) Offline Simplified Shader 7.9ms

Figure 1: *Result comparison of the original HBAO shader, our runtime selected shader and the offline simplified shader. In this demo, our system automatically searches optimal simplified shaders under a tolerated visual error threshold, and achieves 1.6 to 2.5 times speedup comparing to the original shader. Moreover, compared with the traditional offline approach, our method has better performance and quality at almost all views. This proves that our system is capable of better adapting to the runtime context and producing instant optimal results.*

Abstract

Traditional automatic shader simplification simplifies shaders in an offline process, which is typically carried out in a context-oblivious manner or with the use of some example contexts, e.g., certain hardware platforms, scenes, and uniform parameters, etc. As a result, these pre-simplified shaders may fail at adapting to runtime changes of the rendering context that were not considered in the simplification process. In this paper, we propose a new automatic shader simplification technique, which explores two key aspects of a runtime simplification framework: the optimization space and the instant search for optimal simplified shaders with runtime context. The proposed technique still requires a preprocess stage to process the original shader. However, instead of directly computing optimal simplified shaders, the proposed preprocess generates a reduced shader optimization space. In particular, two heuristic estimates of the quality and performance of simplified shaders are presented to group similar variants into representative ones, which serve as basic graph nodes of the simplification dependency graph (SDG), a new representation of the optimization space. At the runtime simplification stage, a parallel discrete optimization algorithm is employed to instantly search in the SDG for optimal simplified shaders. New data-driven cost models are proposed to predict the runtime quality and performance of simplified shaders on the basis of data collected during runtime. Results show that the selected simplifications of complex shaders achieve 1.6 to 2.5 times speedup and still retain high rendering quality.

CCS Concepts

•***Computing methodologies*** → *Shader simplification, Runtime optimization;*

1. Introduction

Modern flexible rendering pipelines allow programmers to customize their own computations in shaders and obtain diverse effects. With the increasing complexity of shader code and the wide use of graph-based shader authoring tools, the simplification of shader has become more and more important and challenging. Several approaches have been proposed and developed to automate the entire process [OKS03, Pel05, SALY*08, SAMWL11, WYY*14, HFTF15]. However, most of these methods simplify shaders in a preprocess, which has several limitations. First, the preprocess lacks an actual runtime context, e.g., scenes, positions of lights or cameras, etc. Although people usually use some example contexts or average all possible values of parameters to create a general context, the adaptability to the change of runtime context is compromised (Fig. 1). Second, code that performs well on one type of graphics card often faces bottlenecks on another. In this case, the simplification process must be repeated to port it to new hardware. Given a large number of hardware customized shaders in current AAA games (Over 18000 in Bungie's Destiny [HFTF15]), optimizing all of them for various hardware platforms in a preprocess is impractical.

To address these limitations and inspired by the idea of recently developed online auto-tuning work [BHT*10, JTD*12], in this paper, we explore the problem of runtime shader simplification and develop a system. The proposed system is able to optimize the shader according to the present scenes or parameters, and instantly selects an optimal simplified shader for the current context. Although our system is only capable of optimizing one or two of the most time-consuming shaders, the reduced overall computational costs and resources have a better chance to be utilized for performing other tasks, such as enhancing quality of visual effects or improving accuracy of simulation.

Two main challenges are addressed to bring such a system into practice: 1) The enormous number of all possible simplified variants form an extremely large optimization space. 2) The runtime optimization of simplified shaders should be performed fast enough to satisfy users' interactive requirements, implying simplifying shaders in tens or hundreds of milliseconds. For these challenges, we propose a two-stage simplification scheme. The first stage involves a reduced optimization space containing a set of representative simplified shader variants, which can be managed and used for runtime search. Here, a preprocess is employed to select these shader variants. We present a new representation called the simplification dependency graph (SDG) to organize these representative simplified shader variants, as well as a specific clustering algorithm to construct it. Two heuristic estimates of quality and performance are presented in a data-oblivious manner to map each simplified shader variant to a high-dimensional space. In such a space, similar simplified shaders are grouped to form a compact optimization space. The second stage is the runtime optimization process. A parallel discrete optimization algorithm is developed to find optimal simplified shaders from the reduced optimization space. New data-driven cost models that consider the costs of simplified shaders being already evaluated are presented to predict the quality and performance of unevaluated simplified shaders. The results show that our system is capable of efficiently optimizing shaders on different scenes with different runtime parameters from scratch.

The contributions of this work are as follows:

- The first automatic shader simplification system that selects the optimal simplified shader with runtime context fast and efficiently.
- A new representation, simplification dependency graph (SDG), to organize simplified shaders and form the optimization space.
- Novel quality and performance cost models to predict costs of simplified shaders in preprocess and at runtime.
- A runtime multi-objectives search algorithm to find the optimal simplified shaders.

2. Related Work

Code Auto-Tuning Auto-tuning tools help programmers automate the tedious and error-prone process of tuning and porting application code among different hardware architectures and platforms. With the recent developments of hardware and software, many studies have explored the automation of the process of tuning the code for (parallel) computers [NTCS10]. Several domain-specific auto-tuners such as ATLAS [WD98] for dense linear algebra, OSKI [VDY05] for sparse linear algebra, FFTW [FJ05] and SPIRAL [XJJP01] for signal processing, etc., have been successful in producing highly-optimized architecture-specific code. These successes have motivated a general interest in extending the offline auto-tuning methodology to the online tuning framework. Baskaran et al. [BHT*10] proposed a method to achieve the online tuning of parallel program parameters (e.g., tile sizes). Tiwari and Hollingsworth [TH11] presented an auto-tuning framework to compile code and tune parameters at runtime for parallel programs. Jordan et al. [JTD*12] developed a multi-objective autotuning framework to balance efficiency and speedup.

Generally, the automatic simplification of shaders is a domain-specific auto-tuning technique. However, compared with traditional auto-tuning methods, shader simplification has its specific requirements. First, shader simplification seeks to exploit the tradeoffs between performance and visual quality. Therefore, it is not a variable-accuracy auto-tuner, whereas many traditional auto-tuners are very sensitive to errors and only tune performance parameters, e.g., tile sizes of parallel programs on multicores. Second, different shader simplification rules usually result in a discrete optimization space, whereas many auto-tuners entail a continuous parameter space. Finally, shader simplification optimizes the computation over rendering pipelines, thus requiring GPU-specific integration. Hence, despite the availability of several open auto-tuning frameworks [AKV*14, SBR*15] and online auto-tuners [BHT*10, TH11, JTD*12], the shader simplification during runtime remains an open problem.

Shader Simplification The pioneering work of shader simplification [OKS03] was proposed to simplify procedural shaders. Inspired by this work, Pellacini [Pel05] presented a method to automatically generate a sequence of simplified shaders by error analysis of a fixed set of expression rules. Sitthi-Amorn et al. [SAMWL11] introduced the genetic programming into automatic shader simplification to search for the Pareto-optimal simplified shaders that best represent the tradeoffs between rendering time and quality. Wang et al. [WYY*14] proposed new simplification rules that treat the shader simplification as signal approximation on surfaces to simplify shaders across multiple shader stages. However, these methods perform optimization in an offline scheme and are thus unable to optimally adapt to the runtime changes of context. Additionally, such an offline preprocess is usually time consuming, espeically when the scene features a large domain of uniform parameters, e.g., a scene with many possible viewports or light directions. In these cases, existing methods need to evaluate one simplified shader at every viewport and direction, and average the performance and quality errors. In contrast, our proposed approach only optimizes for the instant parameters, therefore is faster and more efficient, and the simplified shader is optimal for the present context.

Scherzer et al. [SJW07] and Sitthi-amorn et al. [SALY*08] explored the idea of utilizing temporal coherence between different frames. A cache-based re-projection strategy was used to simplify shaders. However, this method still requires hours of training the error and performance model for each context.

Dorn et al. [DBLW15] presented a new method that automatically generates band-limited procedural shaders. Yang et al. [YB18] extended this idea to smooth procedural shader with mean-variance of the program. These methods only focus on analyzing and transforming a set of analytic expressions to improve a specific visual property (low sampling rate), hence they can not be applied to general shader simplification.

He et al. [HFTF15] proposed a system that automatically generates level-of-detail (LOD) shaders. They also introduced a heuristic model to estimate performance and a fast search algorithm to reduce the simplification process from hours to minutes. We extend their work from offline simplification to simplification during runtime by reducing the optimization time from minutes to tens of milliseconds. This significant improvement in optimization time results in a new shader simplification framework that comprises different cost models and a search algorithm.

He et al. [HFF16] proposed another system for rapid exploration of shader optimization choices, which virtualizes the shader and the rendering engine. In this way, it can move the computation at different rendering stages and simplify the shader. Their recent work [HFH*17] proposed a new modular shader language designed to update parameters more efficiently. The goals of both of these works are different with our interest that we seek the optimal shader variant under changing rendering context.

3. Overview

This section first briefly describes the problem of automatic shader simplification, and then provides an overview of the proposed solution.

3.1. Problem Definition

Traditional automatic shader simplification is an offline process to generate a sequence of increasingly simplified shader variants f_k. The process starts from an original shader f_0, where each step in the sequence has a decreasing rendering quality between f_0 and f_k, as well as a decreasing computation cost between f_{k-1} and f_k. While using one shader variant f_k to render a particular scene, the computation cost is usually measured by the rendering time t, and the rendering quality can be measured by errors defined at pixels as the following:

$$\varepsilon(\mathbf{v},\mathbf{u}) = \int\int_{xy} \parallel f_0(\mathbf{v},\mathbf{u}) - f_k(\mathbf{v},\mathbf{u}) \parallel_2 dxdy \qquad (1)$$

where x and y are the integer pixel coordinates, $\mathbf{v}$ is a set of input geometry primitives with attributes, $\mathbf{u}$ is a set of uniform parameters (such as camera position and light position), and $||\cdot||_2$ is the norm, (i.e., L^2 norm used in this paper). Based on the measured rendering time and image error, shader simplification is formulated as a multi-objective problem in the two-dimensional cost space (ε, t), and seeks a Pareto-optimal solution [SAMWL11, WYY*14, HFTF15]. To consider the variations of geometry primitives and uniform parameters, offline shader simplification usually integrates the error over the domain of each uniform parameter, and averages over different geometry models. As a result, the Pareto-optimal shaders optimized by these offline approaches are only averagely optimal with respect to all the uniform parameters and different geometry models, but not instantly optimal to the present context. In addition, the shader may fail in some general cases, such as, changes in the rendering context that were not considered during offline optimization.

The proposed simplification framework, instead, seeks to find optimal shader simultaneously in response to changes of rendering context. The users sets an error tolerance of rendering quality as $\varepsilon_{\max}$. Theoretically, the optimization problem can be formulated as the following:

$$f_{opt} = \arg\min_f t(\mathbf{v},\mathbf{u}) \quad \text{s.t.} \;\; \varepsilon(\mathbf{v},\mathbf{u}) \leq \varepsilon_{\max}, \qquad (2)$$

where f_{opt} is the optimal shader that produces less error than $\varepsilon_{\max}$ but runs the fastest in terms of the runtime context of geometry $\mathbf{v}$ and the uniform parameters $\mathbf{u}$.

3.2. System Design

Designing a practical shader simplification system that optimizes the shader for current rendering context entails several challenges. First, because the system computes during runtime, the optimization process must be very fast to satisfy the interactive requirements of users. However, the number of shader variants for a complete application can be enormous. Simplified shader variants should be judiciously selected to keep the optimization time at a manageable level. Second, given that shader simplification is carried out in a two-dimension space, the search algorithm should be very efficient to explore the space at runtime.

To solve these two challenges, we design a two-stage system. The system takes an HLSL shader program as input, initially preprocesses on it to construct an optimization space and then explores in the space at runtime. Fig. 2 illustrates these two stages.

- Preprocess stage: process and prepare simplified shader variants for runtime optimization, including parsing HLSL shader, optimizing the shader variants into a small set of representative shader variants, compiling these shader representative variants, and generating a simplification dependency graph.
- Runtime optimization stage: iteratively optimize the shader during runtime, including monitoring runtime context changes, searching the candidates for optimal shader variant, evaluating these candidate shader variants and selecting the optimal one.

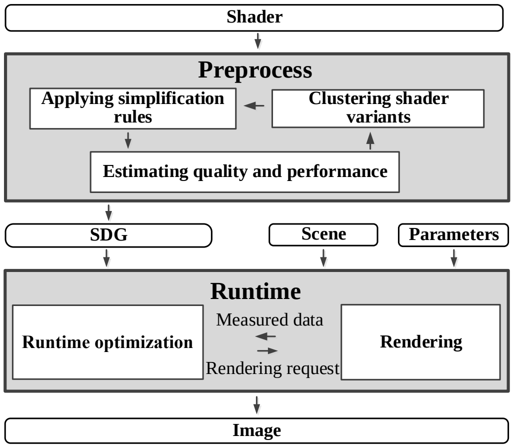


Figure 2: *System overview*

4. Preprocess Stage

Various simplification rules have been proposed and introduced to simplify the shader code. While applying these rules to a shader, the number of simplified shader variants grows exponentially with the number of instructions of the code. A brute-force approach to search an optimal simplified shader from all of them at runtime is, however, impractical. Therefore, we take a preprocess stage to simplify the optimization space into an affordable scale that can be efficiently managed and explored. This simplification mainly takes three steps. First, the shader code is parsed to create its abstract syntax tree (AST) and program dependency graph (PDG) [FOW87]. Then, simplification rules are applied to generate simplified shader variants. Finally, these variants are clustered into representative shader variants by two estimates, *parameter influence* and *instruction cost*, and organized into a new representation, the simplification dependency graph. All the representative shader variants are compiled into binary code for use at runtime.

4.1. Simplification Rules

Inspired by previous approaches, three types of simplification rules were considered: the expression reduction rules modified from the previous approach [Pel05], the shader code transformation rule, and the surface subdivision rule [WYY*14].

Expression reduction is a type of effective and efficient rules that eliminates algebraic operations. It has been used widely in several shader simplification systems [Pel05, SALY*08, SAMWL11, HFTF15]. In this paper, we adopt three expression reduction rules proposed by Pellacini [Pel05], but discard the rules requiring runtime data and information, e.g., the average substitution rule. More precisely, let us consider the binary operator $a \bigotimes b$, when both a and b are expressions. We will have two simplified expressions, $a \bigotimes b \to a$ and $a \bigotimes b \to b$. For loops from c_b to c_e, we randomly choose two values, i and j, and substitute the loops from $c_b + i$ to $c_e - j$ instead, where we ensure that $c_b + i < c_e - j$.

Code transformation [WYY*14] moves the slice of one piece of code computing on pixels to on vertices. This transformation can be regarded as using a linear basis defined on vertices to approximate pixel-wise computations. The surface subdivision rule is combined with the code transformation rule that tessellates triangles into smaller pieces. Although the tessellation takes extra time, tessellated triangles improve code transformation by providing a denser domain to approximate the original computations on pixels. Owing to the different tessellation levels, tessellation rule can generate a set of simplified shader variants with different computation times and rendering qualities.

4.2. Constructing the Optimization Space

Once these rules are combined and applied to the AST and PDG of the original shader, a collection of all simplified shader variants is obtained. The next task is to select some representative variants to represent all other variants and form the optimization space for the runtime optimization.

Note that while we select these representative variants, the runtime context is unavailable at this time. Therefore, the selection is performed in a context-oblivious manner. We utilize the dependence of code and simplification rules to obtain two heuristic estimates of error and performance costs. These estimates map each simplified shader variant to a high-dimensional space, in which variants are clustered into groups, and organized into the SDG. These steps are described in following sections.

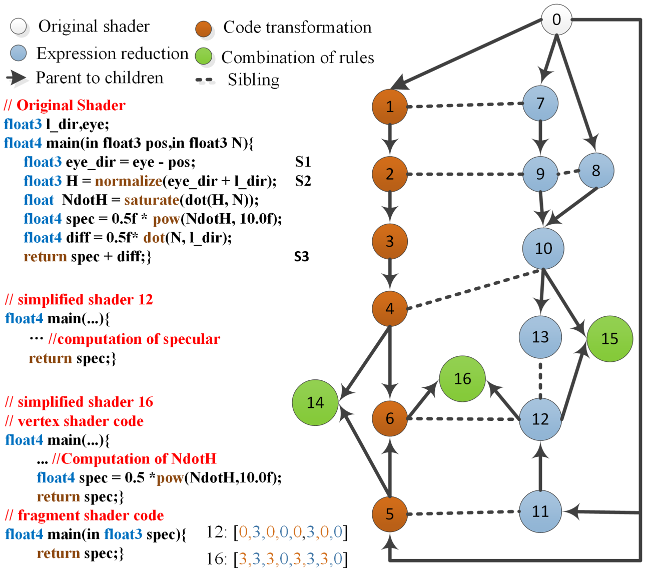
4.2.1. Simplification Dependency Graph

The SDG is a directed acyclic graph where the root node is the original shader, and each children node is a simplified shader variant. We use a directed edge to represent the dependency of simplification from one shader variant to another; specifically, one node A having a directed edge to node B indicates that the shader variant B is simplified from A by applying one rule on A. Fig. 3 shows the SDG of an example shader. In the figure, the nodes directly linking to the original shader are shader variants with only one simplification rule. As more and more rules are applied to these nodes and their children, the entire simplification space is represented in a graph.

To better understand the dependence in the graph, let us consider two statements, S_1 and S_2 in Fig. 3, where S_1: *eye_dir = eye - pos*; and S_2: *H = normalize(eye_dir + l_dir)*. Applying the transformation rule to statement S_2 (Variant 2) will inevitably transform statement S_1 (Variant 1). In this case, the shader variant with the transformation of S_1 will be the parent of the variants with the transformation of S_2. When the expression reduction rule is applied to statement S_2 as *H = normalize(l_dir)* (Variant 9), S_1 is eliminated. In this case, applying the expression reduction rule on S_1(Variant 7) is seen as the parent of Variant 9. To bridge nodes that simplified by different rules, we add sibling edges between two nodes that have different rules applied on same expression/statement, take Variant 2, 8, 9 as an example. The green nodes represent variants that are generated by applying more than one simplification rule upon original shader. For example, Variant 6 moves the statement S_3: *return spec + diff;* to vertex shader, and Variant 12 reduces the diffuse light calculation, so Variant 16 only has specular lighting calculated in the vertex shader.

4.2.2. Estimating Quality and Performance

Our goal is to reduce the number of variants and use only a small set of representative ones to approximate the space of all variants. Ideally, if these representative shader variants well represent other



```
// Original Shader
float3 l_dir,eye;
float4 main(in float3 pos,in float3 N){
    float3 eye_dir = eye - pos;                    S1
    float3 H = normalize(eye_dir + l_dir);         S2
    float  NdotH = saturate(dot(H, N));
    float4 spec = 0.5f * pow(NdotH, 10.0f);
    float4 diff = 0.5f* dot(N, l_dir);
    return spec + diff;}                           S3

// simplified shader 12
float4 main(...){
     ··· //computation of specular
     return spec;}

// simplified shader 16
// vertex shader code
float4 main(...){
       ... //Computation of NdotH
       float4 spec = 0.5 *pow(NdotH,10.0f);
       return spec;}
// fragment shader code
float4 main(in float3 spec){
       return spec;}
```

Figure 3: *The simplification dependency graph*

variants, then in the runtime optimization, the Pareto-optimal variants of the representative shader variants should be or be close to the original Pareto-optimal variants in terms of quality and performance space (ε, t). However, the quality and performance are unavailable at this time. Therefore, we need new metrics to predict the actual rendering quality and performance. Two heuristic cost estimates, namely *parameter influence* and *instruction cost*, are proposed in this paper, to replace the actual costs, quality and performance, of one shader variant. Note that these cost estimates are not designed to accurately compute the actual quality or performance of simplified shader variants, but only need to roughly reflect the order of quality and performance in the cost space when we actually evaluate them with real data.

Parameter influence is designed to measure how much impact the different simplification rules have on one shader variant with respect to the inputs(including geometry attributes and uniform parameters). The basic observations behind this estimate as follows: first, if two variants both simplify one parameter, then at runtime, these errors produced by the two variants have correlations; second, for the same expression or statement, different simplification rules will produce different errors; finally, the more simplifications performed on the statements of one parameter, the more errors will be produced. Based on these observations, we define the parameter influence as a vector, of which each element corresponds to one input uniform parameter with one type of simplification rules. For example, in Fig. 3, the shader has two uniform parameters and two geometry attributes, and we only apply two types of simplification rule, the expression reduction and the code transformation here for simplicity. Therefore, the parameter influence is an eight-element vector. The value of the $(2k+i)$-th element measures how much the computations of the k-th parameter have been simplified by i-th type of rules as

$$q[2k+i] = \sum_{j}^{N} r_i^j(k) \tag{3}$$

where N is the total simplified expressions by the i-th rule directly or indirectly dependent on the k-th parameter in this variant, r_i^j returns how many elements are simplified by the i-th rule in the j-th simplification on the k-th parameter.

Examples are shown in Fig. 3. According to Eq. (3), we denote the parameter influence vectors as $[N_0, N_1, pos_0, pos_1, eye_0, eye_1, l\_dir_0, l\_dir_1]$, in which the code transformation and expression reduction rules are denoted by subscripted 0 and 1 respectively. As can be seen, Variant 12 deletes the diffuse term from the original shader by reducing the uniform parameter N and l_{dir}. Both N and l_dir are float3, thus the $r_i^j(k)$ returns 3. There is only one expression with respect to N and l_dir. As a result, the parameter influence vector of Variant 12 is [0,3,0,0,0,3,0,0]. Another example is Variant 16, which is a combination of different rules: the diffuse term is first reduced, and then the specular term is moved to the vertex shader. The specular term itself has a parameter influence vector [3,0,3,0,3,0,3,0]. Based on the combination of different rules, the simplified parameter influence for Variant 16 is then denoted as [3,3,3,0,3,3,3,0].

Instruction cost is designed to estimate the computation cost. For one shader, the computation cost directly relates to the number and type of instructions of the shader code. However, obtaining an accurate estimation of the cost of different instructions is a challenge. In this paper, we use the performance model proposed by He et al. [HFTF15]. This model weighs the cost of DAG nodes by the number of scalar instructions needed to perform them (e.g., a float4 addition incurs cost 4), and assigns a cost of 100 units to texture operations. To unify the total cost of the entire shader program including per-vertex and per-fragment computations, a weighted cost of the vertex, C_v, and fragment, C_f, plus a penalty term for the size of the vertex-fragment interface, N_a, is employed as follows:

$$C_{total} = 0.3C_v + 0.7C_f + 10N_a \tag{4}$$

4.2.3. Clustering Simplified Shader Variants

By using these two heuristic cost estimates, we can map all simplified shader variants to a high dimensional space, and cluster them into a small number, e.g., hundreds or thousands, of representative simplified shader variants. Given that the full optimization space could be very large, especially for a complex shader with several simplification rules, we adopt the idea of the genetic programming [WYY*14, SAMWL11] to iteratively cluster shader variants. In supplemental document, we give the pseudo-code of this algorithm.

In the clustering, we always maintain a set of representative variants as seed variants, where the original shader is set as the initial seed variant. Then, at each iteration, we use these seed variants to populate a new generation of simplified shader variants by applying simplification rules on only one expression or statement in each seed variant. So based on our SDG's definition, for each populated variant, it connects to the seed variants that generated it.

While clustering variants, since these two cost estimates represent different purposes of the optimization, combining them numerically and defining a scalar distance function are difficult. Instead, observing that the instruction cost is a scalar metric, we divide the values of the instruction cost into buckets, and group variants in each bucket by their parameter influence vectors. More specifically,

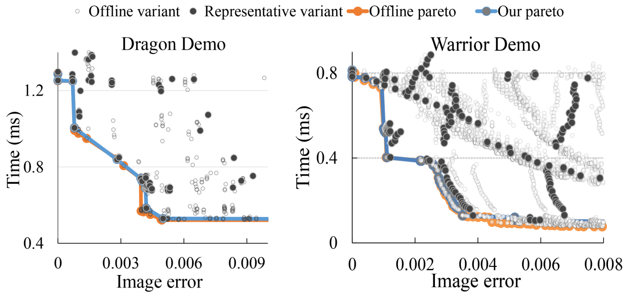


Figure 4: *Comparison of our reduced and offline optimization space.*

first, we sort all shader variants by their instruction costs. Then, we uniformly divide the values of costs into N groups, i.e., N=8 as default, and in each group, we use vector dot $(\cdot)$ to measure the difference between two parameter influences. K-Means clustering is used to cluster variants into M clusters, where M=10 as default. In each cluster, the variant closest to the cluster center is selected as the representative shader variant and used as one of the new seed variants for the next round of population. After each iteration, we have at most 80 representative variants. When the specified number of iterations/variants are generated, or there are no statements/expressions left to be simplified, we terminate the clustering algorithm. These representative variants are organized as the SDG and will be used as the reduced optimization space at the runtime stage.

The total number of shader variants is influenced by the complexity of shader code as well as the numbers of rules applied on it. As we reported in Tab. 1, although shaders in the *Head* and *Sibenik* demo have very different LOC(lines of code), the only used expression reduction rule results in a similar number of variants. However, even the shader in *Dragon* demo has a small number of LOC, more variants are generated by being applied with all three rules. The maximum number of selected representative shaders is controlled by the group numbers (N, M) and a maximum allowed iterations. In our test, 5-10 iterations with default group number setting are usually good enough to create a reduced optimization space. Ideally, if the reduced optimization space well approximates the original optimization space, the Pareto-optimal shaders obtained in both optimization spaces should be the same. To test it, we compare these Pareto-optimal variants of one view that are obtained in our optimization space and the full space using an offline genetic programming method [SAMWL11, WYY*14]. In Fig. 4, we visualized these Pareto-optimal variants of the *Dragon* demo (Left) and the *Warrior* demo (Right). More results can be found in the supplemental document. As shown in these results, the Pareto frontiers generated by our system are quite similar to those generated at offline.

5. Runtime Optimization Stage

Once the rendering context, e.g., the scene and uniform parameters, are available at runtime, our system seeks an automatic scheme to search the collection of representative simplified shaders for the optimal shader. Generally, the rendering and optimization tasks are simultaneously performed in different threads. In the main thread, rendering tasks are assigned to GPUs to generate images to the user. Meanwhile, all optimization tasks are computed in the working thread except some draw calls, which are executed in the main thread and coordinated between two threads.

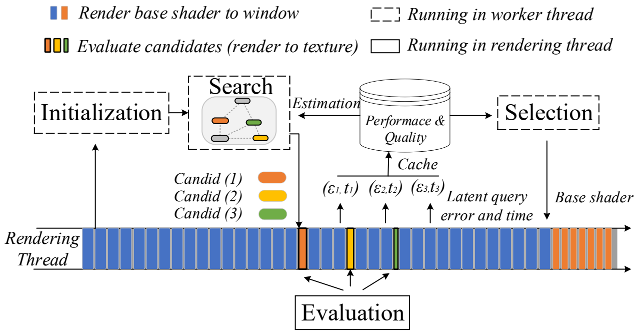


Figure 5: *One iteration of runtime optimization*

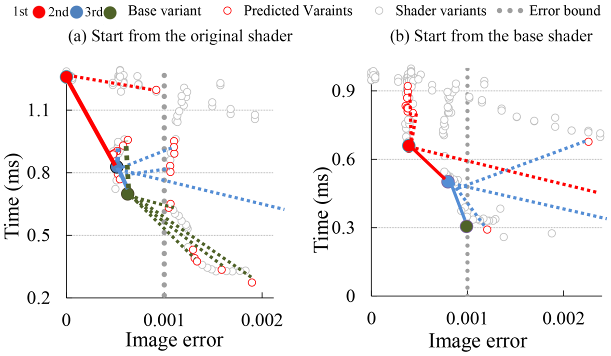


Figure 6: *A visualization of runtime optimization.*

The optimization is iteratively performed. One iteration of our runtime optimization is shown in Fig. 5. A base shader is set as the main shader generating images for the user. At the very beginning, the base shader is set as the original shader. Then, our system performs four steps in each iteration. First, an *initialization* step is used to monitor the changes in rendering context, including the scene and uniform parameters. Once the rendering context dramatically changes, a new round of optimization is triggered, the optimization moves to the *serach* step. In this step, our system searches for some candidate shaders that may potentially reduce computational cost but still retain the quality error under the error threshold. It explores optimization space to search variants, then uses proposed runtime prediction models to predict most promising variants . These candidates are actually rendered in the *evaluation* step to obtain the actual rendering error and time. These data can be used to predict shader variants' performances and qualities during *search* step. When all promising candidates are evaluated, a *selection* job is invoked to select the best candidate from all the evaluated variants to replace the base shader or to start new search in the next iteration.

We visualize two rounds of runtime optimization during *Monster* demo in Fig. 6, one starts from the original shader (Left), the other starts from the simplified base shader (Right), variants are plotted in (ε, t) space. Starting from the base shader (red dots), the system searches possible variants in the SDG (marked as red circles). Based on our proposed cost models, four most promising candidates are selected and evaluated (linked by the dashed and solid lines to current base shader). In these examples in Fig. 6, from the evaluated variants, the system selects the optimal shader variant (the blue and olive green dots) to replace the previous base shader. After several iterations, it converges to one optimal shader variant. The pseudocode of the entire runtime optimization is given in the supplemental document.

5.1. Initialization

Since our optimization is performed during runtime, the scenes and uniform parameters are all subject to changes. To detect such a change efficiently, we monitor some changes of scenes (load/unload of the scene, dramatic changes of camera and uniform parameters, etc.). Besides these drastic movements, we also periodically (e.g., 200 ms as default) compare the current rendering result with the previous result rendered by the base shader to handle incremental changes. Once the error is larger than a threshold, we regard that the optimization context changes and trigger a new round of optimization. To compare quality and performance of two shaders rendered at different times during searching the optimal shader, we use the same input data cached after the optimization starts. Specifically, we take a snapshot of the scene and all values of uniform parameters for a short period of time (e.g., 200 ms) and use this snapshot to carry out optimization during this period.

If there is no change, our optimization continues its iterations. If there is a small change, we use current base shader as the start point to launch a new round of optimization (Fig. 6(Right)). However, if a big change occurs, we immediately set the base shader to the original shader and restart the optimization. In supplemental documents, we give out the pseudo code of this step.

5.2. Searching

The optimization space is organized into the SDG with discrete nodes, therefore we formulate the searching of optimal shader variants as a discrete optimization problem. The key is to make the optimization efficient and fast. First, we propose data-driven cost models to estimate quality and performance from runtime data. While using such models to estimate shader quality and performance, rather than running the shader to measure its actual values, we can avoid actually evaluating shaders, accelerating the speed of the searching algorithm. Then, having the estimates of time and quality, we employ a parallel searching algorithm to explore the graph. Note that in this step, no actual draw calls are required, therefore it is totally performed in the working thread.

5.2.1. Data-Driven Cost Models

While we construct the SDG, we used a heuristic model in the lack of runtime context. Now, at the runtime, we can further improve our estimates by taking into account of the data that shader variants actual performed (even only a small number of variants are evaluated). We propose two data-driven cost models to estimate the quality and performance respectively.

Runtime Performance Model To better estimate the rendering time of a shader, besides the number of instructions, we now extend Eq.(4) to include the rendered scenes and the actual hardware draws into consideration. The overall performance model is designed as

$$C_{\text{total}} = N_v C_v t_v + N_f C_f t_f + N_a * t_a \tag{5}$$

where N_v is the number of vertices, N_f is the number of fragments, and t_v, t_f and t_a are the units of time to perform each piece of computation on one vertex, one fragment or the overhead, respectively. We directly query N_v and N_f from APIs of DirectX and store them in the data snapshot, which is used for this period of optimization. But, these units of time depend on different hardware and are difficult to query. Instead, we use actually evaluated time to optimize them by regression. Initially, we set them 1.0, 1.2 and 1200 as default. While more and more shader variants are evaluated, we fit these units of time to the actual data, and use them in later estimates of the performance of shader variants.

Runtime Quality Model In the preprocess stage, we propose the parameter influence to measure the differences of simplified shader variants with respect to different uniform parameters. But, the parameter influence is only a relative metric, not the actual error to let us rank variants under certain rendering context. Therefore, we need a runtime error model to convert parameter influences to errors. Our basic idea is to take advantage of the evaluated shader variants, and regard the error produced by one simplified shader variant as a combination of some 'basis' simplifications. Notice that in the SDG, one shader variant, if it is not directly simplified from the original shader, is a child of a combination of other nodes with different simplifications. If we regard the entire error also as a combination of their parent nodes, we can project and spread their errors onto its parent nodes, or gather and estimate the error from them as well.

More precisely, once we have evaluated a shader variant and obtain the error, for example e_j of variant s_j, to spread the error onto its parent nodes, we first retrieve its parents from the SDG. Then, we regard that the parameter influence vectors reflect the error contributions of parents, therefore project the error e_j to each parent as:

$$e_{i_k,new} = \frac{(\mathbf{q}_j \cdot \mathbf{q}_{i_k})}{\sum_{i_k}(\mathbf{q}_j \cdot \mathbf{q}_{i_k})} e_j \tag{6}$$

Then, the projected error is averaged by all previous computed errors on the node i_k. To accelerate the projection when the dimension of influence vector is high, we precompute and cache all dot products of parent nodes for one variant along the path to the original shader.

While predicting the error of one shader variant, s_j, it can be predicted as:

$$e_j = \sum_{i_k} (\mathbf{q}_j \cdot \mathbf{q}_{i_k}) e_{i_k} \tag{7}$$

where i_k is the index of all parent nodes in the simplification dependence graph of variant s_j, $\mathbf{q}_j$ and $\mathbf{q}_{i_k}$ are parameter influence vectors of the shader variants, s_i and its parent s_j. In some cases, not all parent nodes have been evaluated or spread error. So, for such a parent node without error, we use the error of the closest evaluated shader variant as to compute its child, where the distance is defined in the parameter influence space.

To better understand these heuristic models, we use the normalized τ-distance [Ken38] to evaluate every prediction with the real cost. The averaged τ-distances on all demos are reported in Tab. 1. The small τ-distances demonstrate that our cost models performed well in predicting runtime quality and time. In Fig. 7, we plot the τ-

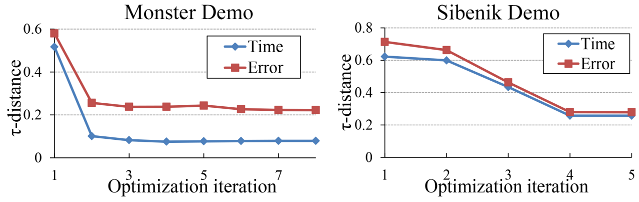


Figure 7: *The convergence of our predicted quality and performance's τ-distances.*

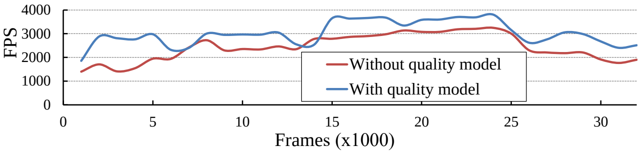


Figure 8: *Optimization with and without the quality model.*

distances of performance and quality in one round of optimization. At first several iterations, the τ-distances have high values. However, while more and more variants have been evaluated, both τ-distances drop quickly. Additionally, we also conduct experiments on the *Monster* demo to measure the impacts of the quality model in the optimization. We turn on and off it in the optimization. If the quality model is off, we only use the performance model to select candidate shader variants. The FPSs (frame per second) of two approaches are shown in Fig. 8. The results show that missing the guide of the quality model, the runtime optimization converges slower, resulting in a significant performance drop.

5.2.2. Parallel Searching

Starting from some seed nodes, the searching step aims to explore neighbors, and find most promising candidates that reduce the computation cost as well as produce errors under the error threshold. In our system, seed nodes are these variants that are Pareto-optimal variants in last iterations. To efficiently explore the optimization space, we use the parallel Best-first search [WDH88]. Specifically, given an error threshold, we maintain a list to store unexplored nodes that produce smaller quality errors than the threshold, and all nodes are sorted by performance. The quality and performance are both estimated by aforementioned cost models. To add new nodes to the list, we visit all connected nodes of seed nodes, not only children, but also including parents and siblings. After adding new nodes, we then select best candidates from the list. According to the goal of our optimization, these best candidates should be Pareto-optimal, so our system enumerates nodes in the list and selects K-best candidates with most performance gain and on the Pareto frontier. In our system, K is set to 4 as default.

At the very beginning of optimization, no shader variants have been evaluated, so neither the time or error can be predicted by cost models. In this case, we only use the heuristic performance estimate in the preprocess stage to predict the performance, and select K fastest candidates to start the optimization.

5.3. Evaluation

Once having these candidates from the searching step, we then perform the real evaluation of them to actually obtain the error and time. Such an evaluation is to insert draw calls in the main thread and read back the error and time from the pipeline. As a common practice to hide the latency between CPU submission and GPU execution, we postpone the read back of both time and error by 3 frames (Fig. 5, Evaluation). We use the performance query APIs of DirectX to measure the performance. To reduce the noise brought by the query, we average the actual measured time with the time we estimated to obtain the measured time. To get image error of a shader variant, we render the shader variant to texture and compare it with the reference image generated by the original shader. We use the compute shader [Boy08] to compare Mip-mapped values at upper levels to reduce the overhead of comparison. This optimization works well in practice. In the supplemental document, we show this MIP-map error is accurate enough.

5.4. Selection

After the evaluation step, we obtain the actual costs of the quality and performance of several shader candidates. Multiple candidates may be Pareto optimal, so our system selects the one based on the following two criteria. First, the shader variant should have the biggest performance gain but is still under the error threshold. Second, the shader variant should not produce much error comparing to the base shader, i.e., the error is less than a threshold. The second criterion is to avoid the obvious sudden changes of the image when replacing the base shader. Once one newly evaluated shader variant is selected, we then replace the base shader with it. But, if no better variant exists, we then iterate back to the searching step to find other shader candidates. If in several iterations, the base shader variant has not been updated. We then regard the optimization converges. The number of blank iterations used to detect the convergence is set to 3 by default.

6. Results

We use a desktop PC with an Intel CoreTM i7 3770 CPU and two different graphics cards (an NVIDIA GeForce GTX 680 with 2GB RAM and a GTX 980Ti with 4GB RAM) to generate the results in this paper. Except for the Sibenik demo, all images are rendered at a resolution of 1920×1080. All shaders in six demos are optimized during runtime with rendering context changes of uniform parameters. Please refer to the supplemental video to see the entire runtime optimization process. To avoid the cost of recording videos, we use an external video capture card to directly record the graphics card output through DVI at 60 FPS. Note that due to the lower recorded FPS, the captured video may exhibit some blurring effects. To compare our method with others, we record the user input and then play back these inputs in different methods.

6.1. Shaders and Demos

Six demos (Fig. 1, Fig. 9) are used to evaluate our system. Configurations and statistics of these demos are listed in Tab. 1.

Dragon demo: The *Dragon* model is rendered with a shader that has four octaves of Perlin noise and Phong BRDF shading model.

Monster demo: The *Monster* model has both albedo and specular textures. The glossy reflections from environment lighting are computed by the GPU importance sampling shader. 40 samples are used in the original shader.

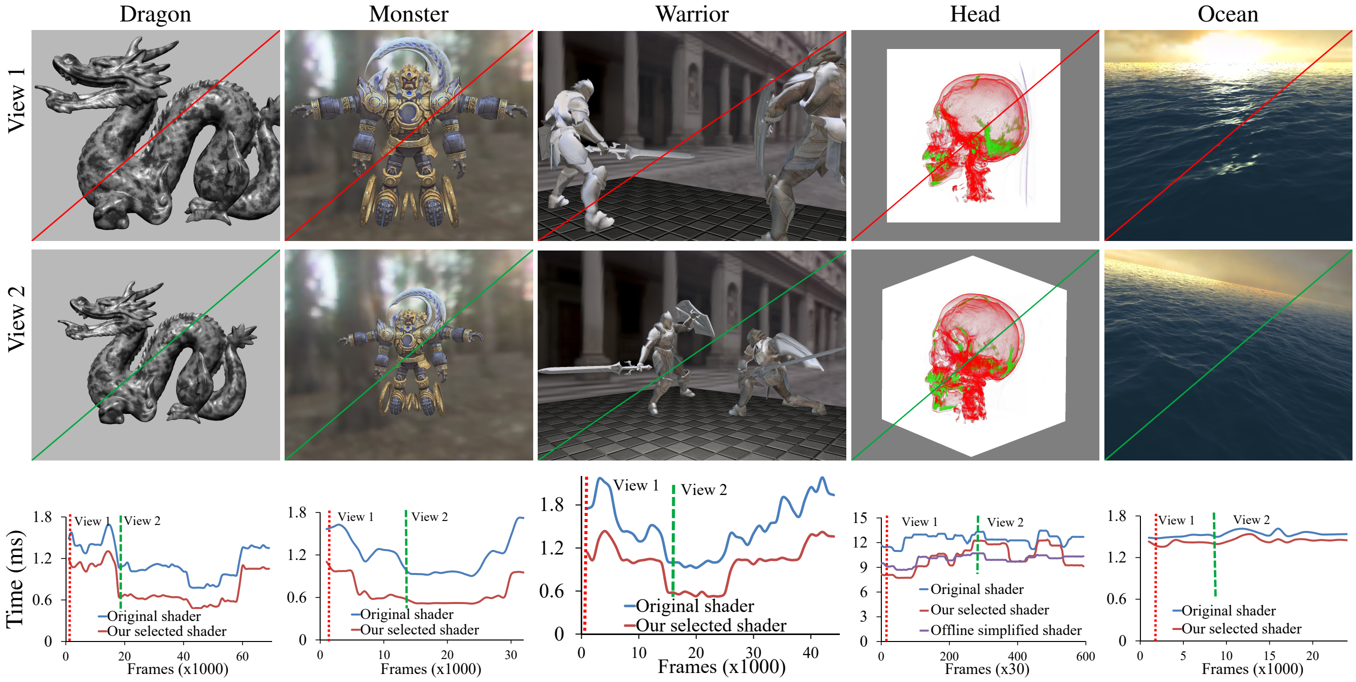


Figure 9: *Results of different demos, both original shader (Upper left) and selected simplified shader (Bottom right) at two views are presented.*

	Dragon	Monster	Warrior	Head	Sibenik	Ocean
Scene						
Triangles	130K	14K	16K	12	2	1400K
Vertices	75K	8K	10K	8	4	475K
LOC	94	117		50	175	88
Inst. (Pixel)	6K	12K		202K	153K	1K
Preprocess						
SDG Node#	324	835		290	285	460
$Time_{SDG}$(s)	1.5	3.9		0.4	1.0	4.2
$Time_{Compile}$(s)	27.2	64		7.5	424	19
Shader Mem(mb)	1.4	3.4		0.59	1.15	2.2
Runtime						
ε_{max}	0.003	0.002	0.004	0.001	0.014	0.014
$Overhead_{CPU}$	0.2%	1%	3%	0.4%	0.3%	0.3%
$Overhead_{GPU}$	0.17%	0.4%	1.15%	1.1%	1.3%	0.3%
Seq. time(s)	87	32	67	242	121	39
Saved time(s)	29	15.6	23	54	52	36
Saved time(%)	33%	49%	34%	22.3%	43%	7.6%
τ-dist. of time	0.12	0.08	0.39	0.06	0.25	0.29
τ-dist. of error	0.20	0.22	0.22	0.15	0.27	0.28

Table 1: *Configuration and statistics of demos, from top to bottom: scene configuration, preprocess statistics, and runtime statistics.*

Warrior demo: We apply the same GPU importance sampling shader in the *Warrior* demo. Two intensely fighting warrior models are included.

Head demo: We adopt the ray marching volume rendering shader to a 3D volume head [DCH88]. The head data is stored in a 256 × 256 × 225 3D texture. In the original fragment shader, a total of 800 steps of each ray are marched from the front faces to the back faces of the bounding box. In each step, the transferred color and transparency are computed from an intensity transfer texture and then accumulated to the final color.

Sibenik demo: We adopt the HBAO [BSD08] shader and render a 1280 × 720 AO image in four passes. They are the depth buffer generation, full-resolution AO with 24 × 24 samples per pixel, along with two bilateral filter passes. Given that the other passes are simple, our system only simplifies the most costly shader on the second pass: the AO sampling pass. We still use the final result to compute image errors instead of the output of AO pass.

Ocean demo: This demo contains three major steps: inverse FFT based on spectral algorithm [Tes01], Quad-tree tile-based LODing, and ocean shading. We perform an inverse FFT of resolution 128 × 128 on GPU to generate height and normal map. The generated LOD meshes lead to lots of draw calls, about 1.5 million triangles are submitted. While the ocean shading is very simple, constant water body color combine with reflected high specular sunrise environment map, noise texture is used to perturb reflections.

6.2. Simplification Results

In each demo, we interact with the model and leave the simplification system running in the background. We record a sequence of frames and show two example views in Fig. 1 and 9. In addition, we plot performance comparisons between original shader and our selected simplified shaders. For the *Head* and *Sibenik* demos, we further present the performance of the offline simplified shader. The instant FPS of demos can be seen in the supplemental video, we average every 1000 frames (30 frames in the *Head* demo) to obtain the reported time in Fig. 1 and 9. Results show that the proposed system is well adapted to the changes of runtime context and successfully improves the performance. In the following, we provide more details and analyses on the optimizations performed by our system in these demos.

In the *Dragon*, *Monster* and *Warrior* demos, we rotate the model, along with zooming the camera in and out. Generally, the more we zoom out the camera, the simpler shader we get. However, our system performs simplifications differently in these demos. For example, at view 1 of the *Dragon* demo, one of four octaves in the shader is reduced (which saves about 0.2 ms). At view 2, our system reduces two octaves and also moves the noise texture and diffuse lighting to the vertex stage (which saves about 0.5 ms). While in the *Monster* demo, our system tends to transform some lighting computation to tessellation shader with two levels of tessellations, resulting in a 1.6× speedup. When the camera is further zoomed out, our system chooses a more aggressively simplified shader: all lighting sampling is moved to the vertex stage and without any tessellation, which brings 1.8× speedup. The *Warrior* demo shows the capability of our system to handle drastic changes of scenes and cameras. While these two warriors are fighting, our system constantly responds to runtime changes of scenes, and achieves similar performance improvements to those in the *Dragon* and *Monster* demos.

The *Sibenik* and *Head* demos only have fragment shaders executed in screen space. These screen space based shaders are important, since they have been widely used in the deferred shading, but are more challenging, because they are usually more costly than those in forward rendering. Our system also performs well in these two demos. As shown in Fig. 9 (Head, View 1), just after first few frames (total 20 shader variants are evaluated), our system finds an optimal shader skipping the last 30% of marching steps but still retains good rendering quality. This simplified shader brings 1.4× speedup. While we rotate the camera about 45 degrees (View 2) requiring more steps to march through the diagonal of the volume, our system automatically reverts to select a shader with more marching steps, which leads to an increase in rendering cost (still faster than original shader), but ensures quality. Fig. 1 shows results of the *Sibenik* demo. For an HBAO shader, our expression reduction rules mainly result in two simplifications that reduce the number of sample rays and the searching steps at each sample. At View 1, the number of searching steps reduces to half, since most occlusion can be found by only a few steps. At View 2, while the floor occupies a large portion of scene space, our system tends to select shaders with fewer ray samples (since most rays are unoccluded), but keep the searching steps in one ray sample (so as to find occluders in some distances). At View 3, both the number of ray samples and searching steps are reduced a little bit to adapt the visible scene at this view.

The *Ocean* demo shows an extreme case that a scene with a large number of triangles but with a very cheap shader. The total time spent on executing original shaders (both vertex and fragment shaders) occupy less than one-fourth of the total rendering time. However, our system is still able to squeeze a little bit speedup. In the Fig. 9 (Ocean, View 1), our system moves most computations from pixel to vertex. Additionally, at View 2, while the reflected sun lights become less noticeable, our system further removes some specular perturbations to save time. These optimizations saved about a half of the shader execution time (0.12∼0.18 ms), resulting in about 5% increase in FPS.

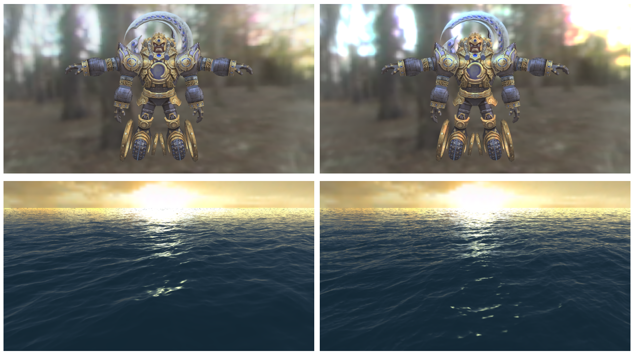


Figure 10: *Two example applications for the saved rendering cost from runtime simplification. For each row, left shows the original shading results, while the right show results of simplified shader with additional bloom effect (Top) and finer (256×256) inverse FFT simulation (Bottom), but render them cost less and similar rendering time respectively.*

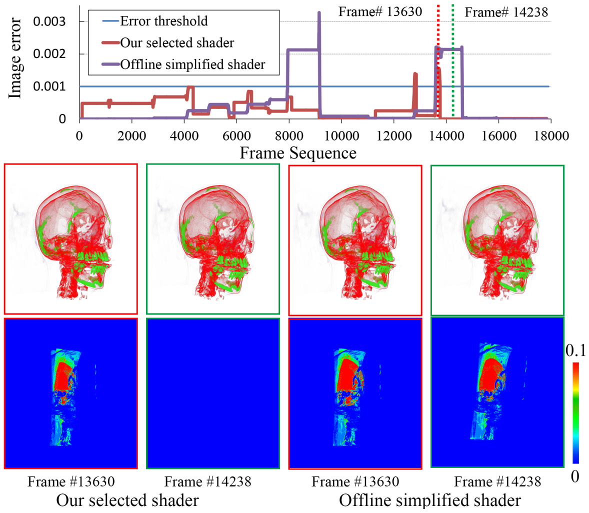


Figure 11: *Comparison of optimization during runtime and offline optimization. Top: errors among animations. Bottom: Rendered images and difference images using the runtime selected simplified shaders (Left) and offline simplified shader (Right) at two frames.*

The saved GPU execution time by our runtime simplification system could be used to dynamically invoke other tasks, thereby improving the rendering applications at different aspects. In Fig. 10, we show two examples. In the *Monster* demo, we use the saved time from a simplified shader to enable the bloom effect, which highlights the lighting effects and provides better visual effects (Top right). For the *Ocean* demo, the saved time from the simplified shader gives more time budget for GPU simulation. Since the inverse FFT is quite cheap, we successfully increase the resolution of inverse FFT to 4 times (from 128×128 to 256×256) using saved GPU time. By repeatedly using these inverse FFT map on the ocean, we obtain more details on the ocean surface (Bottom right), but render it at the similar rendering time.

6.3. Comparison with Offline Simplified Shader

In previous offline optimization methods [Pel05, WYY*14, HFTF15], all possible uniform parameters are enumerated to generate a general context and errors are averaged while optimizing. We compare our runtime selected simplified shaders with the shaders optimized by the aforementioned offline strategy. There were 30 different views in or out of the sequence of the demo used as the example contexts to simplify shaders in the *Head* and *Sibenik* demo. In the *Head* demo, we select the optimal one with a similar average performance of our selected simplified shader, and plot its rendering time and errors at Fig. 11(Top). We select two views and visualize the error in Fig. 11(Bottom). The figures show that our system automatically detects the context changes, and quickly optimizes a new optimal shader to avoid further artifacts. However, in contrast, the offline simplified shader fails at adapting to these changes and continues producing these significant artifacts.

An offline simplified shader was also selected in the *Sibenik* demo having a similar averaged error to the error threshold, and the offline simplified shader was compared with simplified shader selected by our runtime system. The results of the comparison are shown in Fig. 1. It can be seen that at some frames, e.g., from 3200 to 4800, the offline shader produces the optimal results as ours. However, at most of the other frames, our runtime selected shader has lower rendering cost and produces tolerable errors, especially at View 2, where the runtime simplified shader is about 1.7× faster than the offline simplified shader. In addition, at View 3, the offline simplified shader is not only slower but also produces much larger error than our runtime selected shader, demonstrating the adaptability of our runtime scheme.

6.4. Overheads and Artifacts

Similar to many other auto-tuning works, our method has a common limitation that the runtime computation may bring some overheads. Mainly, our system has two kinds of overheads. One overhead is the memory that stores the representative shader variants and some cached data, and the other overhead is the time spent in the preprocessing and at runtime. We list the statistics of preprocessing time, memory, and the runtime computation time in Tab. 1.

Most of the time spent in the preprocessing involves compiling shader variants, varying from tens to hundreds of seconds (Row $Time_{compile}$), whereas clustering shaders and generating the SDG only take seconds (Row $Time_{SDG}$). However, this step is only one-time processing, and the processed shaders can be applied to different objects and on different hardware platforms. In our demos, these preprocessed shader variants and the SDG occupy few MBs, we suggest applying this runtime optimization only to those performance critical ones. Compared with resources such as complex geometry models and high-resolution textures, such a memory cost is quite affordable for many applications.

Two kinds of computational resources are required for our runtime optimization: CPU and GPU. The overall overheads of extra CPU and GPU time for all of our demos are measured and reported in percentage by comparing the total sequence time (Row $Overhead_{CPU}$,$Overhead_{GPU}$). The reported GPU overheads include all performing draw calls to generate ground truth and evaluate candidate shader variants, image down-sampling and Mip-map error calculation. Our runtime system usually takes 5-8 iterations to complete one round of optimization, in which 20 to 30 variants are evaluated. The total time spent to evaluate shader variants varies with demos, e.g., from less than 15 ms in *Monster* demo to a maximum of 80 ms in the *Sibenik* demo. In the cases such as *Dragon* and *Monster* demos, the newly optimized shader variant is much faster than the original shader. Usually only after one iteration, the performance gain has already compensated the cost of optimization iterations. Therefore, punctual slowing of instant FPS was not observed in these two demos. However, when the original shader was very costly, instant latencies from the optimization may bring some impacts on the FPSs, such as at frame 9300 in the *Sibenik* (Fig. 1) demo and at frame 13800 in the *Head* demo (Fig. 9). Please refer to these frames in the supplementary video for the visual and instant FPS impacts brought by these latencies. But, in most cases, the overheads of our optimization are negligible.

Although the instant overhead for one round of optimization is not neglectable in some demos, the overall overhead of both CPU and GPU is still low, e.g., usually around or less than 1%. The exception is the *Warrior* demo because of the more rounds of optimization triggered by the animation, and the overall CPU and GPU overhead in this demo is only 3% and 1.15% respectively. There are three reasons the overhead was lowered. First, cost models are employed to predict performance and quality, significantly reducing the cost of runtime evaluations. Second, the optimization does not necessarily need to be frequently performed. Only when the scene has a large change and the system detects that the visual error of the current shader is larger than the threshold, the optimization is actually launched. Third, the optimization does not need to start from scratch. In many cases, the previous simplified shader is a good candidate to start (Fig. 6(Right)), leading to fast converge.

Our optimization is progressively conducted at runtime. Therefore, updating the optimal shaders may occasionally cause inconsistent artifacts. Sec. 5.4 introduced a criterion to guide the selection of visually similar shader variants. However, this strategy cannot eliminate all artifacts. Another source of inconsistent artifacts is from the assumption of context coherence. As discussed in Sec. 6.3, all monitoring and snapshotting are performed at a period. If the assumption is violated in this period, artifacts may be produced before a new round of optimization is performed.

7. Conclusion

This paper presented a runtime shader simplification system, and to make it practical, a two-stage algorithm was designed, the preprocess and the runtime optimization. In the preprocess, the SDG was proposed to represent simplified shaders, as well as a specific clustering algorithm to reduce the optimization space. Two heuristic metrics were introduced to estimate quality and performance. At runtime, to efficiently search optimal shader variants, we employ a parallel discrete optimization algorithm and explore in the reduced optimization space. Results show that our system outperforms the previous offline simplification method and can adapt to different scenes with respect to runtime contexts. However, in the current implementation, we did not consider optimizing different levels of shaders for multiple instancing objects. To support different levels of optimality for objects with one shader, several interesting optimizations might exist, such as grouping similar objects and sharing predictions inside the group. We would regard them as important future works.

Besides, our system also has several directions worthy of further exploration. First, in the current implementation, the shaders are compiled at the preprocess stage. The time to compile hundreds or thousands of shader variants is still long to be fit in the runtime optimization. Therefore, it will be an interesting work to combine the pre-compiling with runtime compiling to enable greater adaptivity of runtime shader simplification. Second, we use the queried information of DirectX pipeline to predict the performance, but with more pipeline information, we may provide more accurate estimation about the performance of shaders and scenes. Third, we propose the parameter influence to estimate the quality change of simplified shader. But how the visual error is related to all kinds of input parameters is still an interesting future work.

Acknowledgements

We would like to thank all reviewers for their insightful comments. This research was partially funded by National Key R&D Program of China (No. 2017YFB1002605), NSFC (No. 61472350), Zhejiang Provincial NSFC (No. LR18F020002) and the Fundamental Research Funds for the Central Universities (No. 2017FZA5012).

References

[AKV*14] Ansel J., Kamil S., Veeramachaneni K., Ragan-Kelley J., Bosboom J., O'Reilly U.-M., Amarasinghe S.: Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on* (2014), IEEE, pp. 303–315. 2

[BHT*10] Baskaran M. M., Hartono A., Tavarageri S., Henretty T., Ramanujam J., Sadayappan P.: Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), ACM, pp. 200–209. 2

[Boy08] Boyd C.: The directx 11 compute shader. *shading Course SIGGRAPH* (2008). 8

[BSD08] Bavoil L., Sainz M., Dimitrov R.: Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks* (2008), ACM, p. 22. 9

[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

[DCH88] Drebin R. A., Carpenter L., Hanrahan P.: Volume rendering. In *Computer Graphics (Proceedings of SIGGRAPH)* (1988), vol. 22, ACM, pp. 65–74. 9

[FJ05] Frigo M., Johnson S. G.: The design and implementation of fftw3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. 2

[FOW87] Ferrante J., Ottenstein K. J., Warren J. D.: The program dependence graph and its use in optimization. *ACM Trans. on Program. Lang. and System. (TOPLAS) 9*, 3 (1987), 319–349. 4

[HFF16] He Y., Foley T., Fatahalian K.: A system for rapid exploration of shader optimization choices. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 35*, 4 (2016). 3

[HFH*17] He Y., Foley T., Hofstee T., Long H., Fatahalian K.: Shader components: modular and high performance shader development. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 36*, 4 (2017), 100. 3

[HFTF15] He Y., Foley T., Tatarchuk N., Fatahalian K.: A system for rapid, automatic shader level-of-detail. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA) 34*, 6 (2015), 187. 2, 3, 4, 5, 11

[JTD*12] Jordan H., Thoman P., Durillo J. J., Pellegrini S., Gschwandtner P., Fahringer T., Moritsch H.: A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (2012), IEEE, pp. 1–12. 2

[Ken38] Kendall M. G.: A new measure of rank correlation. *Biometrika 30*, 1/2 (1938), 81–93. 7

[NTCS10] Naono K., Teranishi K., Cavazos J., Suda R.: *Software automatic tuning: from concepts to state-of-the-art results*. Springer Science & Business Media, 2010. 2

[OKS03] Olano M., Kuehne B., Simmons M.: Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 7–14. 2

[Pel05] Pellacini F.: User-configurable automatic shader simplification. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* (2005), vol. 24, ACM, pp. 445–452. 2, 4, 11

[SALY*08] Sitthi-Amorn P., Lawrence J., Yang L., Sander P. V., Nehab D., Xi J.: Automated reprojection-based pixel shader optimization. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)* (2008), vol. 27, ACM, p. 127. 2, 3, 4

[SAMWL11] Sitthi-Amorn P., Modly N., Weimer W., Lawrence J.: Genetic programming for shader simplification. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA) 30*, 6 (2011), 152. 2, 3, 4, 5, 6

[SBR*15] Sampson A., Baixo A., Ransford B., Moreau T., Yip J., Ceze L., Oskin M.: Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01 1* (2015). 2

[SJW07] Scherzer D., Jeschke S., Wimmer M.: Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (2007), Eurographics Association, pp. 45–50. 3

[Tes01] Tessendorf J.: Simulating ocean water. *Simulating Nature: Realistic and Interactive Techniques. SIGGRAPH 1*, 2 (2001), 5. 9

[TH11] Tiwari A., Hollingsworth J. K.: Online adaptive code generation and tuning. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 879–892. 2

[VDY05] Vuduc R., Demmel J. W., Yelick K. A.: Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series* (2005), vol. 16, IOP Publishing, p. 521. 2

[WD98] Whaley R. C., Dongarra J. J.: Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing* (1998), IEEE Computer Society, pp. 1–27. 2

[WDH88] Whitlock D., Dey P., Hyatt R.: A parallel best-first search. In *Proceedings of the 1988 ACM Sixteenth Annual Conference on Computer Science* (1988), CSC '88, pp. 735–. 8

[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 (Proceedings of SIGGRAPH ASIA) 33*, 6 (2014), 226. 2, 3, 4, 5, 6, 11

[XJJP01] Xiong J., Johnson J., Johnson R., Padua D.: Spl: A language and compiler for dsp algorithms. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 298–308. 2

[YB18] Yang Y., Barnes C.: Approximate program smoothing using mean-variance statistics, with application to procedural shader bandlimiting. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 443–454. 3