A System for Rapid, Automatic Shader Level-of-Detail

Yong He, Carnegie Mellon University
Theresa Foley, NVIDIA
Natalya Tatarchuk, Bungie
Kayvon Fatahalian, Carnegie Mellon University

Abstract

Level-of-detail (LOD) rendering is a key optimization used by modern video game engines to achieve high-quality rendering with fast performance. These LOD systems require simplified shaders, but generating simplified shaders remains largely a manual optimization task for game developers. Prior efforts to automate this process have taken hours to generate simplified shader candidates, making them impractical for use in modern shader authoring workflows for complex scenes. We present an end-to-end system for automatically generating a LOD policy for an input shader. The system operates on shaders used in both forward and deferred rendering pipelines, requires no additional semantic information beyond input shader source code, and in only seconds to minutes generates LOD policies (consisting of simplified shader, the desired LOD distance set, and transition generation) with performance and quality characteristics comparable to custom hand-authored solutions. Our design contributes new shader simplification transforms such as approximate common subexpression elimination and movement of GPU logic to parameter bind-time processing on the CPU, and it uses a greedy search algorithm that employs extensive caching and upfront collection of input shader statistics to rapidly identify simplified shaders with desirable performance-quality trade-offs.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—[I.3.7]: Computer Graphics—Three-Dimensional Graphics and Realism;

Keywords: shader simplification, level-of-detail, real-time rendering, shader optimization

1 Introduction

Modern video game engines are tasked with rendering highly complex scenes and require efficient and high-quality level-of-detail (LOD) systems to achieve their performance goals. These systems consist of simplified shaders used to render far-away objects, a set of distances for switching among shaders and policies for managing LOD transition regions. Many AAA games, in particular those with open worlds or user-customizable assets, feature a large collection of unique shaders that define the diverse surface properties of game objects. Often these shaders are not hand-authored in full or optimized for LOD performance by direct code tweaking by a programmer. Instead, shaders are frequently authored by content creators (such as technical artists) using a set of programmer-created shader components combined in an artist-friendly node graph user interface [Andersson and Tatarchuk 2007; Epic Games 2015a]. During offline asset processing the resulting shader node graphs are compiled down to high-level language shaders with additional state (such as texture or constant inputs) used for rendering. For example, Bungie's AAA title *Destiny* included over 17,000 artist-authored shader graphs compiled to over 180,000 unique vertex and fragment shaders (for a single console platform) [Bungie 2014]. This number multiplies for each shipping platform since target-specific optimizations are employed to efficiently support multiple hardware generations with diverse performance characteristics. Thus, custom manual optimization of shader code for LOD purposes is intractable for game developers. Automated systems are needed to perform this task.

In this paper, we focus on the challenge of automatically generating shader LOD policies for shaders typical of those in modern games. Of course, any automatic system must produce good simplified shaders: that is, the resulting shaders used in the LOD policy should be comparable in visual quality and performance to those authored manually by an expert programmer. Second, to be viable for game design workflows (where assets are continuously created and modified throughout a game's development) such a system must be fast: that is, it must be able to construct a policy in seconds or a few minutes to facilitate fast development iteration times or convenient artist workflows.

We present a system that meets both these goals. Our focus is on generating LOD policies for surface pattern generation components of shaders (the use case in which automatic tools are most attractive), but the system can also simplify full shaders with both surface and lighting computations. The system operates on shaders used in both forward and deferred rendering contexts, requires no additional semantic information beyond input shader source code, and in many cases produces acceptable LOD policies in about a minute. Our design incorporates ideas from prior shader simplification work but contributes new shader simplification transforms (including one that approximates GPU-shader computations with CPU-side processing) and an algorithm that uses greedy search, caching, and extensive collection of statistics about the input shader to rapidly identify simplified shaders with desirable performance-quality trade-offs.

2 Related Work

Our work is inspired by prior efforts to automatically synthesize lower-cost approximations to shader programs. Initial shader simplification efforts considered only local transformations to a single shader stage (intra-stage optimizations) [Olano et al. 2003; Pellacini 2005; Sitthi-Amorn et al. 2011] such as removing arithmetic or texture sampling operations from fragment shaders. We extend these ideas with a new transform: *approximate common subexpression elimination*, a single general operation that subsumes the functionality of many local simplification rules.

More recently, Wang et al. [2014] implemented a simplifying transformation that moved per-fragment shader terms to the vertex processing stage. This capability allowed their system to automate reduction of sampling rate for low-frequency terms, which has long been an optimization employed manually by graphics programmers. We build upon this idea to add a new rate of computation which executes not per frame or draw call [Proudfoot et al. 2001], but only when the value of uniform inputs to the shader change (which can be as infrequently as scene load). This computation rate is missing from prior simplification work, but we find it valuable to avoid "baking" values into shader literals (or fixed constant buffers)

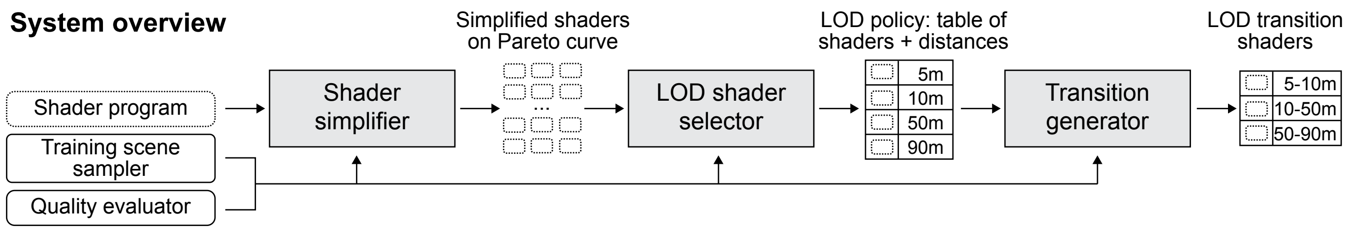


Figure 1: *Given an input shader, a training scene sampler, and shader quality evaluator function, the system automatically generates a sequence of simplified shaders forming a LOD policy.*

during simplification, so that low-LOD shaders are not overly specialized to particular art assets.

This per-uniform-parameter change rate of computation, which we call a *parameter shader* stage, is similar to that implied by the "uniform" rate qualifier in the Renderman Shading language [Hanrahan and Lawson 1990] or the "static" qualifier described by Guenter et al. [1995] since these computations occur once at the time of shader instance creation. While these systems support an initialize-time rate of computation for similar reasons as us (to realize the performance benefits of specialization), they require the programmer to explicitly specify what values are computed at this rate. In fact, complete programmer control over the rate of shader terms was considered a feature by designers as it was thought undesirable for the compiler to make optimizations that modified the output quality or global-scale code structure of shaders [Proudfoot et al. 2001; Foley and Hanrahan 2011]. In contrast, our system automatically determines what terms from input vertex and fragment shaders are acceptable to move to new computation stages—making such decisions for the programmer (even though they *will* impact output quality) is the whole point of an automatic system for generating shader LODs.

Finally, in contrast to prior work which focused on the single task of producing a series of simplifications to a shader, our objective is to generate a complete LOD policy. This includes not only generating simplified shaders but also selecting object distances at which they should be used and managing transitions between LODs to avoid artifacts such as popping. Many aspects of our design focus on compiler performance, since discussion with game developers indicates that to be viable for game design workflows, compiled results must be presented to a shader author in seconds or a few minutes, rather than hours as was the case in prior work based on genetic algorithms [Sitthi-Amorn et al. 2011; Wang et al. 2014].

The search for simplified shaders is a form of program auto-tuning, for which open frameworks for both traditional and variable-accuracy optimization exist [Ansel et al. 2014; Sampson et al. 2015]. We chose to build our own shader simplification framework from scratch to retain full control over the search strategy and to optimize GPU integration for profiling. However, it would be interesting to determine if the techniques described here could be implemented efficiently within the context of these more general auto-tuning frameworks.

3 System Overview

Figure 1 illustrates the end-to-end operation of our shader LOD generation system. The inputs to the system comprise: a shader program P for which a LOD policy is desired; a *training scene sampler*, which defines the space of scene configurations used for gathering performance and quality data; and a *shader output quality evaluator*, used by the system to assess the error introduced by generated shaders. The output of the system is a LOD policy, which includes simplified shaders and distance ranges at which they apply.

In our system, like that of Wang et al. [2014], a shader program (or "shader" for short) specifies the behavior of multiple stages of the graphics pipeline. Specifically, we represent a shader program P as a directed acyclic graph (DAG) where the nodes, corresponding to individual scalar, vector, or matrix operations, are partitioned into a triplet of sub-graphs $\{P_p, P_v, P_f\}$, one for each stage of the pipeline. These sub-graphs represent the operations performed in GPU vertex (P_v) and fragment (P_f) shading stages, along with a *parameter shader stage* (P_p) which includes host-side logic for computing uniform parameter values. Section 4.1 describes how the parameter stage facilitates aggressive shader simplification while still allowing for dynamic choice of uniform parameter values. While the approach described in this paper should extend naturally to additional rates of computation (e.g., per-render-target-sample or per-coarse-fragment [Vaidyanathan et al. 2014; He et al. 2014]), for simplicity our current implementation only supports these three stages.

The training scene sampler must be provided by the user of our system, and includes both a complete scene definition (geometry and textures) and also a sampling strategy for generating any required lighting, material, and camera parameters when evaluating the input shader program and its simplifications. Each parameter to be varied is represented as a distribution and a sampling strategy that generates independent samples from its domain. Currently we use uniform distributions for all parameters.

Since our system might be applied to both forward and deferred shaders we cannot assume semantic or perceptual properties of shader outputs. This task instead falls to the user-provided output quality evaluator, which is responsible for comparing the output of a simplified shader against the "gold" output of the original unsimplified shader. For forward shaders producing RGB color outputs, one might use a simple distance metric on colors, but for a deferred pattern generation shader, producing a *G-buffer* containing multiple material attributes with complex packing, evaluating quality may be more involved. We discuss our specific quality evaluation scheme for deferred shaders in Section 7.

Given an input shader P, training scene sampler S, and shader output quality evaluator function E, we seek to automatically generate a shader LOD policy $L(P, S, E) = \{(P_i, d_i)\}$ that specifies a set of simplified shader programs P_i and transition distances d_i. For each P_i, the policy specifies a camera-to-object distance range (d_i, d_{i+1}) in which P_i is the preferred approximation to P: that is, the highest performing shader with acceptable output quality in that distance range. In addition to maintaining similarity to the input shader P, a desirable policy will reduce visual artifacts such as aliasing and avoid discontinuities when transitioning between LODs.

The proposed system generates a shader LOD policy in three steps (Figure 1). First, a *shader simplifier* explores the space of possible program simplifications to generate many lower-cost approximations to P. The set of simplified shaders lying on the performance-quality Pareto frontier is then provided to the *LOD shader selector*, which employs a more extensive performance and quality analysis to select a small set of shaders that provide the best performance-quality trade off over specific ranges of viewing distances. Finally, to ensure smooth transitions between the selected shaders, the *transition generator* emits new shaders that smoothly blend between the outputs of shaders selected for neighboring distance ranges. The implementation of these three steps will be discussed in detail in the following sections.

4 Shader Simplification

At a high level, the goal of the shader simplifier is to trace out a performance-quality Pareto curve by starting with the original shader and applying one simplifying code transformation at a time to produce a sequence of shaders that decrease in quality, and increase in performance. In generating this sequence, we must address a number of conflicting goals:

- The simplifier should generate the output shaders in as little time as possible, ideally within seconds to a few minutes.
- The generated shaders should lie as close as possible to the true (globally optimal) performance-quality Pareto curve.
- The simplifier should produce enough shaders along the curve to allow the LOD policy generator to find good candidates for a wide range of object viewing distances.

The interplay between these goals leads to a few key strategies in our simplification algorithm, illustrated in Figure 2.

The most important strategy we use to improve simplification speed is to perform a single greedy descent pass with no backtracking; at each step the system applies one transformation to yield a new shader in the search space, then proceeds from there (see Figure 2, black dots). In order for this greedy approach to approximate the true Pareto curve as well as possible, the simplifier must try to make good (well informed) decisions at each step.

In order to make good choices, the simplifier considers a large space of candidate transformations (Figure 2, outlined dots). However, considering many transformations is costly, largely due to the cost of performing GPU rendering using the candidate shaders to evaluate the error they introduce. In order to mitigate the cost of measuring the error of many candidate transformations, the simplification algorithm employs two key strategies.

First, we observe that as a shader is simplified the error introduced by each remaining candidate transformation is often stable. Therefore the simplifier caches and re-uses error estimates across steps in its search, and employs heuristics to determine when to invalidate these cached estimates as simplification proceeds.

Second, the algorithm prioritizes and aggressively filters the list of candidate transformations that it considers at each search step. (Figure 2's gray dots failed to pass this filtering.) One form of filtering is based on static information gathered by a data collection phase executed prior to entering the main search loop. Since this preprocessing is amortized over many iterations of simplification it is tractable to collect extensive (per-instruction) data about the shader that is used to prioritize candidate transformations and also enable advanced simplifying transformations. A second form of filtering is enabled by the goal of producing a dense sampling of the quality axis of the Pareto curve: we consider only candidate transformations that induce the smallest error increase, with the expectation that these candidates may also lead to small steps in the overall performance-quality space.

Before describing the simplification search algorithm in detail in Section 4.4, we describe the program transformations available to

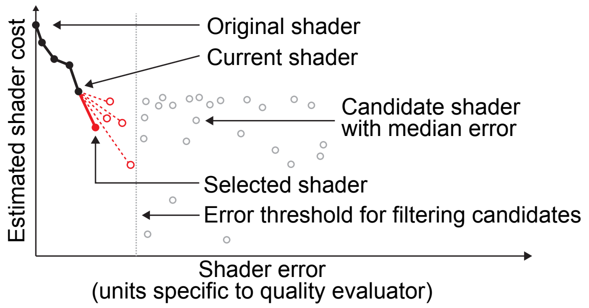

Original shader
Current shader
Candidate shader with median error
Selected shader
Error threshold for filtering candidates
Estimated shader cost
Shader error (units specific to quality evaluator)

Figure 2: *Our greedy search algorithm attempts to follow a performance-quality Pareto curve. At each step we select the simplified shader with the best ratio of change in performance versus error, among candidates below an error threshold.*

simplification, how shaders are instrumented, and how the system measures the error incurred by simplified shaders.

4.1 Program Transformations

The shader simplifier uses three classes of transformations that manipulate the DAG defining a shader program (and the sub-graphs P_p, P_v, and P_f). All transformations operate at the granularity of single nodes in the DAG, which correspond to shader instructions.

Move to vertex stage (`m2v`). This transformation, also implemented by Wang et al. [2014], lifts a per-fragment operation into a per-vertex operation. It is a viable program approximation for terms that vary slowly across the object's surface, since the per-fragment value of these terms is well approximated by interpolation of coarsely sampled results. The transformation moves node n (in P_f) into P_v, and copies any other per-fragment nodes that n depends on into P_v as well. Any unnecessary duplication of computation in P_v and P_f will be eliminated by dead-code elimination optimizations applied as part of compiling the simplified shader.

Approximate common sub-expression elimination (ACSE). Detailed profiling of per-node outputs (as opposed to the final output color) of input shader programs makes it possible to consider advanced approximation strategies. We implement a new simplification transformation that replaces the output of node n_i with the output of n_j (these nodes may be in different stages) when profiling identifies that the outputs of the two nodes are similar in all training scene samples. Since this transformation is similar to the standard compiler transformation of common subexpression elimination, but is applied even when the outputs of the two nodes are not exactly the same, we refer to it as *approximate common sub-expression elimination* (ACSE). If n_i directly depends on n_j, applying ACSE to these nodes is equivalent to binary operator elimination described by Pellacini [2005]. However, unlike binary operator elimination, ACSE is a global transformation that can find simplifications between distant nodes in the DAG, even if the nodes reside in different stages or are not related by dependencies.

Move to parameter stage (`m2p`). This transformation lifts a per-fragment or per-vertex operation into the parameter shader stage, where it is evaluated on the host CPU and provided to the GPU stages as a uniform parameter.

Prior work suggests the use of simplifying transformations such as average substitution ($n \rightarrow$average(n)) [Olano et al. 2003; Pellacini 2005] or Bezier approximation [Wang et al. 2014]. The problem with computing statistics of terms at compile time and embedding the results into the program as immediates (or "baking" values into constant buffers or textures) is that the resulting simplified shaders are highly specific to the test scene used for error evaluation and fail to generalize to novel runtime configurations. To address this issue,

Input fragment shader

```
parameter vec4 paintColor0, paintColor1;
in vec3 normal, view;
out vec4 color;

float fFresnel1 = clamp(dot(normal, view), 0.0, 1.0);
float fFresnel1Sq = fFresnel1 * fFresnel1;
vec4 paintColor = paintColor0 * fFresnel1 +
                  paintColor1 * fFresnel1Sq;
color = paintColor * lighting(...);
```

Output parameter shader (generated from simplification)

```
// input values provided by engine
vec4 paintColor0, paintColor1;

vec4 paintColor = vec4(0.0);
for (int i=0; i<NUM_SAMPLES; i++) {
  int sample_idx = random(i);
  vec3 normal, view;  // unique samples per iteration
  float fFresnel1 = clamp(dot(normal, view),0.0,1.0);
  float fFresnel1Sq = fFresnel1 * fFresnel1;
  paintColor += paintColor0 * fFresnel1 +
                paintColor1 * fFresnel1Sq;
}
// value used as uniform parameter to fragment shader
paintColor /= NUM_SAMPLES;
```

Output (simplified) fragment shader

```
parameter vec4 paintColor;
out vec4 color;
color = paintColor * lighting(...);
```

Listing 1: *Move to parameter stage code transformation.*

we defer the computation of average(n) to a new parameter shader stage that executes at runtime on the host CPU once the values of shader input parameters are known.

To understand this challenge more precisely, consider the per-fragment logic from the CARPAINT shader (Section 7) shown at the top of Listing 1. The code computes the car's paint color as a Fresnel-weighted blend of two application-provided colors `paintColor0` and `paintColor1`. Since subtle variations in paint color are difficult to observe at a distance, a plausible optimization is to disregard the Fresnel effect under these conditions and set the value of `paintColor` to a constant equal to the average paint color over all training pixels (e.g., red). However, the resulting shader would not generalize to use cases where the input parameters `paintColor0` and `paintColor1` were changed to values not used in the training scene configurations (e.g., shades of green). Profiling the shader over a wider range of input color values during simplification (e.g., both red and green input parameter samples) would result in high error since the average paint color (now a shade of brown) would poorly approximate the car's appearance for almost any choice of parameter values.

The move-to-parameter-stage optimization provides the opportunity for aggressive code simplification when per-fragment terms are well approximated by their average for a specific set of input parameters (i.e., their value is largely invariant to surface position), but the values of these input parameters may vary significantly at runtime (and are not predictable at compile time). By generating code to compute these uniform values at runtime, the transformation yields shaders that retain the performance benefits of average substitution and baking, but can still be used in a broad range of runtime contexts.

Similar to the move-to-vertex-stage transform, move-to-parameter-stage moves node n into P_p, and copies the nodes that n depends on into P_p as well. For example, in the CARPAINT shader, since the per-fragment variation in `paintColor` (due to view and normal variation) is found to be insignificant at a distance, the computation of `paintColor` is moved to the parameter shader (Listing 1-center). One important difference in the move-to-parameter-stage transformation is that if the node n being moved (or a node it depends on) has been subjected to approximation (e.g., via ACSE) then the transformation instead copies the original unapproximated computation of n into P_p. We found this somewhat *ad hoc* choice yields higher quality simplified shaders because it avoids unnecessary approximation.

The parameter shader stage is executed at runtime, under engine control, when shader parameter values become available. Our current implementation runs the parameter shader stage at material load time and stores the output in a GPU-accessible buffer (extensions could support additional parameter shader rates, such as in response to dynamic game events, per-draw-call, or per-frame). To robustly estimate the average value of `paintColor` needed as input for simplified fragment stage code (Listing 1-bottom), the parameter shader evaluates `paintColor` at NUM_SAMPLES random vertex positions on the object's surface, each time from a random view position. Therefore, the parameter shader terms `normal` and `view` take on unique values (plumbed from shader vertex or uniform inputs) for each sample. Since `m2p` will only be performed when data collection determines the error introduced by the transform is low, only a small number of samples is needed to robustly compute the average (16 in our implementation).

In the rest of this section, we will use the notation `apply(t,P)` to refer to the program that results from applying transformation `t` to shader program `P`. For example, `apply(m2v(n),P)` is the shader that results from moving node `n` from the fragment to the vertex stage.

4.2 Shader Instrumentation

In order to gather data needed to guide simplification decisions, our system has the ability to *instrument* a shader and tabulate the values computed by a given node. We will use the notation `instrument(P, n)` to refer to the operation that collects output samples from node `n` of shader program `P`.

For fragment-stage nodes, generating an instrumented shader simply requires writing the result of the node to a fragment color output. For vertex-stage nodes, a pass-through fragment shader is used to interpolate the node's output and emit per-fragment data. This choice allows direct comparison of the outputs of vertex and fragment nodes at any covered pixel. Instrumenting a shader by collecting rendered pixel data means that our system does not obtain data from occluded regions and captures one value per node per covered pixel. This is appropriate for our needs, since we focus on opaque materials and shaders without looping control flow. A more general solution might use atomic read-modify-write operations to emit data from the instrumented shader.

The training scene sampler is used to sample input scene configurations (e.g., by sampling shader input parameters) and provide the geometry and texture data for rendering. Each invocation of `instrument(P,n)` collects data for frames rendered from each of these scene configurations.

4.3 Measuring Error

In order to measure the error introduced by a simplified shader, the system instruments the shader as described above, and compares its output to that of the original unsimplified shader at all covered pixels. Although a LOD policy will likely utilize the simplified shader at greater viewing distances than the original, our experiments in Section 7.3 show that the ranking of shaders based on error is largely invariant to the distance at which objects are rendered during error analysis. Therefore, to enable efficient, direct comparison of pixel outputs the simplifier renders images using each shader with the same scene parameters and from the same (up close) viewing distance.

Once shader outputs have been collected for all scene configurations generated by the scene sampler, the user-provided quality evaluator is used to determine the error between the two shaders.

```
def measure_error(P_old, P_new):
  data_old = instrument(P_old, P_old.output_node)
  data_new = instrument(P_new, P_new.output_node)
  return quality_evaluator(data_old, data_new)
```

To avoid compounding of small errors as the simplification search process proceeds, it is important that the error of new shaders be measured relative to that of the original unsimplified shader. The simplifier measures the error introduced by applying a new transformation to an existing shader against the original unsimplified shader (`P_orig`) as follows:

```
def measure_xform_error(xform, P):
  return measure_error(P_orig, apply(xform, P))
```

4.4 Search Algorithm

Listing 2 provides an outline of the search algorithm used to trace the performance-quality Pareto curve. After initial data collection, each iteration of the search filters and ranks a set of candidate transformations, and picks one to apply. The goal is to pick the candidate that gives the best increase in performance relative to the increase in error (the solid red dot in Figure 2), while limiting the time spent evaluating candidates. In the rest of this section, we discuss the design of this algorithm, and the meaning of its parameters (K_1, K_2, K_3) in more detail.

4.4.1 Data Collection

Before beginning the search process, the algorithm first uses the instrumentation ability described in Section 4.2 to collect data about each input shader node. This data will be used to prioritize the transformations considered at each search step. The subroutine `collect_data` at the top of Listing 2 shows the overall operation of this phase. By pulling this work out of the search loop, our system can afford to collect detailed data on a per-instruction basis, which would be impractical to perform at each search step.

Each transformation rule described in Section 4.1 requires different information to be collected during this initial phase. For move-to-vertex and move-to-parameter transformations, the transformation is applied to each node in the original shader, and the error incurred by the simplified shader is measured using the approach described in Section 4.3.

For the ACSE transformation, instrumentation is used to tabulate each node's output at a set of randomly selected pixels for each training scene configuration (outputs at 100 pixels per configuration are recorded by default). The same set of scene configurations and pixel positions is used when tabulating output for each node so that the similarity of any two nodes can be estimated by computing the relative difference (using an L2 norm) between their tabulated outputs.

4.4.2 Proposing Candidate Transformations

A naive search strategy would consider every possible program transformation at every step of the simplification process. This quickly becomes intractable for large shaders because the number of possible ACSE transformations is quadratic in the number of nodes in the shader. To accelerate simplification, each transformation rule is instead allowed to *propose* only a fixed number of candidate transformations at each step (K_1).

```
def collect_data(P_orig):
  {P_p, P_v, P_f} = P_orig
  for each node n in {P_v, P_f}:
    m2p_err[n] = measure_xform_error(m2p(n), P_orig)
  for each node n in P_f:
    m2v_err[n] = measure_xform_error(m2v(n), P_orig)
  for each node n in P_orig:
    acse_samples[n] = instrument(P_orig, n)
  return {m2p_err, m2v_err, acse_samples}

def search(P_orig):
  # collect per-instruction data prior to loop
  data = collect_data(P_orig)

  error_cache = {}
  P_cur = P_orig
  loop:
    add_output_shader(P_cur)

    # allow each rule to propose up to K1 transformations
    # using data collected outside the search loop
    candidate_xforms = propose_xforms(P_cur, P_orig, K1)
    if candidate_xforms is empty:
      break

    # populate cache of measured errors
    measure_and_cache_error(candidate_xforms, error_cache)

    # filter candidates to a subset with low error
    filtered_xforms = select from candidate_xforms
      where error < K2*median_error

    # estimate performance using heuristic
    estimate_performance(filtered_xforms)

    # pick transformation with best bang-for-buck
    t = pick_best_xform(filtered_xforms)
    P_new = apply(t, P_cur)

    # validate that error cache didn't lead us astray
    cached_err = error_cache[t]
    measured_err = measure_error(P_orig, P_new)
    if( abs(cached_err - measured_err) > K3*cached_err )
      error_cache = {}
      continue # restart loop iteration

    P_cur = P_new
```

Listing 2: *Shader simplification search algorithm.*

Since taking small steps in quality is a goal of the simplification process, each rule prioritizes proposing applicable transformations that are likely to increase error the least. These priorities are driven by the data collected by `collect_data`. For the move-to-vertex and move-to-fragment rules, the per-node transformations are ranked according to the error measured when applying the transformation to the original shader. For ACSE, pairs of nodes (n_i, n_j), where n_i will be replaced with n_j, are ranked by the L2 distance between the vectors representing tabulated outputs. (If the nodes have very similar outputs for all pixels of all configurations, replacement of one with the other is unlikely to introduce significant overall shader error.)

The error estimates used to prioritize transformations are based on data collection from the original shader, and may be less accurate at other points in the search space. Increasing the parameter K_1 allows more candidate transformations to pass this filtering step, which may improve the quality of shaders produced at the cost of longer search times.

4.4.3 The Error Cache

Given the bounded number of candidate transformations proposed by each rule, the system can afford to compute a more accurate measure of error for these candidates. That is, rather than continue to use data collected up front by `collect_data` to predict error, it could measure the actual error incurred by applying each candidate transformation to the current shader.

However, we expect the system to consider many of the same candidates at each iteration of the search loop, and in most cases the error that would be introduced by a transformation will not differ much across iterations. That is, on successive iterations the same transformations are considered for only slightly different shaders. Our algorithm takes advantage of this fact by using an *error cache*.

For each candidate transformation, the system checks if it has already stored a predicted error for that transformation in the error cache. On a hit, the cached error is used, even if it was computed as part of an earlier iteration. On a miss, an error value for the transformation is computed by applying the transformation to the current shader (via `measure_xform_error`); this error is stored in the cache so that it may be used on future iterations. (Section 4.4.5 describes how the error cache is frequently invalidated to avoid use of out-of-date error values.)

After this step, every candidate transformation is associated with a measured error (although in some cases the measurement may be "stale"), making it possible to directly compare the error introduced by candidates from different rules (e.g., move-to-vertex and ACSE). At this point, the algorithm further filters the list of candidates to include only those below a certain error level. In our implementation, candidates are retained when their error is less than K_2 times the median error in the candidate set (this simple predicate automatically adjusts the error cutoff based on the distribution of available candidates, while remaining robust to outliers).

4.4.4 Estimating Performance

Once the list of candidates has been filtered down to a small number of transformations likely to yield a small step in error, the performance gain for each candidate is estimated using a heuristic performance model. Using a heuristic to estimate shader performance (rather than running the shader to measure its actual performance) not only dramatically improves the speed of the simplification algorithm, but is critical for getting predictable and stable performance results required by our greedy search algorithm. In many cases, a single simplification step will eliminate only one or two instructions, a change with insignificant impact on real measured performance (in the noise, even after long-duration performance profiling). The inability to robustly provide performance gradients can cause the algorithm to make bad decisions which, without the ability to backtrack, it cannot recover. (For stochastic or backtracking search algorithms the ability to accurately estimate local performance gradients may matter less.)

Our current implementation uses a simple performance model that captures the broad intuition that eliminating shader instructions is good, and it is better to eliminate expensive instructions and those that execute at a higher rate. The model weighs the cost of DAG nodes by the number of scalar instructions needed to perform them (e.g., a `float4` addition incurs cost 4) and assigns a cost of 100 units to texture operations. Since per-vertex and per-fragment computations are executed at different rates based on triangle size and early occlusion statistics, and have unequal impact on performance due to pipeline load balancing and scheduling, we compute the total cost of the entire shader program as a weighted cost of the vertex (C_v) and fragment (C_f) stage subgraphs, plus a penalty term for the size of the vertex-fragment interface (N_{var}):

$$C_{\text{total}} = 0.3C_v + 0.7C_f + 10N_{var}$$

Parameter shader stage costs (C_p) are given zero weight since they are executed only at scene load. In practice we find this simplistic model with empirically determined weights to be acceptable to guide search. Of course, more accurate models of GPU performance could also be used if available.

Note that before evaluating our performance heuristic on a shader, the system applies standard optimizations such as constant propagation and redundancy removal, including dead code elimination and (non-approximate) common subexpression elimination. These operations are applied to the full shader DAG, and thus exploit inter-stage optimization opportunities. This helps ensure that performance estimates reflects subsequent opportunities for optimization that are created by transformations like ACSE.

4.4.5 Picking and Validating a Candidate

Once the performance of each remaining candidate transformation has been estimated, the algorithm selects one transformation to apply. Multiple candidates may be Pareto optimal, so our system selects the one that provides the best "bang for buck"; that is, the transformation providing the greatest gain in (estimated) performance divided by the increase in (measured and cached) error.

Because the selection process was influenced by previously cached error data, which may be stale, the selected transformation might have introduced more error than expected. As an example, consider a shader where a vector is normalized in the vertex stage, and then normalized again in the fragment stage before being used in lighting computations. The error introduced by eliminating either of the normalizations will likely be small, since normalization is idempotent. However, once one normalization has been removed from the shader (perhaps by ACSE) the error introduced by eliminating the other might now be much greater.

To avoid introducing high-error transformations due to stale error cache data, our system computes a fresh (uncached) error measure for the transformation it is considering applying and compares it to the cached error used during selection. If the relative difference between the measured and cached errors is over a threshold K_3 (if the cached error estimate is shown to be inaccurate), rather than apply the selected transformation, the system invalidates the contents of the error cache, and restarts the current iteration. This invalidation can happen at most once per transformation per iteration, since if the same transformation is selected again the error cache will be up-to-date.

5 LOD Shader Selection

Given the collection of shaders produced during simplification, we seek a LOD policy that specifies which shader to use when rendering the object at a specified distance. Ideally, this policy should prescribe the lowest cost shader that delivers acceptable output quality at a given viewing distance. However, in practice it is desirable to avoid the CPU overhead of frequent shader changes and small draw batches (e.g., when rendering collections of objects) by using only a small number of unique shaders. Minimizing CPU overheads is particularly important since shader LOD presents the opportunity for the GPU to draw more objects in a frame.

Since the simplification process from Section 4.4 only estimates shader performance using a heuristic model, the first step of policy generation is to measure the actual performance and image quality of candidate shaders at specific viewing distances. We choose to perform these measurements at two distances (spanning the object's anticipated viewing range) to produce near- and far-viewing-distance Pareto curves.

Optimal shaders are typically grouped in clusters on the Pareto curve; removing a single instruction will often have little impact on a shader's performance or quality. Therefore, we select a small set of shaders from each curve (10 in our experiments) that are well distributed over the actual performance range. To do so, we evenly divide the actual performance range of the candidate shaders into 10 intervals, and choose the shader with minimum error from each interval.

From this small set of candidate shaders we construct a LOD policy by sweeping from the closest to farthest distance in the viewing range and, at each distance, select the lowest-cost shader that delivers acceptable image quality. To produce policies that utilize only a small number of unique shaders, the algorithm ensures the performance benefit of including a new shader in the policy is sufficiently large to overcome the overhead of an additional shader switch.

```
for each distance d from d_near to d_far by d_step:
  orig_perf = measure_perf_at_dist(P_orig, d);
  for each shader P_i:
    error = measure_error_at_dist(P_i, P_orig, d);
    score[P_i] = 0
    if err < tolerated_err(d):
      score[P_i] = orig_perf - measure_perf_at_dist(s, d);
    if P_i != shader selected at previous distance:
      score[P_i] -= SWITCH_PENALITY;
  shader[d] = shader with maximum score
```

The algorithm uses the subroutine *tolerated_err*, given below, which defines the acceptable error range of a shader based on a user-controlled *quality factor* $Q \subset [0, 1]$. Q determines how rapidly quality is allowed to degrade at large viewing distances (our system uses Q=0.5 by default).

$$\textit{tolerated\_err}(d) = \left(\frac{d - d_{near}}{d_{far} - d_{near}} \right)^{Q} \times \textit{max\_error}$$

6 Transition Shader Generation

One challenge of any system employing discrete LODs (whether for shaders or geometry) is ensuring smooth transitions between quality levels. Today, one common way to avoid "popping" artifacts during LOD transitions in games is to render the transitioning object twice, each time using the stencil buffer to mask a particular subset of covered pixels. The result is a screen-door dithering effect that blends two LODs over a period of consecutive frames [Fatahalian 2015].

As an alternative, our system can synthesize a single shader that interpolates the output of two simplified shaders based on viewing distance. Given P_1 selected for use at distance d_1, and P_2 for distance d_2, the resulting transition shader $P_{1\rightarrow 2}$ is:

```
float t = (d - d1) / (d2 - d1);
vec4 p1_color = // evaluate P1 ...
vec4 p2_color = // evaluate P2 ...
out vec4 color =  mix(p1_color, p2_color, t);
```

where the interpolation parameter t is determined by the distance d to the rendered object.

While this approach must execute computations for both P_1 and P_2, in practice the additional cost is low because of the similarity of these two shaders. Since the simplification process utilizes transformations that move or remove terms from the DAG, the resulting shaders share many common subexpressions. These common subexpressions are eliminated using standard redundancy removal optimizations. We explored using our simplification framework to further optimize transition shaders via approximate transformations, but found this unnecessary due to the effectiveness of traditional redundancy removal techniques.

Given N transition shaders from the LOD policy generation phase, the final output of the system is a set of N-1 transition shaders and the viewing range for which they apply. Our implementation currently uses transition shaders for all viewing distances, although a more aggressive policy might limit their use to distance ranges around where the selected LOD shader changes (at the cost of requiring more distinct shaders in the final policy).

7 Evaluation

We evaluated our shader LOD system on a collection of shaders for both deferred and forward rendering. The BARREL, COUCH, and ROCK shaders, obtained from the Unreal Marketplace [Epic Games 2015b], perform pattern generation for a deferred renderer. These shaders emit surface normal, diffuse albedo, specular, and roughness terms, packed into a 12-byte G-buffer format. The ROCK and CARPAINT [Oat et al. 2003] shaders are complete forward shaders, comprising both surface appearance and lighting. The shaders contain sophisticated effects common in games today such as combinations of multiple levels of albedo and normal maps (see details such as dirt, fabric seams, and paint flecks in COUCH and CARPAINT), ambient occlusion maps, GGX specular reflection [Walter et al. 2007] and environment mapped lighting.

Unless otherwise stated, all results are obtained by performing data collection and quality evaluation on 10 scene configurations generated by the training scene sampler. Each scene configuration is rendered to a 1024$\times$1024 image. During the shader simplification phase, in forward rendering scenarios, shader error is evaluated as the average L2 difference in surface reflectance for all covered pixels. In deferred rendering scenarios, the quality evaluator accepts G-buffer terms as input, evaluates surface reflectance under a collection of lighting environments (we use 8), and then computes error as the average L2 difference in surface reflectance for each of these environments.

Unlike during shader simplification, in the LOD shader selection stage, the distance at which an object will be rendered with a candidate shader is known. The goal of simplification is not to directly replicate the output of a detailed shader, which at distance may exhibit significant aliasing. Instead the simplified shader should approximate the properly filtered (anti-aliased) result of the original shader used at a distance. To perform this comparison, the output of the simplified shader is compared to a supersampled rendering using the original shader, which is then filtered to 1024$\times$1024 for comparison.

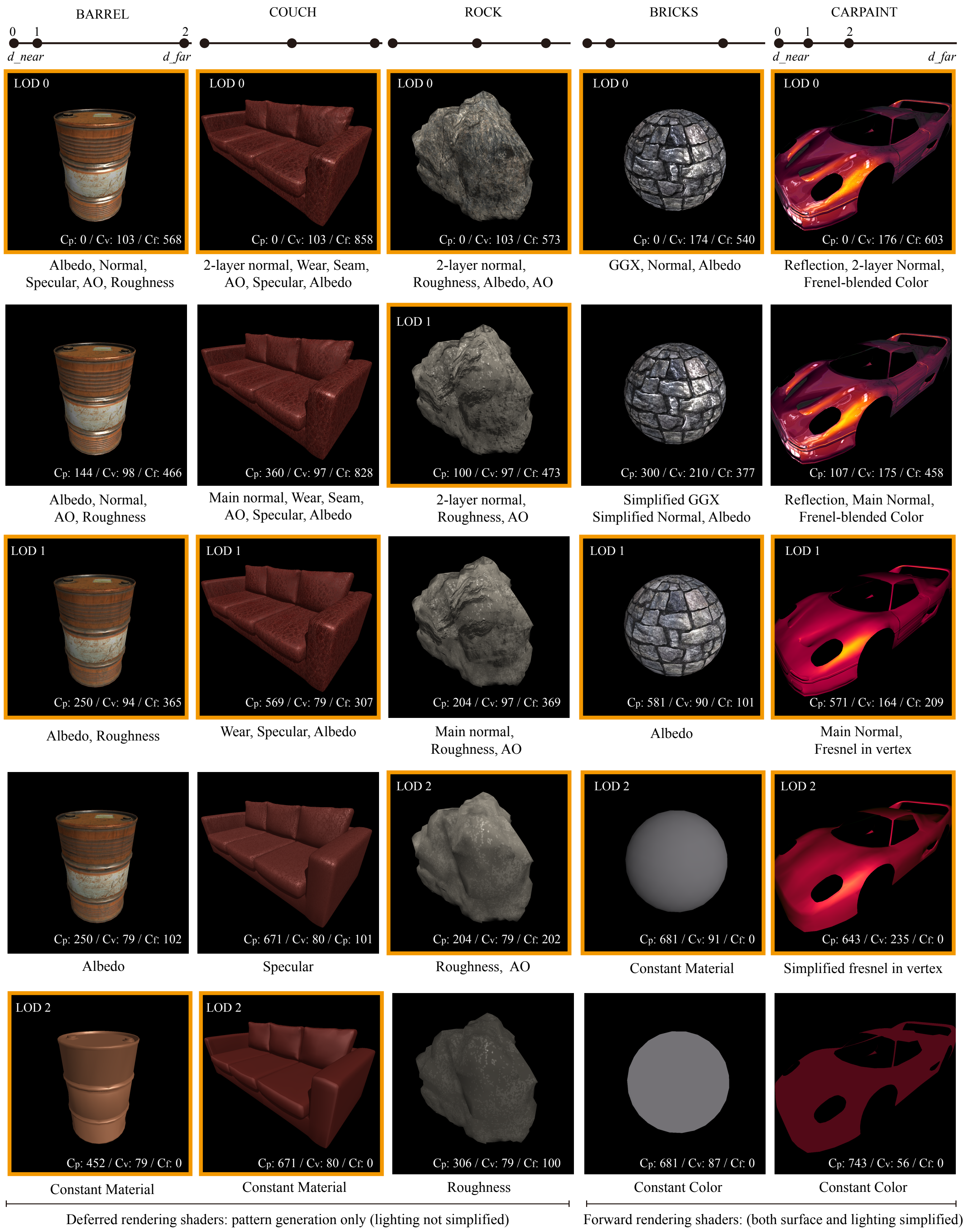

BARREL
COUCH
ROCK
BRICKS
CARPAINT
0
1
2
d_near
d_far
LOD 0
Cp: 0 / Cv: 103 / Cr: 568
Albedo, Normal, Specular, AO, Roughness
LOD 0
Cp: 0 / Cv: 103 / Cr: 858
2-layer normal, Wear, Seam, AO, Specular, Albedo
LOD 0
Cp: 0 / Cv: 103 / Cr: 573
2-layer normal, Roughness, Albedo, AO
LOD 0
Cp: 0 / Cv: 174 / Cr: 540
GGX, Normal, Albedo
LOD 0
Cp: 0 / Cv: 176 / Cr: 603
Reflection, 2-layer Normal, Frenel-blended Color
Cp: 144 / Cv: 98 / Cr: 466
Albedo, Normal, AO, Roughness
Cp: 360 / Cv: 97 / Cr: 828
Main normal, Wear, Seam, AO, Specular, Albedo
LOD 1
Cp: 100 / Cv: 97 / Cr: 473
2-layer normal, Roughness, AO
Cp: 300 / Cv: 210 / Cr: 377
Simplified GGX Simplified Normal, Albedo
Cp: 107 / Cv: 175 / Cr: 458
Reflection, Main Normal, Frenel-blended Color
LOD 1
Cp: 250 / Cv: 94 / Cr: 365
Albedo, Roughness
LOD 1
Cp: 569 / Cv: 79 / Cr: 307
Wear, Specular, Albedo
Cp: 204 / Cv: 97 / Cr: 369
Main normal, Roughness, AO
LOD 1
Cp: 581 / Cv: 90 / Cr: 101
Albedo
LOD 1
Cp: 571 / Cv: 164 / Cr: 209
Main Normal, Fresnel in vertex
Cp: 250 / Cv: 79 / Cr: 102
Albedo
Cp: 671 / Cv: 80 / Cp: 101
Specular
LOD 2
Cp: 204 / Cv: 79 / Cr: 202
Roughness, AO
LOD 2
Cp: 681 / Cv: 91 / Cr: 0
Constant Material
LOD 2
Cp: 643 / Cv: 235 / Cr: 0
Simplified fresnel in vertex
LOD 2
Cp: 452 / Cv: 79 / Cr: 0
Constant Material
LOD 2
Cp: 671 / Cv: 80 / Cr: 0
Constant Material
Cp: 306 / Cv: 79 / Cr: 100
Roughness
Cp: 681 / Cv: 87 / Cr: 0
Constant Color
Cp: 743 / Cv: 56 / Cr: 0
Constant Color
Deferred rendering shaders: pattern generation only (lighting not simplified)
Forward rendering shaders: (both surface and lighting simplified)

Figure 3: *Outputs of automatic shader simplification. Highlighted shaders were selected for use in the final LOD policy.*

	Full-Scene Render Time (ms)					**LOD Policy Generation Time (sec)**		
	No LOD	LOD (no trans)		LOD (with trans)		Simplification	LOD Select	Total
BARREL	2.79	2.12	(1.32×)	2.11	(1.32×)	51.3	25.2	76.5
COUCH	8.92	7.86	(1.13×)	7.95	(1.12×)	114.9	60.4	175.3
ROCK	2.83	2.51	(1.13×)	2.53	(1.12×)	48.6	36.9	85.5
BRICKS	2.07	1.62	(1.28×)	1.64	(1.26×)	60.2	26.7	86.9
CARPAINT	5.62	3.98	(1.41×)	4.07	(1.38×)	95.0	28.0	124.0

Table 1: *Left: For scenes containing uniformly distributed objects (Figure 4), the generated shader LOD policies reduce end-to-end scene rendering time by a factor of 1.1× to 1.4×. Right: compilation time statistics for generating these shader LOD policies. A LOD policy is created for the most complex shader (*COUCH*) in just under three minutes.*

7.1 Simplification Decisions

Figure 3 shows example LOD policies produced for our shaders. Each column shows the output of five shaders produced during simplification. The shaders ultimately chosen for inclusion in a three-shader LOD policy are highlighted in orange. The viewing distances at which the shaders are used in the policy are plotted as dots at the top of each column. Cost estimates for each shader are also given, broken into parameter (C_p), vertex (C_v) and fragment (C_f) stage costs. (Note that stage input/output loads and stores are factored into the model's interface cost component (N_{var}), so C_f=0 implies no arithmetic or texture operations in the fragment stage.) The first three columns are deferred shaders, so simplification applies only to non-lighting terms; all renderings use the same lighting environment. In contrast, for the forward shaders (BRICKS and CARPAINT), lighting is also simplified. These examples were generated using default compiler parameters (K_1=50, K_2=0.15, K_3=0.15).

Manual inspection of generated shaders indicates that despite having no semantic knowledge of the role of individual shader terms, the system makes effective simplification choices that are similar to those in manually-created LOD shaders. For example, the system often first reduces the complexity of specular and normal calculations (e.g., reducing two-layer normal calculations to a single layer: COUCH, CARPAINT), then focuses on simplifying layers of detail maps (wear and seam components of COUCH). Base albedo and normal maps are removed later in the process. One interesting decision was made in the ROCK example, where the shader's base albedo texture is discarded early in the simplification process (before removal of normal, roughness, and ambient occlusion). While this decision may seem unintuitive to a human programmer, in this scenario the compiler's choice works surprisingly well. Note the heavy use of the parameter shader at low LODs, when a term's average is often an acceptable approximation to its value over the entire object. We find code movement to the parameter shader stage to be the most important capability when generating these LOD policies.

While Figure 3 visualizes the output of all shaders up close, the simplified shaders are intended to yield acceptable output for objects rendered at a distance. Since our system tends to select simplified shaders that remove high-frequency terms, we find that they often exhibit less aliasing and are thus preferable to the original shaders at distance. We refer the reader to the accompanying video for a more detailed evaluation of image quality.

7.2 Shader Performance

The primary motivation for shader LOD is to improve rendering performance, while maintaining acceptable output quality. While the results in Figure 3 indicate that simplified shaders are substantially cheaper than the originals in isolation, in real-world scenes we expect to have many objects rendered at different LODs. To assess end-to-end performance in a plausible scenario, we created scenes containing 800 object instances uniformly distributed over a ground plane (Figure 4). Table 1 shows the end-to-end performance of these renderings for engine configurations with and without shader LOD. The performance benefit of using the LOD policy, including the cost of transition shaders, ranges from 1.1× to 1.4×. More importantly, when rendering expensive effects (COUCH and CARPAINT) the use of LOD saves nearly 1 ms of execution time per frame, which is now available for other rendering or game tasks (1 ms can be a significant amount of time in a game frame).

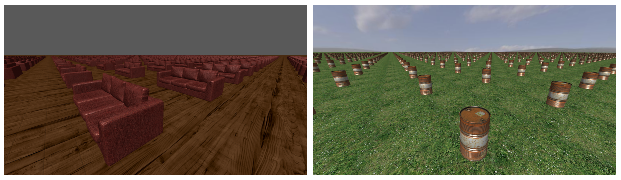


Figure 4: *Example scenes with 800 uniformly distributed objects used to evaluate end-to-end performance of our LOD policies.*

	Data-Collect	Eval Error	Code Gen	Total
Fast	7.9	8.5	0.3	17.8
Default	14.0	99.5	1.3	114.8
No Cache	14.2	432.0	1.6	357.8
All Trans	14.1	615.3	5.4	634.8

Fast:	K_1=50, K_2=0.15, K_3=0.30, 4 configs
Default:	K_1=50, K_2=0.15, K_3=0.15, 10 configs
No Cache:	K_1=50, K_2=0.15, K_3=0.0, 10 configs
All Trans:	K_1=10^4, K_2=0.2, K_3=0.0, 10 configs

Table 2: *Execution time breakdown of various components of the shader simplification process for* COUCH*. Each row provides statistics using compiler settings of increasing compilation cost.*

Table 1 also indicates that the additional cost of using smooth transition shaders during rendering is quite low (no more than a 1% difference in end-to-end rendering time for all examples). Detailed inspection of our transition shaders reveals that the largest overhead incurred by any single transition shader, when compared to the simplified shader from which it is transitioning, occurs for CARPAINT's LOD 1-to-2 transition (20.4% according to our cost model). In all other situations, the overhead of enabling smooth transition shaders is below 12%, and on average is 8%.

7.3 Compiler Performance and Parameter Sensitivity

The right side of Table 1 gives the time required to generate these LOD policies. Policy generation completes in under three minutes for all scenes and, in many cases, under a minute and a half. A majority of time is spent on shader simplification, although the need to conduct robust and accurate performance timings during LOD shader selection incurs non-trivial cost (generating transition shaders has negligible cost). Tighter integration with GPU driver stacks is likely to yield more efficient performance profiling. Table 2 ("default" row) breaks down time spent during shader simpli-

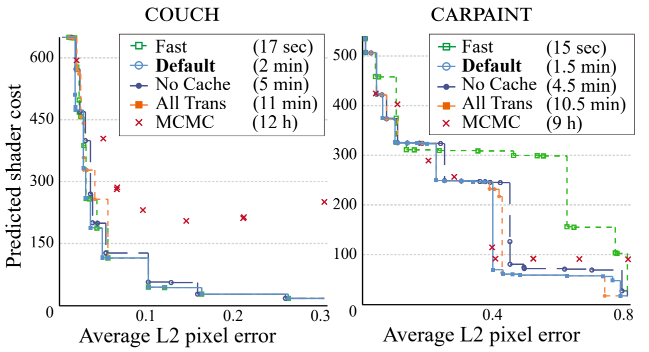


Figure 5: *Our default compiler settings are chosen to produce Pareto curves that are as good as those found using more exhaustive (and more expensive) compiler settings. In many cases (e.g.,* COUCH, *above-left) use of faster settings does not noticeably impact the performance and quality of shaders produced.*

fication into data-collection, error evaluation, and code generation phases. A majority of simplification time is spent running GPU shaders to assess error.

To better understand the run-time and sensitivity of the LOD generation process to compiler parameters, we ran the shader simplifier under a variety of settings: The "no cache" setting sets $K_3 = 0$, disabling the use of the transformation error cache. The "all trans" setting increases K_1, causing all possible transformations to be fully evaluated at each step. Finally, the "fast" setting increases the error cache invalidation threshold K_3 and also reduces the number of scene sample configurations used to assess error from 10 to 4.

As shown in Figure 5, for the COUCH and CARPAINT shaders, we find that more expensive settings than our defaults do not significantly impact the Pareto curve found during simplification. Cached transformation errors remain good approximations to actual errors over multiple search steps, and our transformation prioritization schemes allow for the most beneficial transformations to pass filtering steps. We also found that for many of our examples (see COUCH graph), the "fast" configuration produced as good of policies as our default settings. Although we chose our default parameters conservatively to avoid the need for per-scene tweaking, in practice, acceptable results can often be generated for game-quality shaders in a minute or less.

To further evaluate the results of greedy simplification, we compared our generated shaders against shaders obtained from a non-greedy simplification process based on Markov Chain Monte Carlo (MCMC) search using the Metropolis-Hastings algorithm [Chib and Greenberg 1995]. Even though we allowed the MCMC-based simplifier to run for over 12 hours (red datapoints in Figure 5), it was unable to find shaders with performance-quality characteristics as good as those obtained from the greedy simplification process.

We also evaluated the accuracy of making shader simplification decisions using error values computed at a single (up close) object-viewing distance, as opposed to computing error at the viewing distance the simplified shader is most likely to be used. While it is possible for a shader to have a high error when rendered up close, but low error when rendered afar, we find that, in practice, when a shader A has smaller error than shader B when rendered at a close viewing distance, it also has smaller error when rendered at far distances. Figure 6 provides one illustration of how the Pareto curve obtained by simplification is largely invariant to the object viewing distance used to compute error. Each set of datapoints corresponds to Pareto-optimal shaders as determined by error evaluation at a

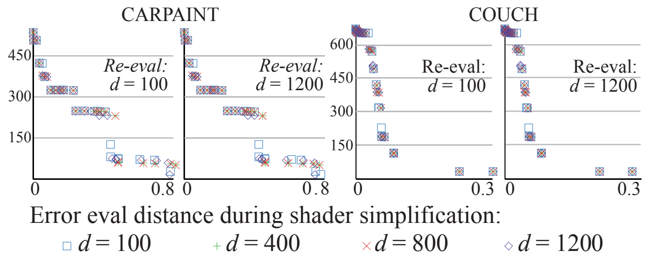


Figure 6: *Pareto curves generated during simplification are largely invariant to the viewing distance used for error evaluation. Each set of data points corresponds to a set of Pareto-optimal shaders as determined by error evaluation at the specified object viewing distance. On each graph, the shaders are plotted according to their error after re-evaluation at the labeled viewing distance.*

single specified distance (d=100, 400, 800, and 1200 world units). On each graph, the shaders from these curves are plotted according to their error as re-evaluated at a different near (or far) viewing distance. The resulting Pareto curves are essentially the same, indicating that measuring shader error at a single profiling distance is a reasonable representation of error observed at other distances.

7.4 Generalization via the Parameter Shader Stage

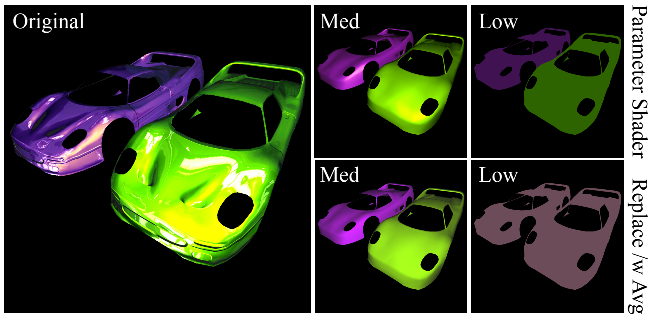
A key aspect of our design is parameter shader stage computation, which allows low-detail shaders to be aggressively simplified without sacrificing the ability to change their input parameter values at runtime. Figure 7 shows the CARPAINT shader used effectively with new car paint colors and the BARREL shader applied to new meshes and texture maps (small images show medium and low-detail renderings). These renderings use the shaders generated from the original LOD policy in Figure 3; no recompilation was required.

In each example, the bottom row of images shows the output of shaders simplified using a replace-with-constant rule from Pellacini [2005], which fails to generalize at low LODs when input parameters from the training phase are baked into the shader as an immediate. Note the grayish color of the car, which is the average of all car colors used during training. While CARPAINT was trained using a variety of paint colors, the novel meshes and textures shown in Figure 7-b were not included in BARREL's scene sampler. As a result, the replace-with-constant rule replaced the albedo texture with a constant brown, which is a good approximation of the texture map used in training (shown in Figure 3) but is not a good approximation for the new textures. Including additional textures in the sampling process is possible, but would increase training time, and high variance in texture values would likely prevent selection of the replace-with-constant optimization. We have found that with a parameter shader stage, training on small number of scene configurations is not only efficient, but the resulting shaders still generalize to assets not available at the time of policy generation.

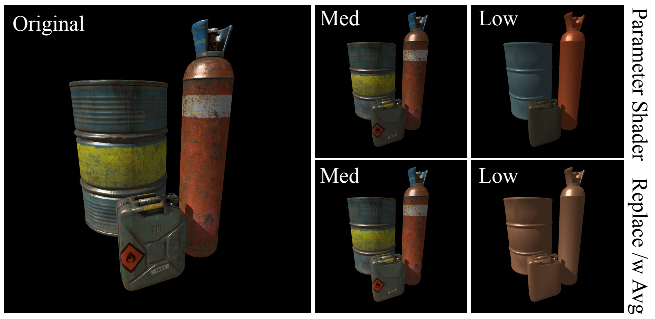
7.5 ACSE Evaluation

To evaluate the utility of the ACSE transformation, we compared Pareto curves obtained from simplification using all transformations in Section 4.1 with those obtained after replacing ACSE with the local operator reduction transformations ($op(a,b) \rightarrow a$ and $op(a,b) \rightarrow b$). These transformations are a stronger version of the binary operation removal rule used by Pellacini [2005], and similar to that used by Wang et al. [2014]. While many applications of ACSE are local operations that replace an operation with one of its arguments, Figure 8 shows that the non-local behavior of ACSE can result in better simplified shaders.

In addition to more efficient shaders, we find that ACSE helps pro-



(a) CARPAINT *shader with different color parameter values. Bottom: baking the average of all possible values into the low-detail shader yields unsatisfactory output.*



(b) BARREL *shader used with new meshes and different textures Bottom: baking the average training scene texture color into the shader prevents runtime usage with novel textures.*

Figure 7: *The parameter-shader stage allows simplified shaders tho generalize to (a) new material parameters and (b) meshes and texture maps not present in the training scene configurations.*

duce cleaner and more intuitive shaders. For example, consider the following fragment shader logic which computes a surface normal from a tangent-space normal map:

```
mat3 tangentMat = mat3(tangent, biTangent, vertNormal);
vec3 tNormal = texture(normalMap, uv) * 2.0 - 1.0;
vec3 normal = tangentMat * tNormal;
```

Suppose an application of `m2p` relocates the tangent space texture fetch and arithmetic to the parameter shader stage, resulting in the following fragment shader:

```
mat3 tangentMat = mat3(tangent, biTangent, vertNormal);
vec3 tNormal = paramTexNormal;
vec3 normal = tangentMat * tNormal;
```

In the code above, the value of `normal` will likely be very close to that of `vertNormal`. ACSE identifies the similarity between `normal` and `vertNormal`, resulting in the following minimal fragment shader code, and a correspondingly simple vertex-to-fragment interface:

```
vec3 normal = vertNormal;
```

8 Discussion

In this paper we have demonstrated a system for automatically constructing shader LOD policies. We find that using an optimized greedy search algorithm, adding parameter binding time processing capabilities (parameter shader), and a simple but general simplification rule (ACSE) yields a system that can process complex game-style shaders to produce policies featuring simplified shaders similar to those created by hand. LOD policy generation completes within a few minutes, making integration into game asset compilation pipelines plausible. While the current system operates on GLSL input with no further semantic knowledge of the shader, a clear next step would be to explore integration of our simplification infrastructure with higher-level shader authoring systems, perhaps using known semantics of key terms to either further optimize search or improve the quality of the resultant shaders.

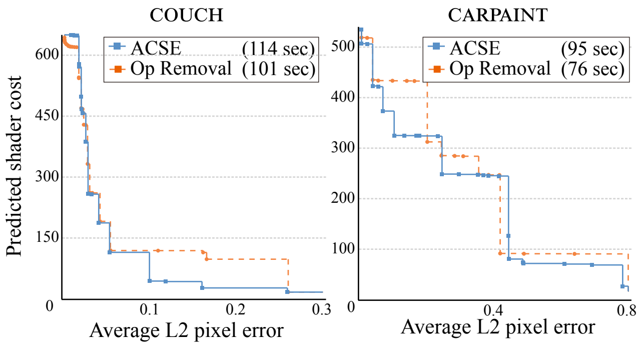


Figure 8: *ACSE is a global operation, which can identify more simplification opportunities then standard binary operator reduction.*

Our experiences suggest a number of ways the GPU software stack could evolve to better support our system or other forms of automatic shader synthesis. First, our current system emits GLSL text for subsequent driver compilation, a process which is both cumbersome and inefficient. We eagerly await lower-level GPU interfaces that can serve as a compiler target for code synthesis systems such as ours. These lower-level interfaces should not only provide an abstract ISA definition to target, but also additional services such as the ability to efficiently query for key static and dynamic statistics about a compiled shader program such as register usage, etc. Innovation in the interface between higher-level software tools and the lower-level driver software stack would not only reduce auto-tuning times but may also enable better results due to the ability to more accurately and efficiently model shader performance.

Our current implementation uses only the GPU's vertex and fragment processing stages, but we strongly believe that the value of a shader simplification system will grow with the ability to optimize over additional computation rates. For example, targeting per-render-target-sample or proposed per-coarse-fragment shading rates [Vaidyanathan et al. 2014; He et al. 2014] is a simple extension that would open up new optimization opportunities for our system. Incorporating tessellation and geometry shader stages is also possible, if they are exposed using a compatible DAG representation, such as in Spark [Foley and Hanrahan 2011]. Parameter stage computations execute at a different temporal rate than GPU pipeline stage computations, and we hope to investigate the utility of multiple host-side rates.

Last, our system uses extensive instrumentation of shaders to prioritize and filter transforms during search and to implement the ACSE transform. We believe there are exciting opportunities for more sophisticated data-driven optimizations such as model fitting [Wang et al. 2014], co-optimization of shaders and geometry, or aliasing detection and removal that would significantly expand the scope and capabilities of automatic graphics code generation systems.

9 Acknowledgments

Support for this research was provided by the National Science Foundation (IIS-1253530) and by NVIDIA. We would like to thank Brian Karis and Angelo Pesce for valuable conversations.

References

ANDERSSON, J., AND TATARCHUK, N. 2007. Frostbite rendering architecture and real-time procedural shading and texturing techniques. In *Game Developers Conference 2007 (GDC)*. Available at `http://www.frostbite.com/2007/04/frostbite-rendering-architecture-and-real-time-procedural-shading-texturing-techniques`.

ANSEL, J., KAMIL, S., VEERAMACHANENI, K., RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARASINGHE, S. 2014. OpenTuner: an extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, ACM, 303–316.

BUNGIE, 2014. Destiny computer game. `http://www.destinythegame.com`.

CHIB, S., AND GREENBERG, E. 1995. Understanding the metropolis-hastings algorithm. *The American Statistician 49*, 4, 327–335.

EPIC GAMES, 2015. Unreal Engine 4 documentation. Available at `http://docs.unrealengine.com`.

EPIC GAMES, 2015. Unreal Engine 4 Marketplace Web Site. `http://www.unrealengine.com/marketplace`.

FATAHALIAN, K. 2015. Tackling the level-of-detail problem through new shading languages and tools. In *ACM SIGGRAPH 2015 Courses: Open Problems in Real-Time Rendering*. Available at `http://openproblems.realtimerendering.com`.

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

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. In *Proceedings of SIGGRAPH 95, Annual Conference Series*, ACM, 343–350.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. *Computer Graphics 24*, 4 (Sep), 289–298.

HE, Y., GU, Y., AND FATAHALIAN, K. 2014. Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Trans. Graph. 33*, 4 (July), 142:1–142:12.

OAT, C., TATARCHUK, N., AND ISIDORO, J. 2003. Layered car paint shader. In *Shader X^2 - Shader Tips & Tricks*, W. F. Engel, Ed. Wordware Publishing.

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

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

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 01, Annual Conference Series*, ACM, 159–170.

SAMPSON, A., BAIXO, A., RANSFORD, B., MOREAU, T., YIP, J., CEZE, L., AND OSKIN, M. 2015. ACCEPT: a programmer-guided compiler framework for practical approximate computing. In *University of Washington Technical Report UW-CSE-15-01-01*. Available at `ftp://ftp.cs.washington.edu/tr/2015/01/UW-CSE-15-01-01.pdf`.

SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. 2011. Genetic programming for shader simplification. *ACM Trans. Graph. 30*, 6 (Dec.), 152:1–152:12.

VAIDYANATHAN, K., SALVI, M., TOTH, R., FOLEY, T., AKENINE-MÖLLER, T., NILSSON, J., MUNKBERG, J., HASSELGREN, J., SUGIHARA, M., CLARBERG, P., JANCZAK, T., AND LEFOHN, A. E. 2014. Coarse pixel shading. In *Proceedings of the Conference on High Performance Graphics*, ACM/Eurographics, 9–18.

WALTER, B., MARSCHNER, S. R., LI, H., AND TORRANCE, K. E. 2007. Microfacet models for refraction through rough surfaces. In *Proceedings of the Eurographics Conference on Rendering Techniques*, Eurographics Association, 195–206.

WANG, R., YANG, X., YUAN, Y., CHEN, W., BALA, K., AND BAO, H. 2014. Automatic shader simplification using surface signal approximation. *ACM Trans. Graph. 33*, 6 (Nov.), 226:1–226:11.