# Automatic Shader Simplification using Surface Signal Approximation

Rui Wang* Xianjin Yang* Yazhen Yuan* Wei Chen* Kavita Bala† Hujun Bao*

*State Key Lab of CAD&CG, Zhejiang University †Cornell University

## Abstract

In this paper, we present a new automatic shader simplification method using surface signal approximation. We regard the entire multi-stage rendering pipeline as a process that generates signals on surfaces, and we formulate the simplification of the fragment shader as a global simplification problem across multi-shader stages. Three new shader simplification rules are proposed to solve the problem. First, the code transformation rule transforms fragment shader code to other shader stages in order to redistribute computations on pixels up to the level of geometry primitives. Second, the surface-wise approximation rule uses high-order polynomial basis functions on surfaces to approximate pixel-wise computations in the fragment shader. These approximations are pre-cached and simplify computations at runtime. Third, the surface subdivision rule tessellates surfaces into smaller patches. It combines with the previous two rules to approximate pixel-wise signals at different levels of tessellations with different computation times and visual errors. To evaluate simplified shaders using these simplification rules, we introduce a new cost model that includes the visual quality, rendering time and memory consumption. With these simplification rules and the cost model, we present an integrated shader simplification algorithm that is capable of automatically generating variants of simplified shaders and selecting a sequence of preferable shaders. Results show that the sequence of selected simplified shaders balance performance, accuracy and memory consumption well.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Rendering;

**Keywords:** GPU shader, real-time rendering, shader simplification, surface signal approximation

## 1 Introduction

GPU shaders play a very important role in computer graphics [Segal et al. 2013; Microsoft 2013a]. Time-critical applications like video games, real-time shading, and high-fidelity visualizations heavily rely on high-performance shader computations. In the latest GPU pipeline, there are five shaders acting in three rendering stages [Kessenich et al. 2013; Microsoft 2013b]. The vertex shader processes vertices in the vertex shader stage; the geometry shader and the tessellation shaders compute and output geometry primitives in the geometry-processing shader stage; and the fragment shader inputs interpolated geometry attributes and outputs shading values of pixels in the fragment shader stage. All of these shaders

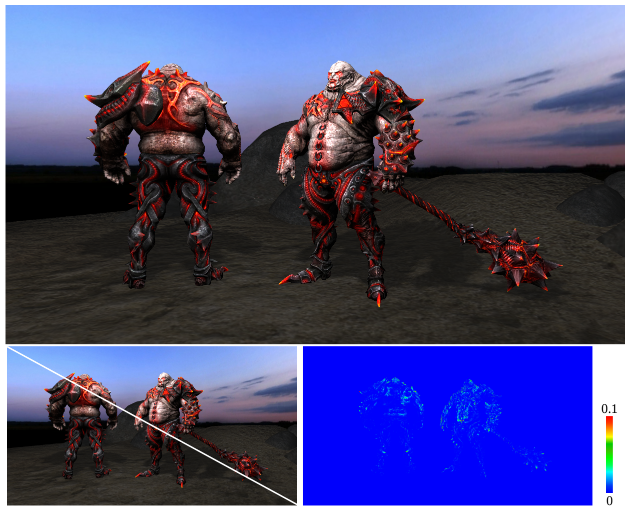


**Figure 1:** *(Top) one result using the simplified shader generated by our approach. (Bottom Left) side by side comparison with the image rendered by the original shader. (Bottom Right) the error image to visualize the difference. Compared with the original fragment shader, our simplified shaders raise the FPS from 61 to 229.*

are fully programmable, reproducible and reusable, allowing flexible personalization of real-time rendering effects.

However, the quality of shaders greatly depends on the experience of shader programmers. Even though there are some rules and toolkits for interactive modulation and optimization of shader programs, the entire process is time-consuming, tedious and sometimes inefficient. In general, the most time-consuming part is the shading computation within the fragment shader. Thus, much efforts have been made to automate the simplification of fragment shaders. Pellacini [2005] proposed a user-configurable shader simplification method for pixel-wise procedural modeling. Nehab et al. [2007] and Sitthi-amorn et al. [2008] presented a reprojection based scheme to optimize the shading of pixels. Most recently, Sitthi-amorn et al. [2011] used Genetic Programming to fully automate the simplification process. However, all of these studies only took simplifications within the fragment shader stage. If we take a broader view of the entire rendering process, the fragment shader is only one stage of the entire rendering pipeline, and other shaders are also of great importance in computing shading values. Foley and Hanrahan [2011] presented a modular and composable shading language to cut across multiple pipeline stages. But, their goal is not the simplification of shaders.

In this paper, we introduce a new approach to represent and approximate shader computations. We regard the shader computations as the generation, modulation and combination of signals on the surface. In this way, the computations in the fragment shader can be approximated and simplified by surface signal approximations, and thereby be or transformed to other shaders. This leads to a novel shader simplification scheme that seeks to "*bake*" shader computations on the surface from the point of view of surface signal processing. Figure 1 shows one of our results applying this simplification scheme, where the simplified shaders raise the FPS from 61 to 229 but still retain good visual quality when compared

*e-mail:rwang, bao@cad.zju.edu.cn

to the original shaders. To achieve this, three new shader simplification rules are proposed: a) transforming the shader computations from the fragment shader to the geometry-processing shaders to redistribute these computations; b) simplifying the shader computations by means of surface signal approximations and caching them on the surface; c) subdividing surfaces into smaller patches to provide different qualities of signal approximations. All these rules bring new challenges for automatic shader simplifications: allowing across-shader simplification greatly complicates the simplification process; leveraging surface signal simplifications requires a well-designed strategy for signal representation, modeling and compensation; subdividing surfaces at different levels greatly enlarges the exploration space of simplified shaders. Additionally, besides the rendering time and the visual error, incorporating the approximation on surfaces introduces a new cost: the memory consumption. This paper presents a new solution to address all of these problems with the following contributions:

- A new shader simplification scheme that allows for re-organization of fragment shader code among different shader stages by means of a novel code transformation rule;
- A surface signal approximation rule that replaces time-consuming shading computations with signal approximations of high-order polynomials on surfaces;
- A surface subdivision simplification rule that incorporates programmable tessellators to balance approximation quality and tessellation of surfaces.
- An integrated shader simplification approach that balances performance, accuracy and memory cost by leveraging the code transformation, the surface signal approximation and the surface subdivision rules.

## 2 Related Work

**Shader Simplification** The pioneering work of shader simplification [Olano et al. 2003] was for procedural fragment shaders. It gains significant speedup by replacing texture fetches with less expensive operations. Then, a more general algorithm proposed by Pellacini [2005] automatically generates a sequence of simplified fragment shaders based on the error analysis of a set of expression rules. Thereafter, to adapt to more changes in the input, Nehab et al. [2007] proposed a reprojection-based shader optimization method that uses a screen buffer to cache optimized values. This method was extended to automate the use of data reprojection as a general and practical tool for optimizing procedural shaders [Sitthi-amorn et al. 2008]. More recently, a Genetic Programming based shader simplification scheme was proposed to select optimal shaders that balance between performance and image quality [Sitthi-Amorn et al. 2011]. Similarly, our approach leverages code analysis techniques, but distinguishes itself from previous approaches in that ours seeks to not only optimize computational operations in the fragment shader, but also distribute computations across different stages of shaders. Our method uses additional buffers to store approximated surface signals, which bears some similarity with those cache-based reprojection methods [Nehab et al. 2007; Sitthi-amorn et al. 2008]. However, the representation and the way we use these buffers are totally different.

Foley and Hanrahan [2011] presented a modular and composable shading language, Spark. It allows users to define shading effect across different programmable stages by means of a global optimization. However, the optimization used in their method is designed to eliminate dead-code and make preparations to generate HLSL shaders. Our shader simplification that generates a sequence of optimized shaders is beyond their optimization scope.

**Surface Signal Approximations** Generating the visual appearance of a surface is a process of signal generation, modeling and reformulation. Many types of surface signals have been used: colors, texture maps, bump maps, displacement maps, bidirectional texture maps, etc. [Akenine-Möller et al. 2008]. To represent lighting signals on a surface, either triangle meshes are adapted to fit the signal [Hanrahan et al. 1991; Hoppe 1996] or the signals are represented with nonlinear basis functions [Zatz 1993; Lehtinen et al. 2008; Sloan et al. 2003]. Remeshing is impractical for shader optimization, so we choose to approximate the surface signals with basis functions. We specifically choose Bézier functions because of their wide usage in graphics.

Recently, Kavan et al. [2011] proposed a vertex-based surface signal approximation technique, called vertex-baking. By taking linear approximations of the ambient occlusion on surfaces and storing them on vertices, their method is able to consume less memory and achieve better performance than that directly using ambient occlusion maps. Alternatively, our method provides non-linear approximations on surface signals, and targets a different problem: the shader simplification.

**Code Analysis** In our paper, we use several code analysis techniques to parse shaders. Abstract Syntax Trees (ASTs) [Muchnick 1997] are used to construct the syntactic structure of shader code, which has been widely used in previous shader simplification approaches [Pellacini 2005; Sitthi-amorn et al. 2008; Sitthi-Amorn et al. 2011]. Program Dependence Graphs (PDGs) [Ferrante et al. 1987] that encode the data and control dependencies of shader code are used to construct program slices [Weiser 1984] of shader code. The dependencies of code have been used in specialization shaders [Guenter et al. 1995]. Unlike their approach, our focus is on automatic shader simplification.

## 3 Overview

### 3.1 Global Shader Simplification Problem

Shaders are programs designed to compute shading on geometry primitives. In the latest graphic rendering pipeline, five shaders at different stages are used in an integrated rendering pipeline: the vertex shader, the geometry shader, two tessellation shaders and the fragment shader [Kessenich et al. 2013; Microsoft 2013b]. For simplicity, the set of shaders used in a rendering pipeline is called *a shader configuration*.

We represent the entire rendering pipeline as a function $f$, which takes a set of geometry primitives with attributes, and generates shading values for each pixel:

$$f((x,y),\mathbf{v}) = p \circ r(g \circ h(\mathbf{v}),(x,y)) \tag{1}$$

where $(x,y)$ is the screen position of one pixel, $\mathbf{v}$ is the set of geometry primitives, $p$ denotes the fragment shader, $g$ denotes shaders in the geometry shader stage, $h$ is the vertex shader, and $r$ is a fixed rasterization stage that converts geometry primitives into pixels with interpolated geometry attributes. Without loss of generality, in the following text we assume that the geometry primitives processed by shaders are triangles,

The goal of global shader simplification is to generate a sequence of shader configurations, $\{\tilde{p}_k,\ \tilde{g}_k,\ \tilde{h}_k, k = 1,2,3,...\}$, such that each triple of $\{\tilde{p}_k,\ \tilde{g}_k,\ \tilde{h}_k\}$ produces a rendered image with different rendering quality and different costs, such as different time consumption and memory consumption. In particular, the rendering quality of each optimized shader configuration is measured by the screen-space color difference:

$$e_f = \int_I \parallel p \circ r(g \circ v(\mathbf{v}), (x,y)) - \tilde{p} \circ r(\tilde{g} \circ \tilde{v}(\mathbf{v}), (x,y)) \parallel dxdy \tag{2}$$

where $I$ denotes the entire screen space and $\parallel \cdot \parallel$ is the $L^2$ norm of pixelwise color differences. Similar to previous approaches [Pellacini 2005; Sitthi-Amorn et al. 2011], the difference metric can be employed subject to additional uniform parameters, such as a set of lighting and camera positions or a sequence of input frames, $\mathbf{u}$, distributed on the domain $U$:

$$e_f = \int_U \int_I \parallel p \circ r(g \circ v(\mathbf{v}), (x,y)) - \tilde{p} \circ r(\tilde{g} \circ \tilde{v}(\mathbf{v}), (x,y)) \parallel dxdyd\mathbf{u} \tag{3}$$

## 3.2 Our Solution

Our shader simplification scheme is motivated by the following observation: in general, the entire rendering process can be regarded as generating signals of the underlying surface, and different shaders yield various surface signals. Thus, the simplification of shaders can be regarded as an approximation of the produced surface signals. Specifically, the screen position of one pixel, $(x, y)$, corresponds to a local point $(s_i, t_i)$ on a certain object-space triangle $z_i$: $(s_i, t_i) = r^{-1}(x, y)$, where $r^{-1}$ denotes the inverse rasterization function, and $(s_i, t_i)$ denotes the barycentric coordinates in $z_i$. In this way, the shading function on the image $f$ can be approximated by summed surface signals defined on the triangle set of the scene, $\{z_i\}$, as:

$$\begin{aligned} f(x,y) &= \sum_i^M f_{z_i}(r^{-1}(x,y)) = \sum_i^M f_{z_i}(s_i,t_i) \quad (4) \\ f_{z_i}(s_i,t_i) &\approx \sum_j \alpha_{ij} b_j(s_i,t_i) \end{aligned}$$

where $M$ is the number of triangles, $f_{z_i}$ is the approximation function only defined on the triangle $z_i$, $\{b_i\}$ are basis functions and $\{\alpha_{ij}\}$ are weights.

This equation links the fragment shader stage with two other shader stages, the vertex-shader stage and the geometry-shader stage. By approximating signals on the rendered image (defined in screen space) by surface signals (defined on surface), we are able to obtain a sequence of surface signal approximations with different errors. This introduces a new strategy for shader simplification: optimizing shaders by approximating them on the surface, and redistributing computations across multiple shader stages. Given the fact that modern fragment shaders usually dominate the computational budget per frame [Sitthi-amorn et al. 2008], in this paper, we only focus on the simplification of fragment shaders. Below, we describe three rules following this strategy, and the cost model used in our approach.

### 3.2.1 Simplification Rules

**Transform Code between Shaders.** One basic rule is to cut across the boundaries between the fragment shader stage and two other shader stages, and allow for transforming code between them. This rule converts the pixel-wise computations to a combination of computations on vertices and interpolations on pixels. In this sense, the code transformation can be regarded as using a suite of linear basis functions to approximate surface signals. For a triangle $z$, it can be represented as:

$$\begin{aligned} f_z(s,t) &\approx \sum_j^3 b_j(s,t)\tilde{f}(v_j^z) \quad (5) \\ &= \tilde{f}(v_0^z)s + \tilde{f}(v_1^z)t + \tilde{f}(v_2^z)(1-s-t) \end{aligned}$$

where $\{v_j^z, j = 0, 1, 2\}$ are vertices of $z$, and $\tilde{f}$ is the function similar to $f$ but computed at vertices instead of pixels. When the object-space complexity is smaller than the screen-space complexity (usually true in many real-time rendering applications), redistributing computations from pixels to vertices is effective in reducing the computational consumption.

**Approximate Shader Functions on Surface.** The second rule is to approximate shader functions by non-linear basis functions defined on surfaces, i.e., Bézier triangles in this paper. This yields:

$$f_z(s,t) \approx \sum_{i+j+l=n}^{n} B_{ijl}^n(s,t)c_{ijl} \tag{6}$$

where $B_{ijl}^n(s,t)$ are the Bernstein polynomials defined on triangles: $B_{ijl}^n(s,t) = \frac{n!}{i!j!l!}s^i t^j (1-s-t)^l, i+j+l = n$, and $\{c_{ijl}\}$ are control points. It implies that the original surface signal on a triangle $z_i$ can be approximated by a set of Bézier functions with several control points, $c_{ijl}$. With the form of the error metric in Eq.(3), we can compute these control points by minimizing the error defined on the entire surface of object as:

$$\min_{G_{ijl}} \int_{\mathbf{T}} \parallel f(s,t) - \tilde{f}(s,t)) \parallel_2 d\mu(s,t) \tag{7}$$

where $\mathbf{T}$ denotes the entire triangle set, and $\mu(s,t)$ denotes the associated surface measure. By storing these control points, we are able to simplify the runtime computations in shaders by interpolating pre-cached approximations on surface.

**Subdivide Surface.** In Eq.(4), the original signal is approximated by basis functions defined on discrete domains, i.e., piecewise triangles. This enables an alternative way to improve the approximation by partitioning the discrete domain into a denser tessellation. In our application, this increases the number of triangles. With the geometry shader or the tessellation shaders, new geometry primitives (vertices or triangles) can be generated in the rendering pipeline at runtime, and thereby be combined with previous two simplification rules to provide better approximations. Therefore, the code transformation approximation can be further represented as:

$$f_z(x,y) \approx \sum_k^K \sum_j^3 b_j(s_k,t_k)\tilde{f}(v_j^{z_k}) \tag{8}$$

where $K$ denotes the number of new generated triangles, $\{z_k\}$, and $\{v_j^{z_k}\}$ are vertices on the new tessellated triangles. Similarly, the Bézier triangle approximation is changed to

$$f_z(x,y) \approx \sum_k^K \sum_{i+j+l=n}^{n} B_{i,j,l}^n(s_k,t_k)c_{ijl}^{z_k} \tag{9}$$

where $c_{ijl}^{z_k}$ are control points on the new tessellated triangle $z_k$. To guarantee that the approximation domains remain unchanged after the subdivision, the positions of new geometry primitives are all kept on the original triangles.

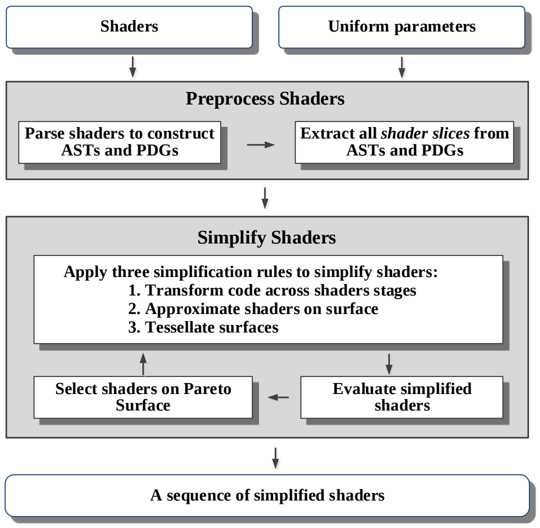

Shaders
Uniform parameters
Preprocess Shaders
Parse shaders to construct ASTs and PDGs
Extract all *shader slices* from ASTs and PDGs
Simplify Shaders
Apply three simplification rules to simplify shaders:
1. Transform code across shaders stages
2. Approximate shaders on surface
3. Tessellate surfaces
Select shaders on Pareto Surface
Evaluate simplified shaders
A sequence of simplified shaders

**Figure 2:** *Algorithm overview*

### 3.2.2 Cost Model

Visual error and rendering time are two common cost functions that have been used in previous cost models [Pellacini 2005; Sitthi-Amorn et al. 2011] to evaluate simplified shaders. In this work, we introduce the third cost function: memory consumption. When we approximate shader functions on surfaces, control points of fitted Beizer triangles are stored as extra textures for accessing at runtime. According to different types of signals, the memory consumption varies. That is, if these Bézier triangles approximate intermediate variables, the simplified shader may require more memory storage. If original textures are approximated, the overall memory consumption may be reduced.

Given the importance of memory consumption in our shader simplification scheme, we define a triple, $(t, e, m)$, to evaluate simplified shaders, where $t$ is the shader computation time, $e$ is the visual error and $m$ is the memory cost. Specifically, we count the rendering clocks of shaders on the GPU as the time cost, compute the average per-pixel $L^2$ distance in the RGB space over all representative frames as the visual cost, and measure the overall memory usage as the memory cost.

We extend the partial order defined in [Sitthi-Amorn et al. 2011] to three dimensions: one simplified shader with $(t_1, e_1, m_1)$ dominates another simplified shader with $(t_2, e_2, m_2)$, if $t_1 \leq t_2 \wedge e_1 \leq e_2 \wedge m_1 \leq m_2$, and $t_1 < t_2 \vee e_1 < e_2 \vee m_1 < m_2$. That is, one simplified shader dominates another if it improves in rendering time, visual error or memory consumption, and is at least as good in the others. Based on the partial orde, the preferred simplified shaders are selected from a Pareto surface in the three dimensional cost space. The actual distribution of simplified shaders in the cost space is plotted for every example shader in the result section.

## 4 Shader Simplification using Surface Approximation

### 4.1 Algorithm Overview

The flowchart of our algorithm is illustrated in Figure 2. The input includes one shader configuration with shaders and a set of uniform parameters sampled from their domains. As a preprocess, we first parse these shaders, and convert shader code into Abstract Syntax Trees (ASTs) [Muchnick 1997] and Program Dependent Graphs (PDGs) [Ferrante et al. 1987]. AST is a tree representation of the abstract syntactic structure of shader code, and PDG records the data and control dependencies of each variable and operation. Then, for each variable and operation in the shader, we compute a program slice [Weiser 1984], called a *shader slice* in our paper. A *shader slice* is a shader program that is taken with respect to a certain program point and a variable *x*; a slice consists of all statements of the program that might affect the value of *x* at a certain program point. One *shader slice* is regarded as one basic simplification primitive for our shader simplification.

Simplifying a shader configuration is performed with three simplification rules. The first rule transforms *shader slices* in the fragment shader to shaders in other stages. The second rule optimizes the original shaders by approximating the surface signals generated by *shader slices* with different orders of Bézier basis functions. The final one is to employ tessellation shaders or geometry shaders to generate new vertices and triangles. For each new generated triangle, we iteratively apply the previous two rules, namely, either transform code from the fragment shader to new vertices, or fit a new Bézier approximation for the new triangle. Every possible variation of applying these rules to the selected *shader slice* will generate a new shader configuration. Each new shader configuration is evaluated with costs concerning rendering time, memory consumption and the difference of rendered image. A Genetic Programming based optimization [Sitthi-Amorn et al. 2011] with our specific modifications is used to select potentially preferable shaders from the Pareto surface in the three dimensional cost space, and drive the optimization iteratively.

### 4.2 Parsing Shaders

Beginning from given shaders in the original shader configuration, we parse these shaders and construct ASTs and PDGs for each shader. In Figure 3, we show an example case where a geometry shader and a fragment shader are given as the original shader configuration.

To extract the *shader slice*, we take several operations in addition to the standard program slicing techniques [Tip 1995]. First, we take the input uniform parameters into consideration in computing the *shader slices* from the PDGs, so as to omit some dependencies that do not occur in the execution of the particular shade code with these parameters. Second, for loops with fixed iterations or iterations determined by uniform parameters, we unfold the loop and encapsulate each iteration into a *shader slice* so that these iterations can be optimized individually using the code transformation or the Bézier surface approximation scheme. All extracted *shader slices* are stored in a list in the order of their locations in the original shader and are processed in the shader optimization step one-by-one.

### 4.3 Transforming Code across Shaders

We take *shader slices* as basic primitives to perform code transformation. Three types of code transformations are supported in our approach: from the fragment shader to the vertex shader, to the geometry shader, and to the domain shader (the tessellation evaluation shader). In practice, the shader model version of the graphics hardware determines the type of the code transformation.

Though destinations are different, operations of different types of code transformations commonly take three main steps. They are statement relocation, repetition removal, and output merge. First, the statements of one *shader slice* are appended to the target shader, and are removed from the fragment shader. If the target is a vertex shader or a domain shader, it is directly appended to the end of all

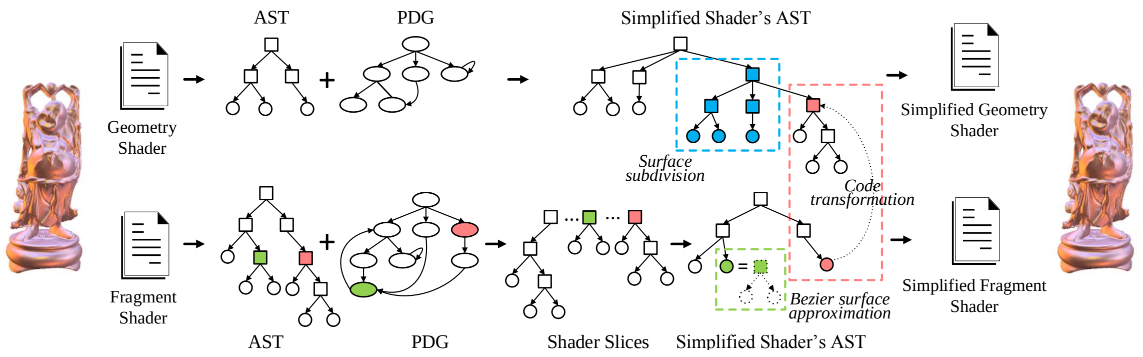


**Figure 3:** *Illustration of our shader simplification process. At first, shader code are converted into ASTs and PDGs. Then, shader slices are extracted from the fragment shader's PDG. Three illustrative cases of applying our simplification rules on shader slices are illustrated. Resultant modifications on ASTs are shown and highlighted in dashed boxes for different cases. In the red dashed box, the shader slice in the red box is transformed from the fragment shader to the geometry shader. In the green dashed box, the shader slice in the green box is approximated by Bézier functions. In the blue dashed box, the surface subdivision is triggered in the geometry shader, where new inserted tessellation code are shown in blue. After all simplification rules are applied, new shader code is generated from these ASTs of shaders, and is used to render scenes.*

computations. If the target is a geometry shader, these statements are added before the vertex emission statements. Thereafter, a code scan is performed to eliminate all repetitive statements. The output variables of the transformed *shader slice* are added into the output list of the vertex shader and domain shader, or the emitted vertex attributes of the geometry shader. One *shader slice* transformation produces a new shader configuration (a simplified fragment shader and an enlarged vertex, geometry or domain shader).

In Figure 3, we illustrate a code transformation for one *shader slice*, of which the statement is marked in the red box. Such a *shader slice* is transformed from the fragment shader to the geometry shader. The resultant changes on ASTs of simplified shaders are highlighted in the red dashed box. After such a code transformation, the computation of *shader slice* in the red box is taken in the geometry shader. Only their interpolations are input to the fragment shader and take part in following computations. Please refer to the supplementary document for more details and code examples.

## 4.4 Approximating Shader Slices on the Surface

Our approach allows for computing the output of each pixel and approximating them on the projected surface. The key idea is to fit the distribution of each output variable (a kind of surface signal) with Bézier triangles. The approximation is performed for each *shader slice* with three steps: sampling the surface signal discretely; fitting discrete samples with Bézier triangles; integrating the computed Bézier triangles to generate simplified shaders. An illustrative example case applying this rule is given in the supplementary document.

### 4.4.1 Sampling Surface Signals

The signal of one *shader slice* on the surface can be modeled by discretely sampling its output variables. For a *shader slice* in a fragment shader, we first extract all triangles shaded by this fragment shader, and then uniformly generate samples on each triangle and compute signal values on these samples. In practice, to compute these output values, we interpolate required geometry attributes and evaluate the values using a shader simulator.

### 4.4.2 Fitting Surface Signals

Fitting sample points generated by one *shader slice* with Bézier triangles is performed globally on the entire sampled surface. To compute the optimal control points of triangle, $z_i$, with certain order, a discrete least square optimization equation can be formed by substituting Eq.(6) into Eq.(7), and formulating it into a matrix formation:

$$\mathbf{A}_{z_i}^T \mathbf{A}_{z_i} \mathbf{c}_{z_i} = \mathbf{A}_{z_i}^T \mathbf{f}_{z_i} \tag{10}$$

where $\mathbf{A}_{z_i}$ is the observation matrix, each element of which is the value of the Bernstein polynomials at the sample points. $\mathbf{c}_{z_i}$ are control points, and $\mathbf{f}_{z_i}$ denote the variable values at sample points.

In many situations, the surface signals are continuous at a large scale. Accordingly, a smooth constraint on the boundaries among surface triangles is demanded, yielding a global least square equation:

$$\begin{bmatrix} \mathbf{A}^T\mathbf{A} & \mathbf{H}^T \\ \mathbf{H} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \alpha \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T\mathbf{b} \\ 0 \end{bmatrix} \tag{11}$$

where $\mathbf{A} = [A_{z_0}, ..., A_{z_i}, ...]$ is a diagonal matrix composed from all triangles, $\mathbf{H}$ denotes the smooth constraints on boundaries, and $\alpha$ is the Lagrange multiplier. Our approach employs open source software [Guennebaud et al. 2010] to solve the linear equation.

### 4.4.3 Simplifying Shaders with Fitted Functions

Note that the motivation for signal fitting is to replace the original *shader slice* with fitted Bézier triangles, and thereby reduce computations. To allow the fitted Bézier triangles be correctly accessed by pixels in the fragment shader, we first store all control points in an additional buffer, and index each set of control points of Bézier triangles by the triangle ID. Then, we bind the buffer to a texture and pass the texture ID and triangle ID to the fragment shader through the vertex shader. To let pixels be correctly interpolated, we explicitly pass the barycentric coordinates to the fragment shader as well. This is done with the geometry shader by creating the barycenteric coordinates of vertices and interpolating within the raterization process [Bæentzen et al. 2008].

## 4.5 Subdividing Surfaces

The runtime surface subdivision relies on the hardware implementation of programmable tessellations. To make it adaptable to different hardware, we implement two versions of surface subdivision with geometry shader or tessellation shaders, respectively. For hardware compatible with shader model 5, we perform the subdivision in tessellation shaders. For those compatible with shader model 4, we use the geometry shader to subdivide surfaces. The implementation details vary for the tessellation shaders and the geometry shader, but mainly take two steps.

First, to generate one simplified shader, we choose a certain tessellation factor to control the number of sub-triangles. In the hull shaders, we use different inner and outer tessellation factors to control the subdivision level and the pattern of sub-triangles. In the geometry shader, we use a number of iterations to generate sub-triangles at different subdivision levels. For simplicity, in the geometry shader, we always partition one triangle into triangular grids with a uniform tessellation factor on each edge.

Second, we apply two different simplification rules on these new generated sub-triangles. For the code transformation rule, we append transformed *shader slice* code into the domain shader or insert the code before the vertex emission statements in the geometry shader. For the Bézier triangle approximation, we assign each sub-triangle a unique triangle ID using the original triangle ID with the tessellation pattern. Then, we use the signal fitting method proposed in the previous section to compute the control points for each sub-triangle. To let each fragment shader correctly index the sub-triangle ID and access corresponding control points, we use the geometry shader to compute the unique ID of each sub-triangle, where the original triangle ID, the local sub-triangle ID and the tessellation pattern are input as parameters. More details and the example code are provided in the supplementary document.

## 4.6 Selecting Simplified Shaders

For each *shader slice*, there are tens of potential simplified variants by means of the three simplification rules. Given the hundreds or thousands of lines of shader code, the size of all variants of simplified shaders is the combination of simplified variants of all *shader slices*. Such a shader space is too large to be fully explored. To better select simplified shaders, we adapt the Genetic Programming (GP) proposed in [Deb et al. 2002; Sitthi-Amorn et al. 2011] with several modifications.

First, in our approach, the element of the population is a combination of *shader slices* applied with our simplification rules. The mutation in our optimization is to change the simplification rules applied on *shader slices* that have been simplified. The crossover is to exchange simplified *shader slices* for two variants. Second, we observe that for one *shader slice*, its simplified variants can be sorted by their partial orders. Thus, before taking the Genetic Programming optimization, we first compute all simplified variants for one *shader slice*, and select local preferred shader variants lying on the Pareto surface. Only those local preferred shader variants are regarded as candidates of this *shader slice* to be mutated by itself or crossover with other *shader slices*. These local preferred shader variants are also used to initialize the population. Third, we order all *shader slices* according to their locations in the original shader code, and use the order to help in mutation and crossover to generate new variants. Since the *shader slices* usually have dependencies on their previous *shader slices*, in code transformation, the location order of the *shader slice* is very helpful to reduce the number of variants.

After the final iteration, all variants ever produced and evaluated are used to compute a single unified Pareto surface. The variants on that surface are output as preferred shaders.

## 5 Results

We implement the approach with Visual C++ and HLSL (DirectX3D). Our shader parser is built on Lex and Bison, and the

| | GPU Importance Sampling | Marble | TF2 | Imrod |
|---|---|---|---|---|
| *Scene* | | | | |
| Triangles # | 20 K | 10 K | 7.2K | 26 K |
| Vertices # | 39.7 K | 8.6 K | 5.1K | 13.3 K |
| Memory # | 2.0 MB | 62.5 KB | 2.0 MB | 5.38 MB |
| *Line of Code* | | | | |
| Source | 120 | 99 | 157 | 194 |
| *Shader slices* | 180 | 66 | 129 | 272 |
| *Shader Simplification* | | | | |
| Generated variants | 1778 | 1501 | 1296 | 7597 |
| Pareto variants | 76 | 59 | 34 | 75 |
| Time (hours) | 2.5 | 7.1 | 1.4 | 6.8 |
| Code transform | 69.7% | 76.08% | 81.7% | 78.9% |
| Bézier fit | 64.4% | 74.7% | 55.9% | 61.5% |
| Subdivision | 26.% | 70.0% | 61.0% | 28.9% |
| *Results* | | | | |
| Speedup @ $0.075L^2$ error | 2.10 | 3.67 | 1.77 | 3.76 |

**Table 1:** *Configurations and statistics of example shaders. From top to bottom: the scene complexity; the code complexity; the statistics in the shader simplification section including the number of all generated variants, the number of variants on the Pareto Surface, the computation time, and usages of different simplification rules; and the speedup ratio given a visual error threshold.*

Genetic Programming-based shader selection algorithm is modified from NSGA-II [Deb et al. 2002; Sitthi-Amorn et al. 2011]. Experiments are conducted on a PC with an Intel Core$^{\text{TM}}$ i7 3770 CPU and different graphics cards. We use the hardware counters, $gpu\_time$, provided by the NVidia PerfSDK to measure rendering times. All images were rendered at a resolution of $1920 \times 1080$. In the following figures, some images are cropped to $1080 \times 1080$ for comparisons. In all results, the texture mipmaps are disabled to better compare the approximated signals with the original surface signals. Please refer to the supplementary files for full resolution images, and the supplementary video for side-by-side comparisons. Note that the supplementary video is downsampled to 720P for a smaller file size.

### 5.1 Example Shaders

We test our approach on four example shaders: the GPU-based Importance Sampling shader on the Buddha model, the Marble shader on the Dragon model, the NPR shader on the Heavy model and a complex lighting shader on the Imrod model. Table 1 summarizes configurations and statistics of these example shaders, including the scene complexity, the shader complexity, details of the shader simplification, and the speedup with 0.075 $L^2$ visual error. For each example, we also plot out the distribution of variants in the 3D cost space. Each shader variant is plotted as a sphere. Variants on the Pareto surface are marked in orange. Three highlighted shaders are colored in blue, green and red respectively. Other variants are plotted in purple. To better understand the distribution of variants in the space, we projected each variant on three planes, i.e., the time-error plane, the error-memory plane and the time-memory plane. These projections provide a good hint on how these variants distribute in the space. The source code of all simplified shader variants on Pareto surfaces are provided as supplementary files.

#### 5.1.1 GPU-based Importance Sampling Shader

The GPU-based Importance Sampling Shader is adapted from [Colbert and Křivánek 2007] and runs on an NVIDIA GTX 760 graphic card. This shader enables real-time relighting of glossy objects under environment maps. The importance sampling guides the sam-

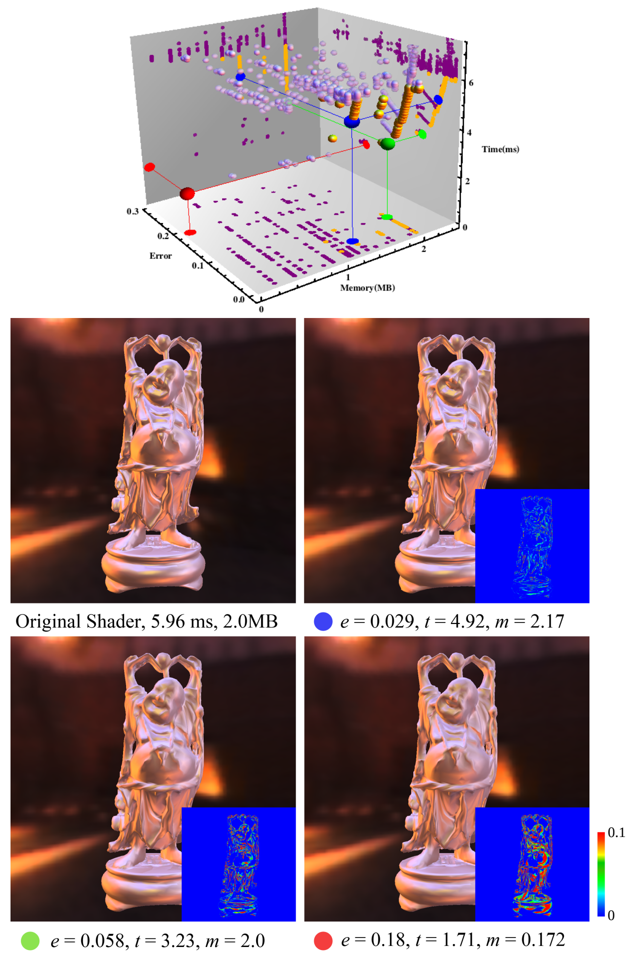


**Figure 4:** *The Importance Sampling Shader on the Buddha model.*

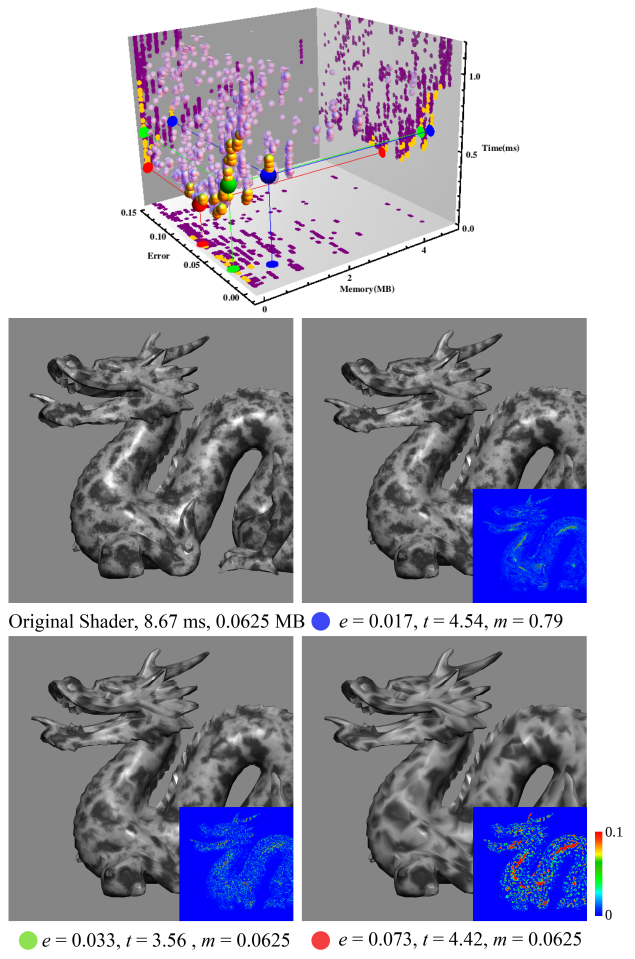


**Figure 5:** *The Marble shader on the Dragon model.*

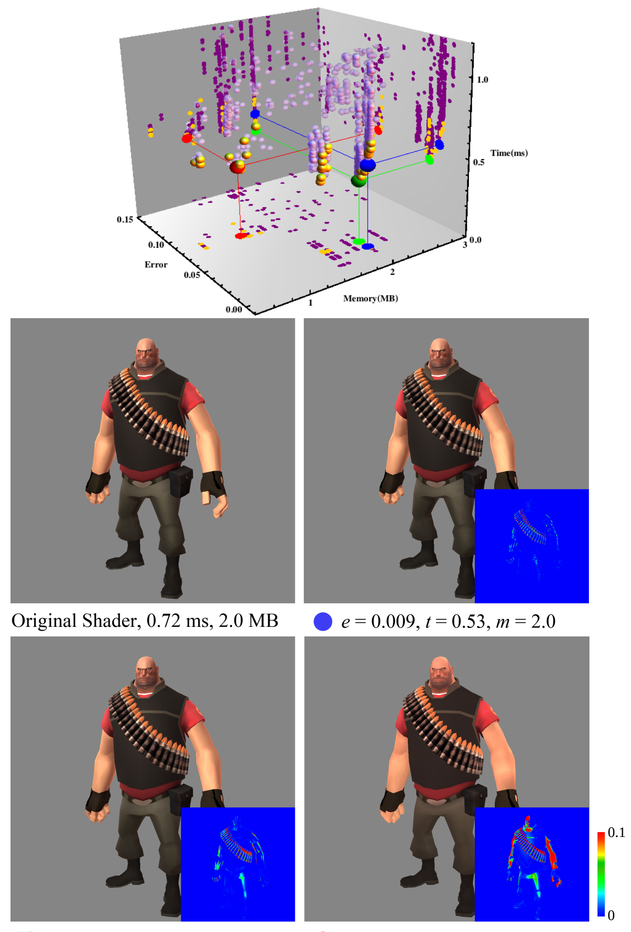
pling of a number of directions from the material function of object. For these directions, a set of mipmap filters with different sizes given by probability density function values are used to filter the environment map. In our implementation, we generate 40 samples, leading to 40 looped iterations in the fragment shader. The performance of each simplified shader is measured by 64 image frames with varying lighting directions and viewpoints. A total of 1778 variants are generated, and 76 variants on the Pareto surface are finally selected as preferable shaders. The code transformation rule is the most used rule, and the subdivision rule is the least used one. This is mainly because the base mesh of the Buddha model is dense. Shader variants with low level tessellations are able to provide good balance between rendering time and visual quality. The distribution of these shaders in the cost space is plotted in Figure 4 (Top). Rendering results of the original shader and three variants are highlighted in Figure 4 (Bottom).

The first highlighted shader, shown in blue, is simplified by both the code transformation rule and the Bézier triangle approximation rule. 18 of 40 sampling iterations are transformed to the vertex shader, and an intermediate variable is fitted by Bézier functions. These two simplifications yield a 1.21 times speedup, a slight loss in visual quality, and an extra buffer to store control points. The second highlighted shader, shown in green, is only simplified by the code transformation rule. All 40 sampling iterations are transformed to the vertex shader. That is, the relighting computations are all taken on vertices instead of on pixels. It causes some loss of pixel-wise glossy reflections but has 1.84 times speedup. The final shader, shown in red, is highlighted to show an extreme case resulting from our simplifications. This shader is simplified by an aggressive Bézier triangle approximation, where the final color of pixels computing from different lighting directions and view positions are all approximated by a 1-order Bézier functions. It yields 3.48 times speedup, and reduces the memory consumption to 0.172 MB. It uses a smaller memory foot print because all environment lighting has been baked on the surface, and only control points are stored for runtime rendering.

### 5.1.2 Marble Shader

The Marble shader renders a model of four octaves (generated with a standard procedural 3D noise function [Perlin 1985]) with the Phong BRDF shading model [Phong 1975]. It runs on an NVIDIA GTX 680 graphics card. The representative image sequence consists of 64 frames that depict the model at different spatial positions and with different lighting directions. A total of 1501 variants are generated by our approach, where 50 of them are on the Pareto Surface. It takes the longest time to simplify this shader. This is mainly because 70% and 74.7% of shader variants are simplified by the subdivision rule or the Bézier approximation rule, or both. While many sub-triangles are tessellated and approximated on surfaces, fitting signals on these sub-triangles occupies a large amount of time. The distribution of these shader variants in cost space is plotted in Figure 5 (Top). Three variants on the Pareto surface are highlighted and shown in Figure 5 (Bottom).

The first highlighted shader, shown in blue, is simplified by all three simplification rules. It contains a code transformation, a 2-order



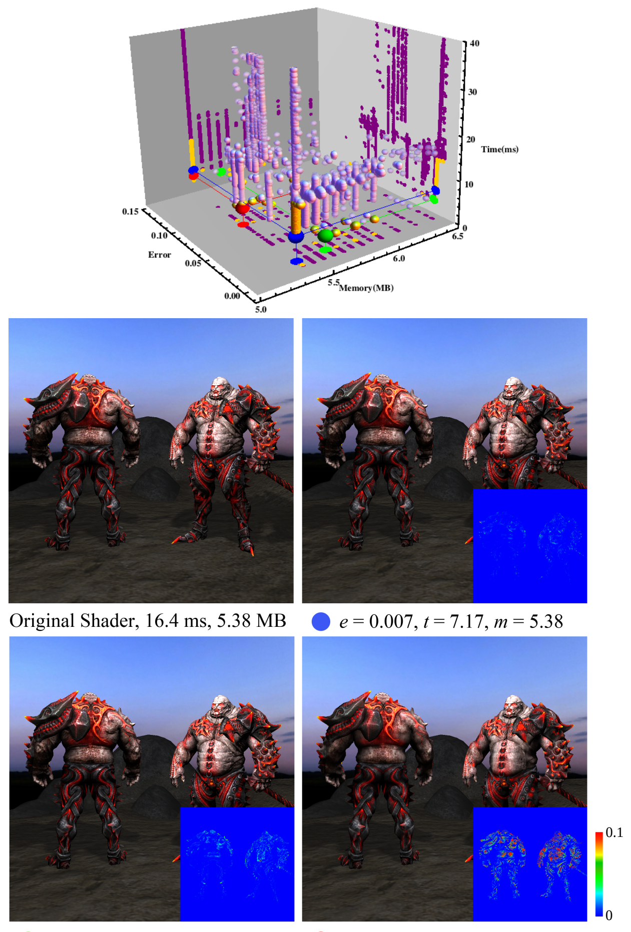
Original Shader, 0.72 ms, 2.0 MB

🔵 $e = 0.009$, $t = 0.53$, $m = 2.0$

🟢 $e = 0.020$, $t = 0.41$, $m = 2.0$

🔴 $e = 0.075$, $t = 0.94$, $m = 1.1$

**Figure 6:** *The NPR shader on the Heavy model.*



Original Shader, 16.4 ms, 5.38 MB

🔵 $e = 0.007$, $t = 7.17$, $m = 5.38$

🟢 $e = 0.009$, $t = 4.36$, $m = 5.59$

🔴 $e = 0.074$, $t = 5.43$, $m = 5.38$

**Figure 7:** *The complex lighting shader on the Imrod Model.*

Bézier triangle approximation and a two-level triangle subdivision (tess_factor=3). A total of 13 sub-triangles are generated to approximate the original noise signal from pixel-level to vertex-level. Additionally, a portion of shading computation that computes specular per-pixel shading is moved to the domain shader to compute per-vertex. After applying these three simplifications, this shader variant runs 1.76 times faster than the original shader with a 0.025 $L^2$ visual error and an extra 0.681 MB memory consumption. The second highlighted shader, shown in green, is also simplified by a combination of rules. In this shader variant, one triangle is subdivided into 6 sub-triangles, and some portion of the specular reflections are computed on vertices. These simplifications make this variant 1.88 times faster and with a 0.026 $L^2$ error. The final highlighted shader, shown in red, moves all shading computations from pixels to vertices, and only fits the noise signal on original triangles by an 1-order Bézier triangle functions. As expected, it heavily speedups the rendering time but produces a large visual error.

We also simplify this shader on an NVIDIA GTX 280 graphic card, which only supports shader model 4.0. On this hardware, we use the geometry shader instead of the tessellation shaders to perform surface subdivision rule. These simplified shaders on the Pareto surface are provided in the supplementary shader files.

### 5.1.3 NPR Shader

We adapt the real-time Non-Photorealistic Rendering (NPR) shader used in the game "Team Fortress 2 (TF2)" [Mitchell et al. 2007] to test our method. This NPR shader combines a variety of view-independent and view-dependent terms. The view-independent terms consist of a spatially-varying directional ambient term plus modified Lambertian lighting terms. The view-dependent terms are a combination of Phong highlights and customized rim lighting terms. All lighting terms are computed per-pixel with an albedo map. We apply this shader on the Heavy character model with 9 point lights. The simplification is taken on 54 image frames with respect to an animated sequence. A total of 1296 variants are generated, and 34 variants are selected from the Pareto surface. Figure 6 shows the distribution of shader variants (Top) with several highlighted shader variants (Bottom).

The first highlighted shader, shown in blue, is simplified by a code transformation rule and a surface subdivision rule. The lighting of 8 of 9 point lights are computed on vertices instead of on pixels. To compensate the error brought by this simplification, 6 sub-triangles (tess_factor=2) are subdivided per triangle to capture the shading signal on smaller patches. This shader variant reduces rendering time to 0.53 ms and only generates a relatively low visual error, 0.009 $L^2$ error. This is mainly due to the low frequency of the NPR shading signal on pixels. The interpolation of shading values from vertices produces good approxiamtions. The next highlighted shader, shown in green, is simplified by the same code transformation of the first highlighted shader, but skips the subdivision. It leads to better performance, 1.76 times speedup compared with the original shader, but has a larger error than the first shader variant. Without the subdivision, some loss of specular reflections on the character's arm can be noticed. The final shader in red is a shader with Bézier triangle approximations. Two *shader slices* are

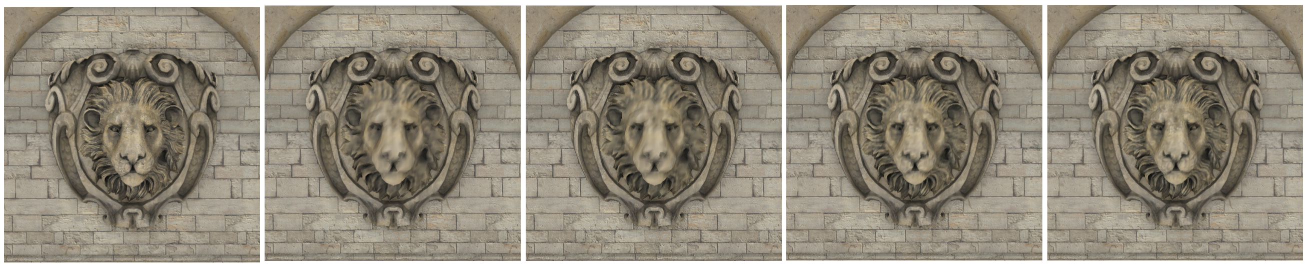

(a) Original shader: Diffuse + AO Texture (b) Gradient-based const. [Kavan et al. 2011] (c) Edge-based const. [Kavan et al. 2011]] (d) 2-order Bézier approx. Our simplification (e) 3-order Bézier approx. Our simplification

**Figure 8:** *Comparisons with the vertex-baking of [Kavan et al. 2011]. The texture and ambient occlusion are applied on the Lion head model with 3.5K triangles. (a) the ground truth shading effect of the Lion head model. (b-c) show results of the vertex-baking technique using a gradient-based and an edge-based regularization respectively. (d-e) show our results with a 2- and 3-order approximation respectively.*

approximated in this shader. First, the shading of 7 point lights is linearly approximated on triangles, while the remaining two lights are still computed independently. Second, the computation of specular lighting effects at rims and with other textures are approximated on triangles. Though the overall error and rendering time of this shader variant are larger than previous shader variants, its total memory consumption reduces to 1.1 MB by baking some textures and the specular lighting effects at rims on the surface.

#### 5.1.4 Complex Lighting Shader

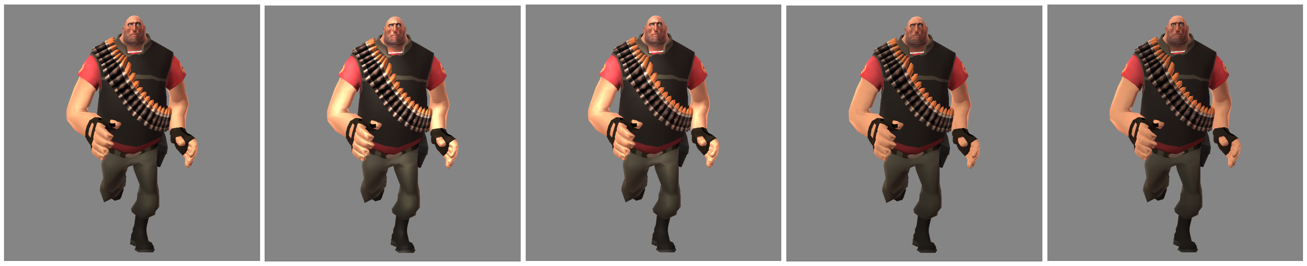
We test our approach on a complex lighting shader on the Imrod model using an NVIDIA GTX 760 card. We adapt the GPU importance sampling shader to this scene, and add a complex shading model using a shadow map (which is computed in an independent pass) and several material textures, including albedo, specular, normal, emissive and ambient occlusion maps. Environment lighting and one point light cast light on the model. We use 64 representative frames with different view positions as input uniform parameters. A total of 7597 variants are generated, and 75 of them are on the Pareto surface. This example shader has the largest number of variants, but only 28.9% of them use the surface subdivision rule. This makes the simplification process take less time than that of the Marble shader. The resultant distribution of shader variants with three highlight results are shown in Figure 7.

The first highlighted shader, shown in blue, is simplified by a code transformation. 37 iterations that sample the environment lighting are transformed from the fragment shader to the vertex shader. From the rendered image, it can be seen that though some shading details are smoothed, it still leaves main details on the surface. The second highlighted shader, shown in green, is simplified by a code transformation and a Bézier triangle approximation. First, the texture fetch of the ambient occlusion map is moved from the fragment shader to the vertex shader. It converts the pixel-wise ambient occlusion signal to a vertex-wise surface signal. For a low frequency signal like ambient occlusion, such an approximation only incurs a slight visual loss but reduces some texture reads. The second simplification is to approximate the shading from environment lighting by a Bézier triangle approximation. This simplification sacrifices some memory but has a big performance improvement, 3.75 times speedup, and only produces a small visual error, 0.009 $L^2$ error. The insight behind such a big improvement is that even in a model that has complex material and is lit by complex lighting, not all interactions between the lights and materials produce high frequency signals. Some resultant signals might be relatively low-frequency. In these cases, our method automatically approximates signals on surfaces and select variants that have good approximations of those low frequency signals. The final highlighted shader, shown in red, is simplified by transforming two computations to the vertex shader. One is the texture fetch of the normal map and some portion of the specular computations using the normal map. It smooths the shading details on the surface. As can be seen in the image, the detailed glossy reflections on the character's belly and back are dimmed. The other code transformation is taken on the sampling of tbe environment lighting. All environment lighting is computed at vertices. Such simplifications produce 0.073 $L^2$ visual error, but achieve 3 times speedup of rendering time. Compared with the second highlighted shader, this shader consumes less memory.

### 5.2 Discussions

**Approximation Errors.** Given these simplification rules, our approach tends to perform a low-order approximation on the underlying surface signals. Apparently, it works well when such signals are low-frequency. Figure 8 illustrates the comparison between our approach and the vertex-baking technique of [Kavan et al. 2011]. It can be observed that with the increasing of orders of Bézier functions, our approach is able to better reconstruct the original surface signals, and yields a more accurate approximation when the surface contains coarse details. However, not all signals are low-frequency and can be approximated well by our low-order approximations. For example, the generation of all four octave noises on the Dragon model, and the specular reflections from the bump mapping on the Imrod model. These approximation errors may be large. In these cases, our Genetic Programming-based optimization always tends to select variants that produce smaller visual errors, or produce large errors but reduce the time or memory consumption, i.e., the variants around the Pareto surface. These variants approximating high frequency signal and producing large errors will be automatically omitted because they are far away the Pareto surface. In this way, without any frequency analysis or knowledge of the frequency of surface signals, our method is still able to automatically explore the shader variant space, and produce a set of preferable simplified shaders. It will be an interesting topic to conduct specific frequency analysis on signals generated by the fragment shader and derive better approximations for the shader simplification. We will regard it as future work.

**Other Simplification Rules.** Compared with previous shader simplification methods, our approach does not simplify within the fragment shader but takes the simplification across multi-shader stages. Thus, it can be regarded as an orthogonal approach to previous methods. In Figure 9, we give two examples showing that our shader simplification can be combined with simplification rules in previous methods and achieve better results. We use two expression simplification rules proposed in [Pellacini 2005] and [Sitthi-Amorn et al. 2011] to simplify the NPR shader respectively. From [Pellacini 2005], we use an expression aggregation rule, $a+b \to a$, and from [Sitthi-Amorn et al. 2011], we use a random operation swap. Results simplified by these two rules are shown in Figure 9(b) and Figure 9(d). Then we take these two simplified shaders as input



(a) e = 0, t = 1.42ms, Original shader | (b) e = 0.043, t = 1.35 ms, [Pellacini 2005] | (c) e = 0.048, t = 1.15 ms, Our simplification on (b) | (d) e = 0.035, t = 1.25 ms, [Sitthi-Amorn et al. 2011] | (e) e = 0.035, t = 1.1 ms, Our simplification on (d)

**Figure 9:** *Results of combined simplification rules from previous methods and our approach. (a) The original shader. (b) The shader simplified by rules proposed in [Pellacini 2005] (c) The simplified shader in (b) is further simplified by our approach. (d) The shader simplified by rules proposed in [Sitthi-Amorn et al. 2011]. (e) The simplified shader in (d) is further simplified by our approach. As can be seen, with a similar visual error, our approach further reduces the rendering time.*

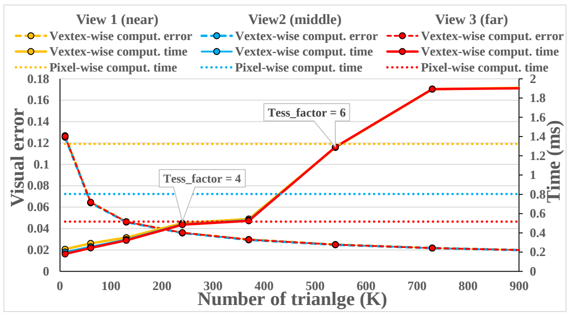


**Figure 10:** *Shader simplification on the Dragon scene with different views. View 1 is the default view used in Figure 5 with 660K shading pixels. View 2 is a zoomed out view with 330K shading pixels. View 3 is the furthest view with 165K shading pixels.*

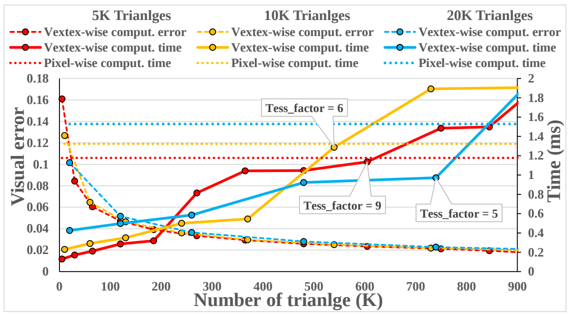


**Figure 11:** *Shader simplification on the Dragon scene with different base models.*

shaders to further simplify them by our rules. Results are shown in Figure 9(c) and Figure 9(e). After applying our simplification, results are generated with similar visual qualities but with additional reduced rendering times.

**Pixel-wise Computation vs. Geometry-wise Computation.** The basic idea of our solution is the approximation of pixel-wise computations by geometry-wise computations. Triangles are used as basic primitives to store and approximate pixel-wise surface signals. Once the number of geometry primitives is much less than the number of pixels, our approach will bring benefits on reducing computational time. To better understand our method, we conduct two experiments on the Marble shader. One experiment uses a fixed view and three Dragon models with different numbers of triangles. The other one uses one Dragon model but with different views, i.e. different shading pixels. For each experiment, we first employ code transformations to move all computations from pixels to vertices. Thereafter, we apply the tessellation rule to subdivide triangles into different number of sub-triangles. Supposedly, with more sub-triangles, the rendering time increases but the visual error decreases. Detailed rendering times and visual errors are plotted in Figure 10 and Figure 11 for these two experiments respectively.

In Figure 10, three different views are tested. One is the default view used in Figure 5 with 660K shading pixels, and the other two are views zooming out with 330K and 165K shading pixels, respectively. Because the geometry-wise computation only depends on geometry primitives, the relative visual errors and the rendering times under three views are similar. From the chart, it can seen that with the increase of tessellation levels, the visual errors fast drop below the 0.075 $L^2$ error threshold, but at the same time, the rendering time of simplified shaders is still much less than the original shader. It indicates that there is a much space to apply our simplification rules. For the near view, the vertex-wise computations on a level 6 tessellation (tess_factor = 6) is still faster than the original shader. Even under the furthest view, a level 4 tessellation (tess_factor = 4) still brings some benefits from the original shader. But it also can be seen that while the number of shading pixels decreases, the simplification gain decreases. For example, with the same tessellation factor, tess_factor = 4, the approximation of transforming all pixel-wise computations to vertex-wise brings a 0.80 ms, 0.30 ms and 0.05 ms faster on different views.

In Figure 11, three base models with 5K, 10K and 20K triangles are tested. We highlight the spots where simplified shaders are just faster than the original shader. From the chart, it can seen that we can apply our simplification rules over a big range of triangle numbers before the vertex-wise computations finally become slower than pixel-wise computations. Another interesting time statistic is that different base meshes have different performance in tessellations. For example, compared with other two meshes, the mesh with 20K base triangles has the best performance at a tessellation of 750K triangles, but the mesh with 10K base triangles performs worst around 600K triangles. Thus, it seems impractical to derive a simple and general equation to determine the best tessellation for a model with a certain number of base triangles. Our automatic method employs an automatic selection step to explore the shader variant space and find the tessellation factors with best performance. This makes our method general for different scenes. Additionally, the different time performance curves of different base models also indicate that for a model with certain shaders, there may exist a best base mesh to apply simplified shaders. Such a base mesh may depend on many factors, such as the hardware, the computation in shaders, etc.. It will be interesting future work to compute the best tessellations for one model with ceratin shaders.

## 6 Conclusion

This paper introduces a novel and practical scheme for automatically simplifying fragment shaders. The key idea is to solve a global shader simplification problem using a surface signal approximation and distributing computations in multiple rendering stages. Three simplification rules are proposed: the code transformation, the shader function approximation and the surface subdivision. We provide an integrated simplification scheme to automatically select simplified shaders, and demonstrate its effectiveness and efficiency with a suite of examples. Our approach offers an orthogonal method to existing fragment shader simplification solutions, and can be combined with them to further simplify fragment shaders.

In the future, we want to continue exploring some problems that have been discussed in the Section 5.2, such as other surface signal approximations, the potential best tessellation for one mesh, etc.. We also would like to explore and study a more sophisticated cross-shader optimization scheme that not only simplifies the fragment shaders but optimize all shaders in all stages.

## Acknowledgements

We would like to thank all reviewers for their thoughtful comments. We also want to thank Chao Xu for preparing demos, thank Dmitriy Parkin (the Imrod model), Jing Wang (the shader of the Imrod model), Jason L. Mitchell, Moby Francke, Dhabih Eng, and the Valve Corporation (the Heavy model) for permissions to use their models and shaders. This work was partially supported by NSFC (No. 61472350), the 863 program of China (No. 2012AA011902), the Specialized Research Fund for the Doctoral Program of Higher Education of China (No. 20110101130011), and the Fundamental Research Funds for the Central Universities (No. 2014FZA5017).

## References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

BÆENTZEN, J., NIELSEN, S., GJØL, M., AND LARSEN, B. 2008. Shader-based wireframe drawing. *Computer Graphics and Geometry 10*, 2, 66–79. Invited paper. Extended version of previous conference paper.

COLBERT, M., AND KRIVÁNEK, J. 2007. Gpu-based importance sampling. *GPU Gems 3*, 459–476.

DEB, K., PRATAP, A., AGARWAL, S., AND MEYARIVAN, T. 2002. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp 6*, 2 (Apr.), 182–197.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July), 319–349.

FOLEY, T., AND HANRAHAN, P. 2011. Spark: Modular, composable shaders for graphics hardware. *ACM Trans. Graph. 30*, 4 (July), 107:1–107:12.

GUENNEBAUD, G., JACOB, B., ET AL., 2010. Eigen v3. http://eigen.tuxfamily.org.

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '95, 343–350.

HANRAHAN, P., SALZMAN, D., AND AUPPERLE, L. 1991. A rapid hierarchical radiosity algorithm. *SIGGRAPH Comput. Graph. 25*, 4 (July), 197–206.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '96, 99–108.

KAVAN, L., BARGTEIL, A. W., AND SLOAN, P.-P. 2011. Least squares vertex baking. In *Proceedings of Eurographics Conference on Rendering*, 1319–1326.

KESSENICH, J., BALDWIN, D., AND ROST, R. 2013. *OpenGL Shading Language Specification*. http://www.opengl.org/documentation/glsl/.

LEHTINEN, J., ZWICKER, M., TURQUIN, E., KONTKANEN, J., DURAND, F., SILLION, F. X., AND AILA, T. 2008. A meshless hierarchical representation for light transport. *ACM Trans. Graph. 27*, 3 (Aug.), 1–9.

MICROSOFT. 2013. *Direct3D 11 reference*. http://msdn.microsoft.com.

MICROSOFT. 2013. *Shader model 5 (DirectX HLSL)*. http://msdn.microsoft.com.

MITCHELL, J. L., FRANCKE, M., AND ENG, D. 2007. Illustrative rendering in team fortress 2. In *ACM SIGGRAPH 2007 Courses*, ACM, New York, NY, USA, SIGGRAPH '07, 19–32.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

NEHAB, D., SANDER, P. V., LAWRENCE, J., TATARCHUK, N., AND ISIDORO, J. R. 2007. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*.

OLANO, M., KUEHNE, B., AND SIMMONS, M. 2003. Automatic shader level of detail. In *Proceedings of Graphics Hardware*, 7–14.

PELLACINI, F. 2005. User-configurable automatic shader simplification. *ACM Trans. Graph. 24*, 3, 445–452.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of ACM SIGGRAPH*, 287–296.

PHONG, B. T. 1975. Illumination for computer generated pictures. *Commun. ACM 18*, 6 (June), 311–317.

SEGAL, M., AKELEY, K., FRAZIER, C., LEECH, J., AND BROWN, P. 2013. *The OpenGL Graphics System: A Specification (Version 4.4 (Core Profile) - October 18, 2013)*. http://www.opengl.org/registry/doc/glspec44.core.pdf.

SITTHI-AMORN, P., LAWRENCE, J., YANG, L., SANDER, P. V., NEHAB, D., AND XI, J. 2008. Automated reprojection-based pixel shader optimization. *ACM Trans. Graph. 27*, 5 (Dec.), 127:1–127:11.

SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. 2011. Genetic programming for shader simplification. *ACM Trans. Graph. 30*, 6, 1–12.

SLOAN, P.-P., HALL, J., HART, J., AND SNYDER, J. 2003. Clustered principal components for precomputed radiance transfer. *ACM Trans. Graph. 22*, 3 (July), 382–391.

TIP, F. 1995. A survey of program slicing techniques. *Journal of programming languages 3*, 3, 121–189.

WEISER, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10*, 4, 352–357.

ZATZ, H. R. 1993. Galerkin radiosity: A higher order solution method for global illumination. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '93, 213–220.