User-configurable Automatic Shader Simplification

Article *in* ACM Transactions on Graphics · July 2005

DOI: 10.1145/1073204.1073212 · Source: DBLP

1 author:

Fabio Pellacini
Sapienza University of Rome
94 PUBLICATIONS **2,992** CITATIONS

SEE PROFILE

User-Configurable Automatic Shader Simplification

Fabio Pellacini*
Cornell University

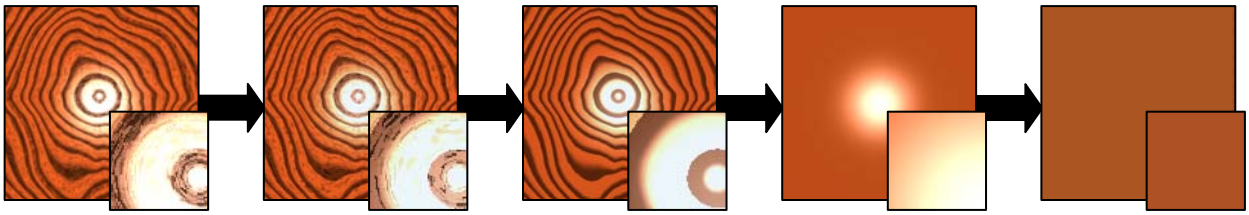

Figure 1: Sequence of increasingly simplified shaders automatically generated by our algorithm. Insets are enlarged 8 times.

Abstract

Programmable shading is a fundamental technique for specifying appearance in 3d environments. While shading architectures provides fast execution of shaders, shader evaluation is today a major cost in the rendering process. In the same manner in which geometric simplification lets us deal with large models, it would be beneficial to have an automatic technique that trades off shader quality for speed.

This paper presents such a technique by introducing a framework for the automatic simplification of complex procedural shaders, where a sequence of increasingly simplified shaders is generated starting from an original shader together with ranges for all of its input parameters. Our approach works by applying simplification rules to the code of a shader to generate a series of candidates, whose differences from the original one are measured and used to select the candidate with the smallest error. This procedure is repeated until the last shader is a constant. While this automatic procedure generates high quality simplified shaders, the artist might want to emphasize particular aspects of a shader during simplification. Our framework supports this desire by allowing the user to specify additional rules to be considered during simplification. The term *user-configurable* simplification comes from this feature of our system.

We implemented our algorithm to support the simplification of fragment shaders running on graphics hardware. Our results show that automatic simplification of complex procedural shaders is possible with high quality.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation I.3.6 [Computer Graphics]: Methodology and Techniques I.3.7 [Computer Graphics]: Three—Dimensional Graphics and Realism

Keywords: Interactive Rendering, Rendering Systems, Hardware Systems, Procedural Shading, Languages, Level of Detail

*e-mail: fabio@graphics.cornell.edu

1 Introduction

Programmable shading is a fundamental technique for specifying appearance in 3d environments first explored by software renderers [Cook 1984; Perlin 1985], then popularized by the Renderman Shading Language [Hanrahan and Lawson 1990] and today supported by most graphics hardware [Peercy et al. 2000; Proudfoot et al. 2001; Mark et al. 2003; Microsoft 2002; Kessenich et al. 2003]. Examples of the use of this technique can be found in [Ebert et al. 1998; Apodaca and Gritz 2000]. A shader is a procedure, written in a high-level language, that computes some aspect of the visual appearance of an object. Software renderers that implement the Renderman specification, a widely used industry standard, can use shaders to specify surface color and displacement, light color and direction, and volumetric effects. On graphics hardware, shaders are used to specify vertex transformations and pixel colors.

While shading architectures provide fast execution of programmable shaders, shader evaluation is today a major cost in the rendering process, and for large enough shaders such as the ones used in movie production environments or complex interactive applications, it can match and exceed the costs associated with hidden surface removal and geometry processing. This cost becomes especially problematic in real time applications, where hard constraints on framerate need to be met. In this context, geometric simplification is often used to provide a trade off between image quality and speed with respect to mesh size, but no technique is yet available to provide a similar trade off for arbitrary programmable shaders. In a seminal work that inspired our own, Olano et al.[2003] introduced the first approach for automatic shader simplification geared toward reducing memory accesses in realtime shaders, but did not address the reduction of arbitrary computation.

This paper introduces a framework for shader simplification, where a sequence of increasingly simplified shaders is generated given an original shader and ranges for all of its input parameters by applying a set of simplification rules that reduce the amount of computation needed at each step. The simplified shaders will depend not only on the original one, but also on the domain of its input parameters, giving us simplified shaders that can be used with changing parameters at run time.

While the basic simplification rules are often enough to provide high quality simplifications, shader authors might want to emphasize particular aspects of a shader in the case of extreme simplification. For example, the author might prefer to have a more accurate diffuse response and lose the highlights rather than having the error spread equally between the two. Our algorithm supports this feature

by allowing the artist to specify additional rules for the simplification algorithm. The term *user-configurable* simplification comes from this feature of our system.

We implemented our framework in the context of fragment shaders written in a subset of Cg [Mark et al. 2003] running on graphics hardware. Figure 1 shows a shader that generates a complex procedural pattern and four of its simplification generated automatically by our system. Our results show that high quality simplified shaders can be obtained completely automatically.

This paper is organized as follows: Section 2 reviews previous works, followed by a description of our algorithm in Section 3 and implementation details in Section 4. Section 5 describes the results generated, followed by the conclusion and discussion of future work in Section 6.

2 Related work

Automatic simplification of procedural shaders is a relatively new domain and not much work has been done in the area. Traditionally, the creation of shaders at multiple levels of detail has been a manual process; in this context, shaders authors are required to create simplified shaders to be applied to a particular object completely manually. Examples of this process can be found in [Apodaca and Gritz 2000; Goldman 1997]. A more sofisticated approach is found in [Olano et al. 2002] where a library of low-level building blocks is augmented with hand-written levels of detail and any new shader built on this library will inherit these simplifications. While our system allows the specification of user simplified functions, our system does not rely on them to provide simplified shaders and when present can take better advantage of them since these simplifications are only selectively applied when the error introduced is low.

The work that more closely relates to ours and inspired our own is the one presented by Olano et al.[2003], where texture accesses of a procedural shader are simplified automatically to obtained multiple shader levels of detail. With respect to our work, there are two main differences: first we provide a more comprehensive simplification framework that works on any shader code, second we use a more accurate error metric capable of capturing shader error with more fidelity.

Specialization is another important class of techniques utilized to speed up the execution of arbitrary shaders by precomputing parts of a shader based on its input parameters. Program specialization creates a version of the shader where some parameters are expected to be constants by propagating these constants throughout the code. This technique is supported by our algorithm as well as by the Cg system [NVidia 2004]. On the other hand, data specialization [Guenter et al. 1995] creates specialized versions of shaders by storing in textures results of expressions dependent on parameters that are not expected to vary; our framework does not yet integrate this technique since the need to allocate texture memory requires all shaders in the scene to be optimized at once, while we concentrate on generating simplifications that can be used in any scene.

Another body of work related to ours is the more general area of automatic shader transformations. Examples of it are program optimizations performed by an optimizing compiler, some of which are integrated in our framework. The reader can find a review of these techniques in [Muchnick 1997]. Also of notice is the work by Heidrich et al.[1998] where affine arithmetic is used to efficiently sampling procedural shaders.

3 Shader Simplification

3.1 Problem definition

A shader is a program that computes some aspect of the appearance of a scene. Without loosing generality, we can simplify the discussion by considering only shaders that compute the final color of a pixel. It is useful to think about a shader as a function f that takes as input a set of varying parameters v_i (such as object positions, normals and texture coordinates) and a set of uniform parameters u_i (such as object color and light position) and returns the color C_{xy} of an object at each pixel xy in the final image.

Similar to previous approaches in geometric and shader simplification [Hoppe 1996; Olano et al. 2003], our goal is to derive a sequence of increasingly simplified shaders f_k, starting from an original one f_0, such that at each step in the sequence the difference between f_k and f_0 is as small as possible, but the amount of computation required to evaluate f_k is smaller than that of f_{k-1}. Of particular importance in the definition of this sequence is the form of the metric used to measure the error between shaders.

The difference between two shaders applied to a particular object can be measured by integrating over the image the L^2 distance of the colors at each pixel, which we can write as

$$\varepsilon = \int\int_{xy} \left\| f(\bar{v}_i, u_i) - f'(\bar{v}_i, u_i) \right\|^2$$

where $\bar{v}_i$is the value of the varying parameters for the pixel at coordinate xy. While this metric works well in most cases, it ignores the fact that uniform parameters are often varied during rendering since they can be animated or changed to allow the same shader to be applied to different objects. To capture these uses we would like to integrate our previous difference metric over the domain of each uniform parameter, thus giving us

$$\varepsilon = \int_{U_i} \int\int_{xy} \left\| f(\bar{v}_i, u_i) - f'(\bar{v}_i, u_i) \right\|^2$$

where U_i is the domain of the uniform parameters. This also captures view-dependent effects, since camera position can also be expressed as a uniform parameter. Variations over different varying parameters and texture inputs could also be taken into account, by averaging the error of the shaders as applied to different meshes and texture sets. In this paper we ignore these variations since we concentrate on simplifying shaders applied to a particular mesh.

3.2 Algorithm Overview

The input of our algorithm is a shader written in a subset of the Cg high-level language [Mark et al. 2003] and a set of intervals that defines the domain of each uniform parameter. On initialization, the original shader is transformed into a high level abstract syntax tree (AST) representation that maintains explicitly the branch, loop and function-call constructs expressed in the original shader [Muchnick 1997].

In order to generate the next shader in our sequence, we would like to find a new AST whose difference from the original is as small as possible but whose computation requirements are smaller. Similarly to [Olano et al. 2003], this is done by applying a set of simplification rules to the AST that generates a set of candidate ASTs. For each of these candidates, we then estimate the difference between the shader and the original one, picking the candidate that has the smallest error in the set. The sequence is generated by repeatedly

applying this procedure until the final shader is simply a constant, whose AST has only one node.

In this process the two most expensive operations are the evaluation of the error and the generation of the candidate ASTs. To estimate the former we randomly choose a set of values for the uniform parameters defined in their domain, and compute images for the original shader and the current candidate. We then average the difference of the colors of each corresponding pixel in the two images. In this sense, our error estimate is computed by Monte Carlo integration of the average image difference in the domain of the uniform parameters. We chose to use Monte Carlo integration instead of quadrature rules since typical complex shaders have often tens or hundreds of parameters and possibly various discontinuities in the domains of each; in these circumstances Monte Carlo techniques tend to provide more robust estimates. It should be noted that our error metric is very different from the one presented in [Olano et al. 2003], where differences between the results of single nodes in the AST were considered good estimates of the overall error of a shader. This approach has the advantage of requiring no image evaluation to estimate the error, becoming much faster than our algorithm. Unfortunately, we found that using such a metric did not correlate well with image differences in complex shaders for our simplification rules; on the other hand using image differences directly allows us to capture shader error more accurately which is crucial for our application.

3.3 Simplification Rules

In our algorithm the generation of simplified candidate ASTs is obtained by applying a set of rules to the original AST. These rules are defined by two parts: a pattern that is matched to a node of the AST and an expression that defines a new subtree based on the original one. The semantic of pattern matching is similar to the one found in functional languages such as ML [Paulson 1996] or Mathematica [Wolfram 2003] to define functions and makes it very easy to define new rules to be applied for simplification. Once these rules are defined, candidates are generated by creating a new AST for each possible match of each rule, where the subtree starting at the matched node is substituted by the one created by the expression in the rule.

Let us illustrate the process with the following example.[1] Given the expression

$$x + y \cdot z$$

and the rules

$$a+b \rightarrow a \qquad a+b \rightarrow b \qquad a \cdot b \rightarrow a \qquad a \cdot b \rightarrow b$$

our algorithm will generate the following candidates

$$x \qquad y \cdot z \qquad x+y \qquad x+z$$

by testing all of the rules against every node of the AST and applying it only when the pattern in the rule matches the node.

The list of rules used by our simplification algorithm is presented in Table 1. We can observe three different classes of rules. The first one considers binary operators where one argument is a constant and defines the other argument as the simplified expression. This is the simplest of the simplification rules and produces approximated

[1]Remainder: during the remain of this paper we will use the form $pattern \rightarrow expression$ to define a rule that will match the *pattern* to a node in the AST and substitute it with the *expression* provided. We will also use the form $a \Rightarrow b$ to indicate the simplification of a to b.

a. Simplification of binary operators
$e \otimes c \rightarrow e$
$c \otimes e \rightarrow e$

b. Simplification of for loops
$for(i = c_b; i < c_e; i++)block \rightarrow$
$for(i = c_b; i < c_e; i++)if(i \neq j)block$

c. Simplification by average substitution
$e \rightarrow average(e)$

where c, c_b, c_e are constants, e is an expression, $\otimes$ is a binary operator and j is a value between c_b and c_e.

Table 1: Simplification rules.

code by removing some operations. The second class of rules simplifies for loops that iterate over a range, by executing the loop for all but one of the values in the range. We found this rule to be particularly useful since many shaders loop over ranges to compute lighting or to generate patterns [Apodaca and Gritz 2000][Ebert et al. 1998]. The third class of rules simplifies shaders by substituting an entire subtree in the AST with its average value which is estimated by Monte Carlo integration over the range of the shader's input parameters.

These simplification rules were chosen to try to minimize the number of possible rules while covering as many simplifications as possible, allowing a reduction of the number of candidates at each node, without reducing the space of candidates to be searched. For example, let us consider a possible rule

$$a+b \rightarrow a$$

where b is not a constant. This rule is not necessary since it is equivalent to the subsequent application of rules 2 and 1, generating

$$a+b \Rightarrow a+average(b) \Rightarrow a$$

Also note that the rule $a+b \rightarrow a$ will never be selected by our algorithm unless b equals zero, thus making this rule not useful in our context. Another example of interest could be the definition of rules that drop a branch of an $if/else$ statement:

$$if(c)\ block_1\ else\ block_2 \rightarrow block_1$$
$$if(c)\ block_1\ else\ block_2 \rightarrow block_2$$

These rules are also not necessary since they can be expressed by the substitution

$$if(c)block_1\ else\ block_2 \Rightarrow if(average(c))block_1\ else\ block_2$$

followed by the removal of dead code.

It should be noted that for our algorithm to create a sequence of simplified shaders, the only requirement that a rule has to have is that it creates a simplified shader that needs less computation to be evaluated than the original one. It is not necessary that the error introduced is smaller since, if it becomes too large, the shaders generated by this rule will not be accepted during the creation of the sequence. This makes it very easy to add new simplification rules to our framework.

One issue that arises in the case of extreme simplification is that errors might accumulate in the sequence in such a way that the last simplified constant might not be the average of the original shader.

To avoid this issue we would like to only consider candidates whose average is guaranteed to be the one of the original shader. To do this we renormalize our candidates by subtracting their average and summing the one from the original shader following

$$f_c = s_c - average(s_c) + average(f_0)$$

where s_c is a candidate generated by the simplification rules and f_c is its normalized form that will be considered during the simplification process. This normalization not only has the benefit of guaranteeing convergence, but also allows us to consider a larger class of candidates for the simplification.

3.4 Treatment of reused values

Of particular interest in the context of creating simplified shaders is the treatment of expressions whose results are used multiple times. An example of this case would be

$$x := e; \qquad y := f_1(x); \qquad z := f_2(x);$$

where the value of e is used to compute y and z respectively. If a simplification is applied to e, the error in the value of y and z might be different. On the other hand, we could duplicate the AST of the expression e and simplify the copies independently, thus reducing the overall error at the price of more computation. In our case, we decided to simplify e directly during the generation of a sequence to avoid incurring the extra cost of evaluating the expression multiple times.

A similar case is the simplification of function bodies that are evaluated using different parameters. An example of this case is the following

$$y := f(e_1); \qquad z := f(e_2);$$

As before, we have the option of simplifying the body of the function once or to duplicate it and simplify it separately for each invocation. In our case, we decided to always duplicate simplified functions, since in most cases, the only cost associated with the duplication is a slight increase in code size that did not have negative effects on computation time in our tests.

3.5 Pruning the Search Space

Applying the simplification rules can generate two kinds of candidates: lossy simplified shaders and lossless optimized ones. The former are true simplified shaders, while the latter are typically the results of transformations applied by an optimizing compiler. The introduction of optimized shaders in the sequence does not have any benefit since we are relying on an optimizing compiler to use our shaders. Another reason why we would like to avoid exploring these shaders is that optimizations can be performed without the need to estimate error, thus speeding up the algorithm substantially [Muchnick 1997]. It should be also noted that detecting optimizations cannot be done only by examining the error estimates since these might be inaccurate, for example, when computing small images or if the parameter space is not explored well. For a more detailed discussion the reader is referred to [Olano et al. 2003].

In order to reduce the number of optimized shaders in our sequence, we perform various optimizations on a shader before applying simplification rules. While this process cannot prevent completely the explorations of optimized shaders, it limits strongly the number of candidates whose errors need to be evaluated at each step as well as the number of steps required to generate the full sequence. In this sense, introducing an optimizer before the simplification of each shader in the sequence can be thought of as pruning the search space of the simplification algorithm. Our system implements the following optimization algorithms in its optimizer: constant folding, copy propagation, dead code elimination and program specialization [Muchnick 1997]. While other optimizations are possible, these were easy to implement in our system and already gave us a strong reduction in the number of steps necessary for convergence.

Another advantage of including this optimization step is that while each candidate generated is unique, it can be equivalent to others after optimization is performed. Introducing an optimizer gives us a chance to detect and prune these candidates reducing the number of error computations required.

3.6 User Controls on Simplification

The simplification algorithm described so far is fully automatic and already generates high quality simplified shaders. It is possible though that the shader author will want to emphasize particular aspects of a shader especially in the case of extreme simplification. For example, the author might prefer to have a more accurate diffuse response and lose the highlights than to have a surface with simplified diffuse and specular effects.

This behavior can be obtained by the combination of two features. First, we allow the shader author to mark sets of statements as *immutable*, i.e. parts of the AST that cannot be changed by the simplifier. Second, we allow the user to introduce additional rules for the simplifier to consider. These code annotations are part of the input shader and integrated into our language.

While the declaration of immutable blocks is straightforward, handling the definition of new simplification rules might be a bit more cumbersome for shader authors. While our system can support the integration of arbitrary new rules, we noticed that we rarely needed to do so since the shaders generated by the basic algorithm tend to be of high enough quality. Given this observation and the benefits of a straightforward rule declaration, our language only allows the user to specify rules that involves function calls of the form

$$f(\{x_i\}) \rightarrow g(\{x_j\})$$

where x_j is a subset of x_i and f and g are two functions whose body is defined in the shader code. The specification is simple enough that most shader authors can use it easily and, with the addition of some restructuring of the code, can be used to specify most of the interesting rules that we would have liked to add to our system.

These additional rules are particularly beneficial in production environments where large shaders are often created by piecing together calls to low level libraries either by writing the code directly or by using code generators driven by a visual user interface [Pixar 2001; RTzen 2004]. In this case, the author of the low level libraries can provide handwritten simplifications of a specified function as well as its substitution rules. An example of this rule could be one that substitutes *smoothstep* with *step* in the Cg standard library [NVidia 2004]. For each shader our system will then consider these rules and apply them automatically when beneficial for the creation of a simplifier shader. Notice that while this feature might sound similar to [Olano et al. 2002], it has the major benefit of only employing these substitutions selectively when they incur the lowest cost at a particular step in the sequence; by contrast, the cited system might apply the rules regardless of their impact on image quality. This behavior has the additional benefit of allowing shader authors to quickly add new rules without having to worry too much about the generated error since these are guaranteed to only be applied once no other rule would generate a smaller decrease in quality.

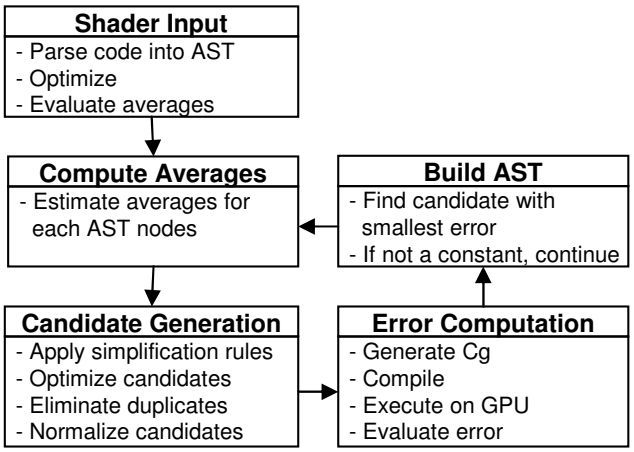

Figure 2: Overview of our system. Source code in Cg language is transformed into its AST form and optimized. From this code, a series of simplified shaders are automatically generated by first computing averages for every node which are then used to generate a series of candidates for simplification by applying simplification rules. The error for each candidate is then evaluated and the shader with the smallest one is selected to be the next simplification in the sequence.

4 Implementation details

A description of our system is in Figure 2. Our algorithm takes as input a fragment shader written in a subset of Cg [Mark et al. 2003] augmented with the constructs to specify simplification hints presented in Section 3 and a set of intervals that define the range of possible values for each parameter, i.e. the domain of the shader. [2] The algorithm starts by converting the shader code in an AST form and optimizing it. Then four steps are required to create a simplified shader.

First, average values are computed for each node that represents an expression in the tree. These estimates are evaluated by Monte Carlo integration over the specified domain of the shader. It should be noted that one evaluation of the full shader provides a value for most of its expression (excluding branching). Thus we can evaluate the average of all the expressions in the AST by simply evaluating multiple times the whole shader.

Once the average values are known, simplification rules are applied to the shader to generate candidates for simplification. These candidates are then optimized and compared to eliminate duplicates. After this step, candidates are normalized such that their averages equal the average of the original shader.

At this stage the error between each simplified candidate and the original shader needs to be evaluated. To do so, we convert each AST to Cg code, compile it and then run it on graphics hardware multiple times corresponding to different values of the uniform parameters chosen to allow Monte Carlo integration of the difference between images. We chose to estimate the error using graphics hardware since shaders can be evaluated much faster than in software, allowing us to increase the number of samples we can use. This is in contrast with the estimation of averages we described previously since in that case the overhead of running on hardware would be much higher providing little benefits but complicating the system substantially.

Finally we choose the shader that has the smallest error amongst the candidates and add it to the sequence. The algorithm is then applied again starting from the computation of the average. It should also be noted that if a simplification candidate has the same error as the current shader, it will be picked by this procedure. Thus error computation can be stopped earlier if this condition is met by any candidates, reducing the number of error evaluation required.

[2] Our system supports most of the Cg language including texturing, branching, for loops, function calls and arrays. The few features that are not supported by our system (such as user defined data structures) can be easily integrated without changing any of the simplification rules.

5 Results

To study the behavior of our algorithm, three shaders were ported to Cg from their original Renderman specification [Apodaca and Gritz 2000] by removing antialiasing code and by implementing the noise built-in function using 3d texture look-ups. These shaders were chosen since they have fairly complicated procedural formulations typical of surface and light shaders in production environments [Apodaca and Gritz 2000; Pellacini and Vidimče 2004]. In particular, the *tiles* shader has a complex procedural pattern that defines its surface color and normal distribution and is illuminated by a simple light source. The *lights* shader shows a diffuse object illuminated by two instances of the complex light shader from [Barzel 1997; Pellacini and Vidimče 2004]. One light from the same formulation is also used to illuminate a procedural pattern in the *wood* shader, combining complex surface and light models. Images from the simplified sequences generated from these shaders can be found in Figure 3 and Figure 4, where different shaders in the sequence were chosen to show particular aspects of our algorithm. These images, together with the simple sequence shown in Figure 1 show that automatic simplification of procedural shaders is possible with our system.

All the simplifications in this section were generated using the basic simplification rules and two additional rules for function substitution

$$\begin{aligned} smoothstep(a,b,x) &\rightarrow linearstep(a,b,x) \\ linearstep(a,b,x) &\rightarrow step(a,x) \end{aligned}$$

where *smoothstep* and *step* are functions defined in the Cg standard library, while *linearstep* is a linear version of the *smoothstep* interpolation. Also all the functions in the Cg standard library where marked as *immutable*. A summary of various statistics regarding the generated sequences for all the examples in this paper can be found in Table 2; timings were measured on an Pentium4 3.4 GHz with a GeForce 6800 GT graphics accelerator while error samples are taken at a resolution of 512 by 512.

All the results in Figure 3 were generated with intervals for all the uniform parameters consisting of single values; in this condition only one sample in the uniform parameters domain is required to compute the error, while 1000 samples in the domain of the varying parameters were used to estimate averages. Under these conditions, the shaders generated simplified sequences of 55/32/43 shaders respectively whose error is graphed in Figure 3, together with the decrease in time spent executing the simplified shaders as measured on a Quadro FX2000 and GeForce FX6800 based graphics accelerators. As it can be noted from the graphs, the error of the simplified shaders increases in the sequence, while the time required to render the image decreases, providing us with the desired trade-off between image quality and render time. Furthermore this trend appear to work on two fairly different architectures showing that our framework is general enough to be applied in various shading scenarios.

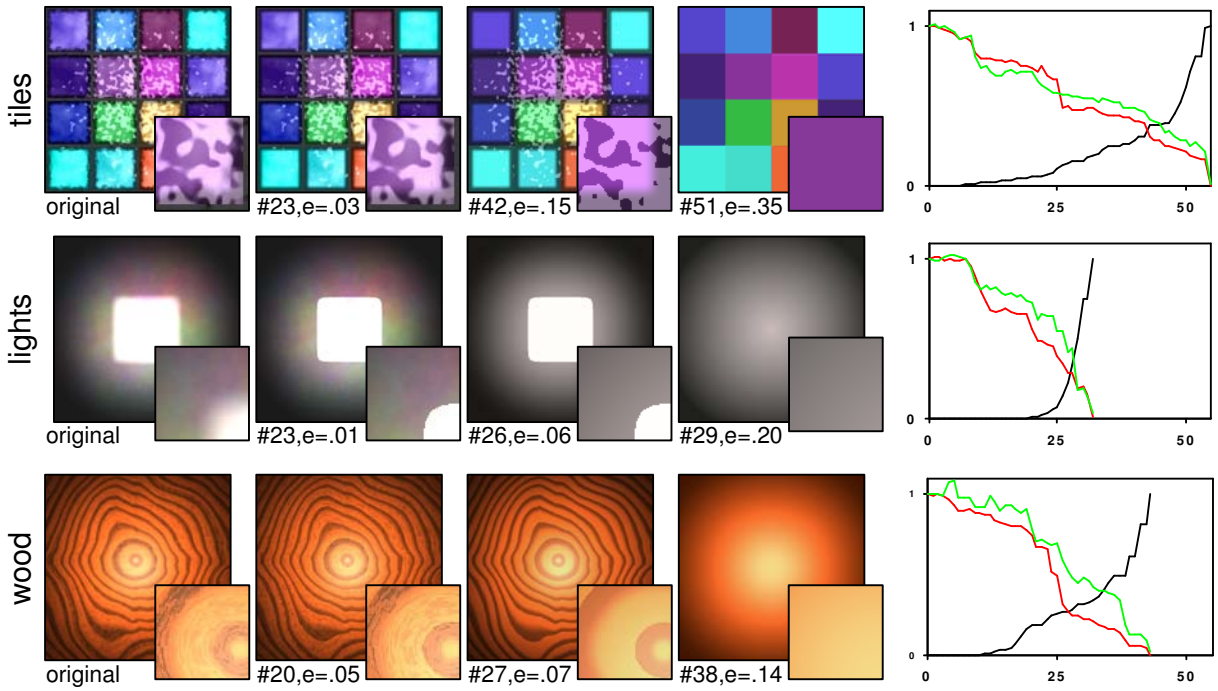

Figure 3: Automatically–generated simplified shaders sequences derived from three original shaders when the domain of each uniform input parameter is a constant (specialized shaders). Original shaders are shown on the right followed by increasingly simplied shaders whose error and position in the sequences are reported below each picture; insets are enlarged 8 times. The graphs on the left show the variation of the normalized shaders' errors (black) and render times (red: QuadroFX2000, green: GeForce6800GT) over the sequences.

It should also be noted that since the domain of the uniform parameters is a point, program specialization is performed on them prior to simplification showing that specialization alone cannot solve the efficiency issue of sufficiently complex shaders. Our framework works well in conjunction with these optimizations using them to aggressively prune the search space for specialized shaders. It can also be noted from these graphs that at the initial stages of the sequence our framework can find simplified shaders with very small error but that gain noticeable improvements in execution speed. As the opportunities for simplification decrease with the size of the simplified shaders, larger errors are introduced quickly in the sequence.

Figure 4 show three series of images rendered using the same sets of simplified shaders but different values for their uniform parameters corresponding to the center and extremes of the intervals used to generate the simplified sequences. For each sequence, a group of uniform parameters was not specialized, allowing us to vary different aspects of each shader. The *tiles* example corresponds to variations of the material generated by a fixed pattern under a fixed lighting; in the *light* case lights are allowed to move and change shape, while the *wood* combines the two by showing a variable pattern under a light that is allowed to move and change shape. For these sequences errors need to be estimated using multiple images to capture variations of the differences in the space of the uniform parameters. In our tests 100 samples were used for error estimation and 1000 for average evaluation. This technique leads to simplified shaders that trade off accuracy across the whole range of uniform parameters as shown, for example, in the second column of the *wood* case. Our system can easily capture this trade-off that would be quite cumbersome for an artist to take into account manually. When compared to their fully-specialized counterpart, these sequences are slightly longer due to the smaller number of possible code optimizations that can be performed on non-specialized examples. These results show that it is possible to automatically simplify shaders whose parameters can be changed at runtime.

During the generations of the simplified sequences, between 32 and 82 shaders were generated after examining only between 1277 and 8873 candidates; this should be compared with the many more possible that where pruned out earlier on and did not need to be tested. Generating the sequences took between 8 and 187 minutes, allocated between candidate generation and error estimation time. The former cost is mostly due to the inefficient match and optimization algorithms we used in our implementation, while the latter increases with the number of samples required for an accurate estimation. Nonetheless the timings show that high quality shader simplification can be achieved efficiently using automated methods.

As reported in Table 2 the percentage of shaders generated by a particular class of simplification rules varies quite a bit. Average and binary operator rules tend to be applied most often since there is a higher number of matches in the shaders; on the other hand, the for and function substitution rules are less often used, but provide useful simplifications respectively for the fractal patterns shown in the *tile* and *wood* examples and for the shape of the lights in the *lights* and *wood* cases.

In general automatically simplified shaders tend to match well our own intuition on how and in what order the original shaders should be simplified. Our algorithm performs simplifications such as the increasing reduction in the details of procedural textures and the elimination of the least important light in a group. Unfortunately there are cases when our algorithm generates shaders that are needed for its convergence, but that do not appear to match our intuition. This happens in the case of extreme simplifications where only a few unsatisfactory matches are available at each step. While stepping through these shaders is unavoidable, the use of better er-

	simple	tiles specialized	lights specialized	wood specialized	tiles	lights	wood
Input Shader							
Instructions (NV30 architecture)	159	329	243	217	329	243	217
Uniform parameters	37	49	49	50	49	49	50
Uniform parameters specialized	All	All	All	All	27	8	14
Varying parameters	8	8	8	8	8	8	8
Simplified Sequence							
Number of shaders	33	55	32	43	82	68	68
Max error	0.27	0.40	0.39	0.28	0.41	0.35	0.33
Candidates tested	2506	5614	1277	3142	8873	6957	7768
Rules							
Binary op. rule	27%	22%	50%	40%	23%	22%	15%
Average rule	48%	60%	41%	42%	62%	41%	72%
For–loop rule	12%	11%	0%	9%	7	0	7%
Function rule	12%	7%	9%	9%	7%	6%	6%
Timings							
Total time	8’	45’	16’	19’	187’	125’	128’
Error computation time	75%	80%	50%	57%	84%	126%	92%

Table 2: Summary of statistics for the simplified shader sequences reported. Images corresponding to this data can found in Figure 1, Figure 3 and Figure 4.

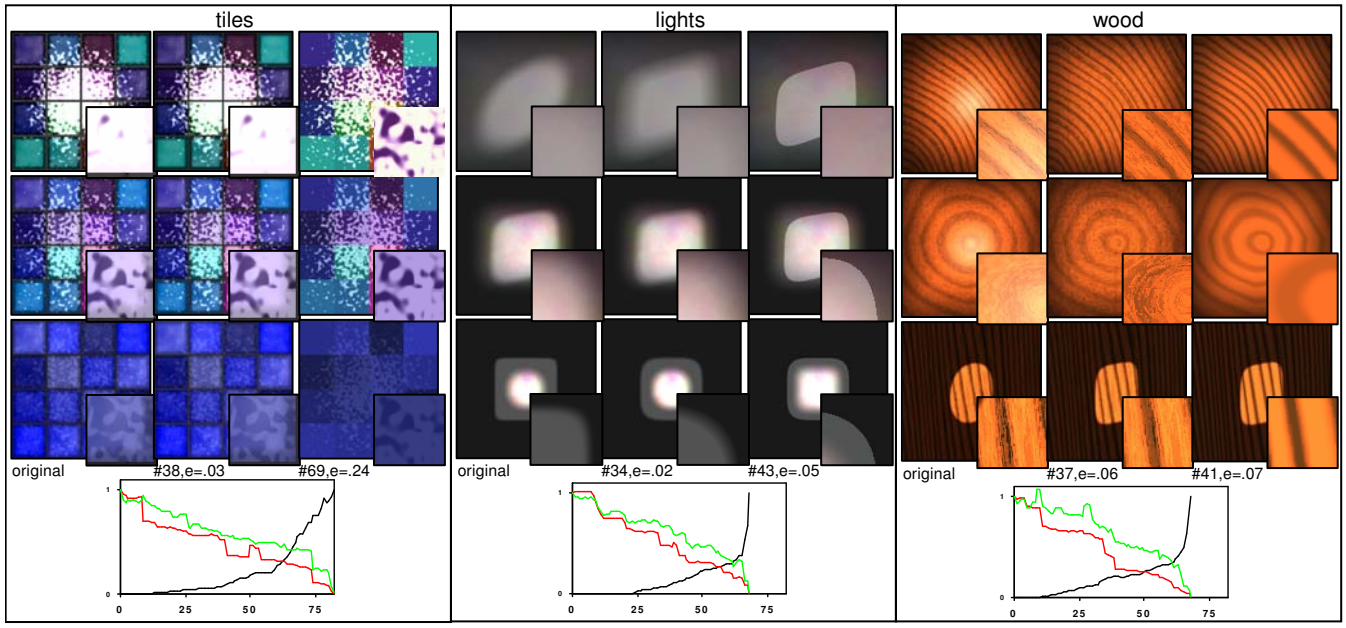

Figure 4: Automatically–generated simplified shader sequences derived from three original shaders when the domain of most uniform input parameters is left to vary. For each sequence, images in each column are rendered using the same shaders shown from left to right with increasing simplification. From these shaders, three images are rendered by choosing values for the uniform parameters at the extremes and center of the domain used for the simplification. For each picture, the error and position in the sequences are reported. Insets are enlarged 8 times. The graphs on the bottom show the variation of the normalized shaders’ errors (black) and render times (red: QuadroFX2000, green: GeForce6800GT) over the sequences.

ror metrics might allow us to automatically detect and remove these unwanted shaders from the final sequences.

Finally Figure 5 shows a demonstration of an important applications of shader simplification in the creation of shader levels of detail where two simplified shaders are smoothly blended together in a manner similar to [Olano et al. 2003]; while we did not investigate this area in detail, this example shows that our simplification framework can successfully be used for shader LODs extending previous approaches to the more complex case of arbitrary procedural shaders.

6 Conclusions and future work

This paper presents a framework for the automatic simplification of complex procedural shaders, where a sequence of increasingly simplified shaders is generated starting from an original shader together with ranges for all of its input parameters. Our approach works by applying simplification rules to a shader to generate a series of candidates, whose differences from the original are measured and used to select the candidate with smallest error; this procedure is repeated until the last shader is a constant. While this automatic pro-

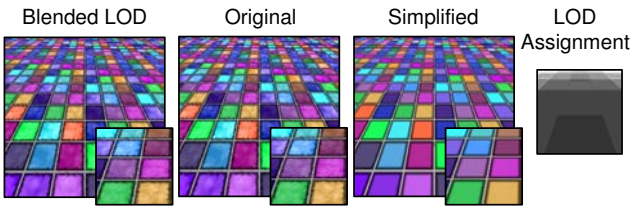

Figure 5: Example of a level of detail application of shader simplification. Two shaders simplified by our algorithm are assigned to patches in the images and blended together to avoid popping.

cedure generates high quality shaders, the user can also tailor the simplification to his specific needs by specifying additional rules to be used during simplification. Our results show that automatic simplification of complex procedural shaders is possible with high quality.

The framework we presented suggests various areas of future research. First we would like to extend our approach to more complex constructs and better error metrics capable of capturing visually-salient errors. Furthermore, it would be beneficial to integrate our approach with techniques capable of automatically splitting vertex and fragment shaders as well as with algorithms that allow for the caching of partial shading computation in lookup tables. It would also be interesting to test our algorithm on shaders edited using visual tools that, building on the principles shown in [Cook 1984], automatically generate the necessary shader code [Pixar 2001; RTzen 2004]; while we believe that our framework would work on this higer level representations, it would require a different set of simplification rules that can match these constructs. Finally we would like to compare our algorithm with other ways to explore the space of possible shaders in the context of level of detail generation, such as algorithms based on search and genetic principles. While in the case of geometric simplification it was found that these greedy approaches work well [Hoppe 1996], it would be interesting to see if this remains true for complex shaders. One final issue that also becomes necessary in order to continue to work in this area is the creation of a testbed of known complex shaders that can be used to compare the performance of different algorithms.

7 Acknowledgments

We would like to thank Lori Lorigo for help with the video and Adam Finkelstein and the referees for their insightful comments. This work was supported in part by Pixar Animation Studios and performed using equipment donated by Intel and NVidia Corporation.

References

APODACA, A. A., AND GRITZ, L. 2000. *Advanced Renderman: Creating CGI fro Motion Pictures*. Morgan Kuafmanm.

BARZEL, R. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools 2*, 1, 1–20.

COOK, R. L. 1984. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, vol. 18, 223–231.

EBERT, D. S., MUSGRAVE, F. K., PEACHY, D., AND WORLEY, S. 1998. *Texture and Modeling: A Procedural Approach*, second ed. Academic Press.

GOLDMAN, D. B. 1997. Fake fur rendering. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, 127–134.

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, 343–350.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, 289–298.

HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. 1998. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics 17*, 3 (July), 158–176.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, 99–108.

KESSENICH, J., BALWIN, D., AND ROST, R., 2003. The OpenGL shading language.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics 22*, 3 (July), 896–907.

MICROSOFT, 2002. Directx graphics programmers guide.

MUCHNICK, S. S. 1997. *Advanced Compiler Design & Implementation*.

NVIDIA, 2004. Cg toolkit user's manual.

OLANO, M., HART, J. C., HEIDRICH, W., AND MCCOOL, M. 2002. *Real-time Shading*. AK Peters.

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

PAULSON, L. C. 1996. *ML for the Working Programmer*, second ed. Cambridge University Press.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, 425–432.

PELLACINI, F., AND VIDIMČE, K. 2004. Cinematic lighting. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed. Addison–Wesley Professional.

PERLIN, K. 1985. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, vol. 19, 287–296.

PIXAR, 2001. Slim documentation. https://renderman.pixar.com/products/tools/slim.html.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, 159–170.

RTZEN, 2004. Rt/shader documentation. http://www.rtzen.com/product/detail,1-6.html.

WOLFRAM, S. 2003. *The Mathematica Book*, fifth ed. Wolfram Media.