Automated Reprojection-Based Pixel Shader Optimization

Pitchaya Sitthi-amorn[1] Jason Lawrence[1] Lei Yang[2] Pedro V. Sander[2] Diego Nehab[3] Jiahe Xi[4]
[1]University of Virginia [2]Hong Kong UST [3]Microsoft Research [4]Zhejiang University

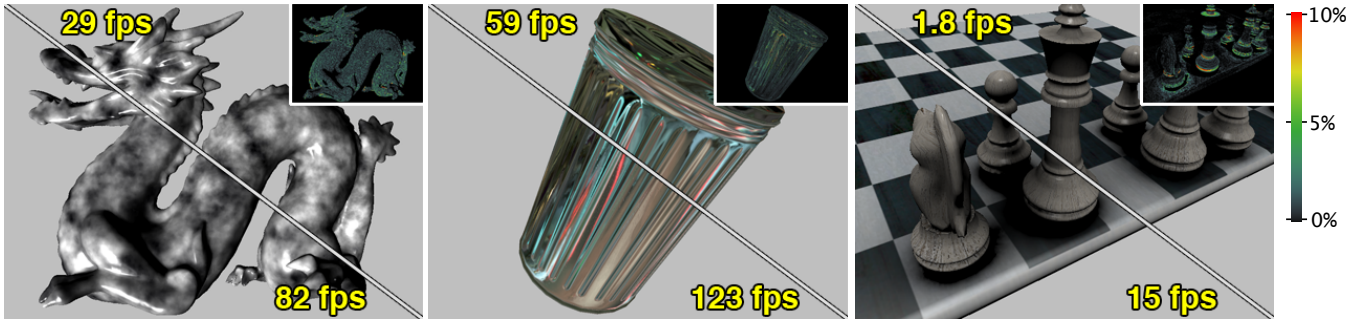

Figure 1: Several optimization results produced with our system. Each image compares (top) an input pixel shader to (bottom) a version modified to cache and reuse some partial shading computation over consecutive frames. Our system automatically selects the intermediate values to be reused and the rate at which cached entries are refreshed so as to maximize performance improvement while minimizing (inset) the visual error injected into the final shading.

Abstract

We present a framework and supporting algorithms to automate the use of data reprojection as a general tool for optimizing procedural shaders. Although the general strategy of caching and reusing expensive intermediate shading calculations across consecutive frames has previously been shown to provide an effective trade-off between speed and accuracy, the critical choices of what to reuse and at what rate to refresh cached entries have been left to a designer. The fact that these decisions require a deep understanding of a procedure's semantic structure makes it challenging to select optimal candidates among possibly hundreds of alternatives. Our automated approach relies on parametric models of the way possible caching decisions affect the shader's performance and visual fidelity. These models are trained using a sample rendering session and drive an interactive profiler in which the user can explore the error/performance trade-offs associated with incorporating temporal reprojection. We evaluate the proposed models and selection algorithm with a prototype system used to optimize several complex shaders and compare our approach to current alternatives.

Keywords: Real-Time Rendering, Reverse Reprojection Cache, Temporal Reprojection

1 Introduction

Procedural shading [Cook 1984; Perlin 1985] is an indispensable tool for modeling realistic surfaces in virtual environments. With the advent of programmable graphics hardware, rendering systems can employ increasingly complex procedural shaders at the pixel level. Figure 1 shows a few examples of custom pixel shaders that can be rendered at or near interactive rates by current hardware.

Modern pixel shaders often perform hundreds to thousands of arithmetic operations, potentially incorporating expensive trigonometric functions and many texture map accesses to produce a final pixel color. As a result, their execution cost can easily dominate the computational budget per frame, exceeding both hidden surface removal and geometry processing. Another important consequence of this increase in complexity is that optimizing pixel shaders by hand has become an exhausting task. Nevertheless, designers are often saddled with tight budgets and considerable manual effort is spent devising the most efficient way to adequately reproduce a desired effect. While there is no substitute for human intuition, there is a clear need for automated tools to help in this process.

Just like a traditional program running on a CPU, a pixel shader can be executed faster with higher-performance hardware or with advancements in compiler technology that can better extract and exploit parallelism in the code. Unique to graphics applications, however, is that shaders can often be more aggressively optimized by *sacrificing accuracy for speed*. Previous techniques have taken advantage of this fact in two different ways: through code simplification and data reprojection.

Given an input pixel shader, the methods proposed by Olano et al. [2003] and Pellacini [2005] automatically generate a sequence of progressively simplified versions. These may be used in place of the original to improve performance at an acceptable level of detail. Data reprojection, on the other hand, exploits the natural temporal coherence in animation sequences by caching expensive intermediate shading calculations performed at each frame which may be reused when rendering subsequent frames. For example, even though a specular highlight moves quickly over a surface due to changes in light position or viewpoint, the albedo usually remains constant. In situations where the surface albedo is expensive to compute (e.g., it requires evaluating a complex procedure as in Figure 1), caching and reusing previously computed values can reduce rendering costs at a minor increase in error.

In general, data reprojection is more flexible than static code simplification, since the former can take advantage of both spatial and temporal coherence, and can adapt to changes in the scene. The method recently proposed by Nehab et al. [2007] runs entirely on the GPU and was demonstrated to provide impressive acceleration results for a number of common effects. However, deciding what to cache and defining a suitable refresh policy are critical for obtaining satisfactory results. Prior systems have left these decisions to a designer, relying on his or her understanding of the semantic structure of the procedure and intuition as to how each choice will

affect final render quality and performance. In many cases, the best candidate for reprojection may have little physical significance and can be easily overlooked among hundreds of choices.

We present a set of techniques to automate the use of data reprojection as a general and practical tool for optimizing procedural shaders. We introduce parametric models of the way caching different subexpressions within a procedure affects its performance and fidelity, which may be trained using a sample rendering session. Additionally, we describe an interactive profiler wherein the decision of what to cache is made according to these models, but guided by user-defined error bounds. This system allows interactive exploration of the error/performance trade-offs involved in using temporal reprojection. We present optimization results for several production shaders (Figure 1) that demonstrate the generality of our approach and compare our work to leading alternatives.

2 Related Work

Prior work on shader optimization has generally taken the form of either code simplification or manual data reprojection. Our approach is the first to combine automatic code analysis with temporal data reprojection. Although we target scanline rendering systems, the analysis and models underlying our approach (Section 4.2 and 4.3) are applicable to ray-based systems as well.

Code Simplification: Methods for simplifying a procedural shader provide a level of detail approximation wherein less complex shaders may be used in place of the original to improve rendering performance. The technique proposed by Pellacini [2005] generates this set of simplified shaders automatically, based on an error analysis of a fixed set of expression transformations, whereas the method developed by Olano et al. [2003] focuses on converting texture fetches into less expensive operations. While simplification provides a clear trade-off between performance and accuracy, the resulting shaders cannot adapt to changes in the inputs in the same way that data reprojection allows. On the other hand, the greater flexibility of reprojection comes at the cost of increased storage overhead. In Section 5, we compare our reprojection system to code simplification techniques.

Data Reprojection: The spatio-temporal coherence of animation sequences has been heavily exploited in both off-line and interactive ray-based rendering systems [Cook et al. 1987; Badt 1988; Chen and Williams 1993; Bishop et al. 1994; Adelson and Hodges 1995; Mark et al. 1997; Walter et al. 1999; Bala et al. 1999; Larson and Simmons 1999; Havran et al. 2003; Tawara et al. 2004]; as well as in hybrid systems that utilize hardware acceleration [Simmons and Séquin 2000; Stamminger et al. 2000; Tole et al. 2002; Woolley et al. 2003; Gautron et al. 2005; Zhu et al. October, 2005; Dayal et al. 2005; Gautron et al. 2007]. The majority of these systems reuse expensive global illumination or geometry calculations such as ray-scene intersections, indirect lighting estimates, visibility queries, etc. One notable exception is the *Shadermaps* system proposed by Jones et al. [2000] which reuses partial shading calculations, although the decision of what to reuse is manual and shaders must be factored by hand. Miller et al. [1998] also propose reusing partial shading data, but rely on a fixed lighting model and a fully manual analysis. Data caching and asynchronous evaluation also play a prominent role in specialized hardware designs [Torborg and Kajiya 1996; Regan and Pose 1994]. Image-based rendering methods [Levoy and Hanrahan 1996; Gortler et al. 1996] and image impostor/billboard techniques [Maciel and Shirley 1995; Schaufler and Stürzlinger 1996; Shade et al. 1996; Aliaga and Lastra 1998; Décoret et al. 2003] can also be regarded as reprojecting shading information into nearby image planes. More recent work has focused on data reprojection methods for real-time applications running entirely on the GPU [Nehab et al. 2007]. A similar reprojection scheme is used by Scherzer et al. [2007] to improve shadow generation and by Hasselgren and Akenine-Moller [2006] to accelerate multi-view rendering architectures. Recently, Sitthi-amorn et al. [2008] introduced a three-pass implementation of temporal reprojection that is better suited for current graphics hardware.

To the best of our knowledge, the problem of automatically selecting optimal subexpressions within a procedural shader for reuse has not been previously studied.

Code Analysis: Our approach employs code analysis techniques similar to those used by Pellacini [2005] for shader simplification. The process of generating an instance of a shader modified to cache a specific intermediate calculation is related to compiler specialization methods studied in graphics [Guenter et al. 1995; Knoblock and Ruf 1996], which have also found use in lighting design systems [Ragan-Kelley et al. 2007]. Unlike this prior work, our focus is on automatic and general methods for identifying optimal shading calculations for reuse.

3 Background: Reprojection Cache

This paper builds on the reverse reprojection cache introduced by Nehab et al. [2007] and later refined by Sitthi-amorn et al. [2008] which is illustrated in Figure 2. These techniques maintain a viewport-sized buffer for both the cache payload and scene depth. The original pixel shader is extended to reproject the generating scene point into the previous frame in order to compute its location in the cache. A cache hit occurs whenever the depth of the reprojected scene point agrees with the corresponding depth in the previous frame's depth buffer, in which case the payload can be reused in computing the final pixel color. Otherwise, a miss occurs and the value must be recomputed from scratch. In either case, the payload and depth buffers are updated and the pixel color is emitted. This technique thus allows reusing shading information at *fixed visible surface locations*, as opposed to fixed locations in the framebuffer. Note that visibility is recomputed anew at each frame and only shading information is potentially reused.

Although a scene point may remain visible across many consecutive frames, its cached value will eventually become stale and should be explicitly recomputed within some predetermined refresh period Δn. The cost of refreshing the entire cache every Δn frames can be evenly distributed by refreshing $1/\Delta n$ of the cache at each frame. Nehab et al. [2007] explore the trade-offs between refreshing the cache along tiled regions and random patterns, concluding that random patterns give less objectionable artifacts, but incur slight performance penalties due to lock-step processing. We use random patterns and force refreshes within 4×4 pixel blocks.

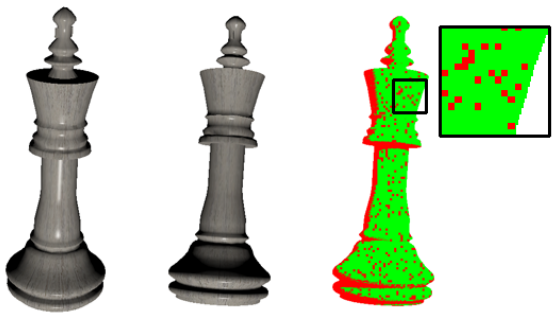

Figure 2: Illustration of the reprojection cache introduced by Nehab et al. [2007]. Their method allows reusing shading data between consecutive frames at mutually visible surface locations, shown in green in the hitmap. Pixels previously occluded or explicitly refreshed are computed from scratch and shown in red. Note that refreshes occur along a random pattern in the framebuffer.

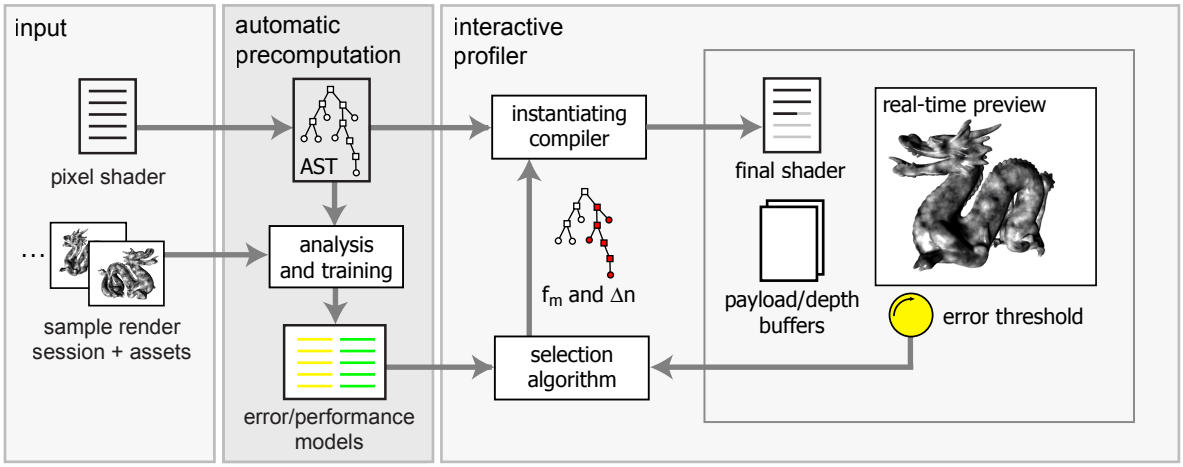

Figure 3: Overview of our optimization system. In a precomputation stage we first compute the abstract syntax tree of an input shader and then train parametric models of the error and performance associated with caching each intermediate calculation over a range of refresh periods. These models drive an interactive profiler in which a selection algorithm chooses the optimal calculation and refresh period based on a user-set error threshold. Our system is able to instantiate new versions of the shader quickly enough to allow the user to interactively explore its error/performance trade-offs.

We build on the three-pass algorithm described by Sitthi-amorn et al. [2008]. The first pass processes cache hits and primes the depth buffer for misses and refreshes. In the second pass, the cache payload is computed from scratch at pixels that were not previously processed, leaving the correct payload value at every pixel in the current viewpoint. A third and final pass computes the final shading using these updated payload values. Sitthi-amorn et al. [2008] show that this three-pass method is more efficient for pixel-bound scenes executing on modern GPUs as compared to methods that require fewer passes but place greater demands on hardware support for multiple render targets and dynamic flow control. Note that our system is not restricted to this implementation, but could instead use that proposed by Nehab et al. [2007].

4 Automated Reprojection

Figure 3 illustrates the main components of our system. The input consists of a pixel shader and a sample rendering session (e.g., geometry and texture assets, camera and animation paths, etc). Our system first computes the abstract syntax tree (AST) [Aho et al. 2006] representation of the shader. From the AST, we can automatically generate a version of the shader modified to cache and reuse an intermediate calculation at a certain refresh period. In a precomputation stage, we use the sample session to fit parametric models of the error and performance associated with every possible reprojection policy. These models drive a selection algorithm that chooses the best cache parameters so as not to exceed an error threshold specified by the user. An instancing compiler generates a version of the shader optimized according to these decisions which is finally bound to a scene under interactive control. Results can be regenerated within a few milliseconds from when the designer changes the error threshold, allowing interactive exploration of the possible optimization choices. The key features of our system are:

- **Ease of use**: fully automatic code analysis and instancing isolate the designer from having to understand or manually modify any source code;
- **Efficiency:** our profiler is able to efficiently analyze production shaders, which often generate hundreds of intermediate values;
- **Interactive feedback:** the designer can interactively select different error bounds and inspect the error/performance trade-offs in the resulting optimized shader.

We focus on shaders that compute the color of non-transparent surfaces, and consider the problem of caching a single intermediate value. Furthermore, once a value and refresh period have been selected by our algorithm they are fixed in the optimized shader.

4.1 Problem Statement

Given the source code to a procedural shader and a representative rendering session, we must automatically identify appropriate subexpressions for caching and automatically transform the input shader to cache and reuse values for those subexpressions.

The key intermediate representation used for both of these tasks is the AST, a static representation of a program, illustrated in Figure 4. The leaf nodes in an AST correspond to program variables and constants and internal nodes correspond to operations (such as a multiplication or function invocation). It is convenient to view the procedural shader code as a single expression that computes a function of its input values. The AST encodes the structure of that expression and the root node represents the final color emitted by the shader. Since we are concerned with caching subexpression values, we need not consider control flow (e.g., conditional branches) explicitly. Richer program representations, such as data dependency graphs or control flow graphs are thus not necessary, but our technique can operate on them if they are available.

Shader inputs may include texture maps, surface normal coordinates, light parameters, animation parameters, texture coordinates, etc. It is convenient to denote the inputs at frame n by a P-dimensional vector X_n. We can then denote the value at each of the M internal nodes at frame n as functions $f_m(X_n)$ of the input vector. Internal nodes commonly represent function evaluation (e.g., $\cos(X_n(3))$) or arithmetic (e.g., $X_n(7)/2$). Lastly, we designate $f_0(X_n)$ as the value at the root node and assume it specifies a vector in the RGB colorspace.

Using a reprojection cache, we can essentially restore a single node f_m in this graph to a previously cached value. This will improve rendering performance whenever the computational cost of evaluating the subtree rooted at f_m exceeds the overhead of supporting the cache. However, if the inputs change (causing the cached value to become stale) or if scene motion forces the cache to be resampled at non-integral locations (leading to reconstruction errors), reusing an internal node may propagate substantial errors into the computed value at the root node.

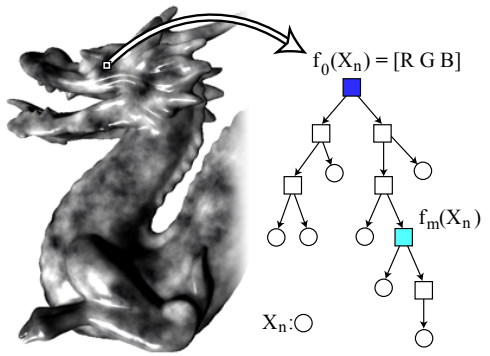

Figure 4: Abstract syntax tree (AST) representation of a pixel shader. The AST encodes the computed function that generates the final RGB output for a given input vector X_n. The root node f_0 represents the final output. Inner nodes such as f_m represent subexpressions of f such as arithmetic (e.g., $X_n(1) \times \pi$) or function evaluations (e.g., texture fetches, $\sin(X_n(3))$, etc.). Those subexpressions are candidates for caching.

The research challenge that this paper addresses is to develop techniques for automatically identifying optimal nodes to cache. We assert that these nodes will exhibit two properties: they can be reused over many frames without introducing significant error into the final shading and their subtree is expensive to evaluate. Because these two factors ultimately depend on run-time user input, our general strategy is to model the *expected* error (Section 4.2) and *expected* rendering cost (Section 4.3) associated with any particular caching policy. These models are trained using a representative rendering session that exhibits typical input patterns. We present an empirical validation of our models in Section 4.4 and describe our selection algorithm in Section 4.5.

4.2 Error Model

Reusing the value of a node cached at a previous frame $f_m(X_{n-\Delta n})$ instead of the exact value computed from the current inputs $f_m(X_n)$ may propagate errors into the final pixel color generated by the shader. By denoting the root value as $f_0(X_n, f_m(X_n))$, we make its dependence on the value of its descendant $f_m(X_n)$ explicit. Note that $f_o(X_n, f_m(X_{n-\Delta n})) \approx f_o(X_n, f_m(X_n))$. Since both values depend on the non-deterministic inputs X_n and $X_{n-\Delta n}$, we treat them as functions of random variables and base caching decisions on the expected value of the L^2-distance between the color produced by the original shader and one modified to cache a node f_m at a refresh period of Δn:

$$\varepsilon(f_m, \Delta n) = P(\text{hit}) E[\| f_0(X_n, f_m(X_n)) - f_0(X_n, f_m(X_{n-\Delta n}))\|], \quad (1)$$

where $P(\text{hit})$ is the probability of a cache hit. Our goal is to compute accurate estimates of $\varepsilon(f_m, \Delta n)$ for each node f_m and a range of refresh periods Δn.

Disregarding for a moment error caused by resampling the cache, the expected value in Equation (1) depends only on the joint probability distribution function $P(X_n, X_{n-\Delta n})$ and the symbolic relationship between the inputs and nodes in the procedure. A simplifying assumption that makes the problem of modeling this equation treatable is to assume that fluctuations in the inputs are stationary, so that $P(X_n, X_{n-\Delta n})$ depends only on Δn. Although this assumption ignores the fact that an interactive session may exhibit input patterns whose statistics change over time (consider, for example, a first-person actor moving between two environments with strikingly different lighting), we found this to be justified in many practical situations. Furthermore, it is possible to derive separate error models from different sample rendering sessions in order to capture these types of higher-level trends.

For the rendering sessions and pixel shaders that we tested, we consistently observed that the correlation between inputs across consecutive frames decayed at an exponential rate. We found a similar trend in the difference in surface colors generated by the original and modified shader: the resulting error always increases with Δn, but at a rate that decays exponentially. The important exception to this rule is when the node has a discontinuous relationship with the root node or appears inside a dynamic block of code. For example, imagine the effect of reusing the result of the termination test in a `for` loop! Fortunately, these cases can be easily detected either in the AST or during training and removed from further consideration.

The remaining factors that influence Equation (1) include the probability of a cache hit (hit-rate) and the nature of the error caused by repeatedly resampling a discrete cache. As before, we found that both of these factors contribute to the final error according to the same exponential trend. That is, both the hit-rate and average reconstruction error increase alongside Δn, but at a rate that decays exponentially. Based on these observations, we model Equation (1) by a parametric function $\hat{\varepsilon}$:

$$\hat{\varepsilon}(f_m, \Delta n) = \alpha_m (1 - e^{-\lambda_m (\Delta n - 1)}). \quad (2)$$

Note that $\hat{\varepsilon}(f_m, 1) = 0$ as required, since this corresponds to refreshing the cache at each frame.

To train our error model we first generate a set of shaders, each modified to cache a valid node in the AST of the input. These are generated automatically and the refresh period is controlled by a uniform parameter. Next, we compute the average pixel error over the entire sample render session between each of these modified shaders and the original at a sparse set of refresh periods (we use $\Delta n \in [2, 10, 18, 26, 34, 42, 50]$). Finally, we fit the error parameters for each node α_m and λ_m to these estimates using a standard downhill simplex search [Nelder and Mead 1965].

4.3 Performance Model

Identifying calculations suitable for reprojection also requires accurate models of the performance associated with caching different nodes in the input. As with error, we focus on modeling average behaviors. The contribution a single pixel shader makes to the overall rendering time will depend on a number of other factors such as the scene geometry, projected size of the shaded surface, etc., which for simplicity we do not consider in our analysis.

Our model assumes that the total amount of time required to render a single frame varies linearly with the number of cache hits and misses, plus some fixed overhead related to issuing and processing

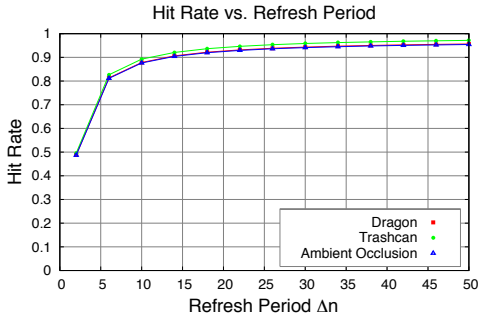

Figure 5: Hit-rate as a function of refresh period for several shaders. We compare (data points) measurements to (solid lines) predictions generated by our model of the hit-rate in Equation (4).

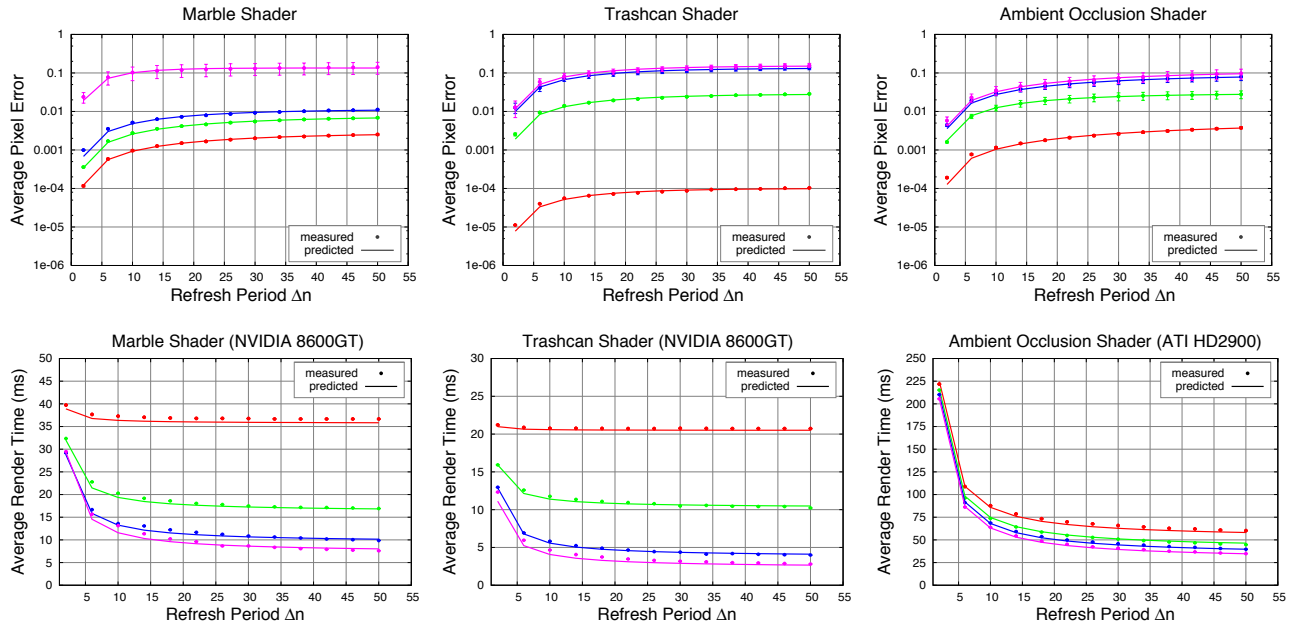

Figure 6: Validation of our error and performance models for several nodes in each shader shown in Figure 1. **Top:** each graph compares (solid line) the average pixel error predicted by our model to (data points) measurements computed from the sample render session over the entire range of refresh periods shown. The error bars visualize the variance in these measurements. **Bottom:** comparison between (solid line) predictions generated with our performance model of the per-frame rendering time of a simple test scene and (data points) measurements of the actual average render time as a function of refresh period. These fits are typical of those we observed in our experiments.

the scene geometry. We estimate these values for each cacheable node f_m using L randomly chosen frames from the sample session. For each frame, we record the total render time t along with the respective number of cache hits h and misses m. We estimate the fixed overhead $E[\text{time\_overhead}]$ and the expected execution time for a cache hit $E[\text{time\_hit}(f_m)]$ and miss $E[\text{time\_miss}(f_m)]$ by solving the following over-constrained linear system:

$$\begin{pmatrix} h_1 & m_1 & 1 \\ h_2 & m_2 & 1 \\ & \vdots & \\ h_L & m_L & 1 \end{pmatrix} \begin{pmatrix} E[\text{time\_hit}(f_m)] \\ E[\text{time\_miss}(f_m)] \\ E[\text{time\_overhead}] \end{pmatrix} = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_L \end{pmatrix}. \tag{3}$$

The average cost of executing a shader modified to reuse some partial calculation will also depend on the total hit-rate. Recall that values are explicitly recomputed in two cases: whenever they are not available in the cache or when their refresh period expires. Figure 5 plots the average hit-rate as a function of Δn for the scenes shown in Figure 1. Based on this analysis, we model hit-rate $\gamma(\Delta n)$ with the parametric expression

$$\gamma(\Delta n) = \mu(1 - 1/\Delta n). \tag{4}$$

Note that the relationship between hit-rate and refresh period is independent of the choice of what to reproject. Therefore, we fit μ using a non-linear regression on the same measurements gathered to train our error model for a single shader. Figure 5 compares the fits to measured data that we obtained for three shaders. Note that the hit-rate never reaches 1.0, which is consistent with the fact that, for a dynamic scene, the probability scene motion will prematurely force a cache miss increases alongside refresh period.

Putting everything together, our model of the average time required to render a single pixel is

$$\hat{r}(f_m, \Delta n) = \gamma(\Delta n)E[\text{time\_hit}(f_m)] + (1 - \gamma(\Delta n))E[\text{time\_miss}(f_m)]. \tag{5}$$

4.4 Validation of Error and Performance Models

We validated our error and performance models using the shaders shown in Figure 1 and further described in Section 5. In each case, we selected four nodes with different error/performance characteristics (these are visualized in Figures 7, 8, and 9, respectively). We also validated these models on different hardware. These fits are consistent with those we observed for the nodes not shown.

The top row in Figure 6 compares predictions generated by our error model to measurements of the average L^2 error computed from the entire sample render session as a function of refresh period. The error bars show the variance in these estimates and provide an indication of the degree to which each node exhibits stationary fluctuations. Note that our predictions are consistently well aligned to the mean.

We similarly validated our performance model by assessing its accuracy in predicting the render time of pixel shaders modified to reproject these same nodes over the same range of refresh periods. Because measuring the time required to render a single pixel is impractical, we instead generated comparisons for a simple test scene for which the number of shaded pixels M is held constant (we simply positioned the model to subtend the entire viewport). For each node and refresh period, we compare measurements of the average per-frame render time to our prediction: $E[\text{time\_overhead}] + M\hat{r}(f_m, \Delta n)$. The bottom row of graphs in Figure 6 show these comparisons. We omit error bars because the variance in these measurements was negligible. As before, we observed close agreement between our performance model and measured data for both NVIDIA and ATI hardware.

4.5 Selection Algorithm and Interactive Profiler

The ultimate goal of our system is to select nodes and refresh periods that offer the greatest performance improvement with the least amount of approximation error. We base these decisions on the models previously described, but leave to a designer the task of determining how to best balance accuracy and speed. Specifically, the user sets a threshold on the average pixel error in the resulting

shader $\varepsilon_{\max}$. Our system simply selects whichever node and refresh period are predicted to provide the greatest performance improvement without exceeding this threshold. Because the relative gains from reprojection decrease with refresh period, we found it useful to clip these to some maximum value (we used 50).

Finally, our system automatically generates a version of the shader modified according to the selected parameters. The subtree of the AST associated with the cached node is replaced by a subtree corresponding to a cache lookup. In practice, we store compiled versions of these shaders on disk and simply fetch them at runtime. This allows regenerating new results within a few milliseconds from when the threshold is changed, allowing a designer to interactively explore different optimization trade-offs. In our prototype system, we present the user with a view of the final shader and the cache payload. Please refer to the supplemental video for a demonstration.

5 Results

We used a Dell XPS equipped with an NVIDIA GeForce 8600GT and an ATI Radeon HD2900 to generate the results in this paper. Our implementation accepts shaders written in HLSL (shader model 4) and allows caching vector-valued nodes with up to 3 elements represented with 16-bits of precision. We evaluated our approach using the three pixel shaders shown in Figure 1 and described below. These were chosen to be representative of modern production shaders and exhibit a range of common shading calculations. The sample render sessions all consist of roughly 150 frames that capture typical input patterns and can be seen in the supplemental video. Table 1 lists statistics of the results that we report, including the total number of instructions in the input, the number of cacheable nodes in the AST, and precomputation times.

5.1 Marble Shader

This shader combines a marble-like albedo, modeled as five octaves of a procedural 3D noise function [Perlin 1985], with a simple Blinn-Phong specular layer [Blinn 1977]. The graph at the top of Figure 7 depicts the pixel error vs. performance for each of the set of cacheable nodes in this shader over a range of refresh periods. These types of graphs are very useful for assessing the degree to which a shader will benefit from data reuse. Paths that appear near the lower left hand region correspond to nodes that are ripe for reprojection as they allow greater reduction in render time for less visual error. Paths toward the upper right hand region are just the opposite: when reused, these nodes not only fail to improve performance, but also introduce significant errors into the final surface color. The vertical dashed line indicates the time required to evaluate the original shader; points to the left of this line represent performance improvements.

For this shader, nodes tend to organize into four clusters. *Cluster A* contains calculations inside the noise function, including its final value. These are more compute intensive since they rely on a long sequence of texture fetches and ALU instructions, but do not depend on camera or light position and are thus ideal for reuse. *Cluster B*, on the other hand, represents nodes that are relatively inexpensive to compute and depend strongly on view and light position. Examples include the calculation of the cosine-falloff and the specular component. These tend to be poorly suited for reprojection since they are inexpensive to compute from scratch and reusing these values would introduce visible artifacts into the shading. *Cluster C* contains nodes somewhere in between these two extremes. They occur after combining the expensive noise function with inexpensive and view-dependent terms such as `noise()*dot(N,L)`. As expected, they offer slightly greater performance improvement than caching the noise terms alone, but at the cost of significantly higher error. Finally, *Cluster D* represents caching decisions that lead to performing an expensive operation twice on a miss. For example, caching the subexpression `(1.0-noise)` inside the expression `(noise + (1.0-noise)*0.2)` would result in evaluating `noise` in the second pass when the payload is updated and again in the third pass when the final color is computed. Sitthi-amorn et al. [2008] describe this "computational overlap" problem as a potential side-effect of their three-pass algorithm. As indicated by their position in the graph, these are poor candidates to cache.

Input Shader	Marble	Trashcan	Ambient
Instructions	419	367	370
Cacheable Nodes	114	105	37
Precomputation	1.5h	1.2h	9.5h

Table 1: Summary of statistics for the results reported.

The images below the graph in Figure 7 depict the nodes selected by our algorithm at four different error thresholds. These thresholds, along with the error/performance path of their associated nodes are also indicated in the graph. Each image shows the cache payload and refresh period selected by our algorithm, along with the final shading resulting from these choices. We also list the peak signal-to-noise ratio (PSNR) along with the resulting framerate. These images (and those in Figure 1) show a single frame in a sequence in which the camera and light-source rotate around the model. Please consult the supplemental video for interactive versions of these comparisons.

For the error threshold ε_1, our algorithm selects the lowest-frequency octave inside the noise calculation for reuse, choosing to recompute the higher-frequency octaves along with the rest of the shading at each frame. Although this policy results in negligible visual error, it also gives a very small performance gain from 29 to 30 frames per second. Similarly, for the second error threshold ε_2, our algorithm chooses a node slightly higher in the AST that corresponds to the sum of the three octaves with the lowest spatial frequencies. Although this calculation can also be safely reused without introducing large errors, it still does not give a significant performance gain. Nevertheless, these two examples help verify and illustrate the behavior of our algorithm. At the third error threshold ε_3, our algorithm selects the complete noise calculation at a refresh period of $\Delta n = 35$. We found this point offered a profitable balance between accuracy and speed, giving a 2.8x improvement in performance at an acceptable level of error. This selection was used in Figure 1. At the highest error threshold ε_4, our algorithm selects the final pixel color. This node will naturally always provide the greatest performance gain, but often introduces visible artifacts, as is the case in this example, since this shader contains strong light- and view-dependent components. Additionally, the difference in performance improvement between ε_3 and ε_4 is small since the noise calculation accounts for the majority of the computational effort.

5.2 Trashcan Shader

We also evaluated our system using a shader from ATI's *Toyshop* demo [Advanced Micro Devices 2006], which combines a simple base geometry with a high-resolution normal map and environment map to reproduce the appearance of a shiny trashcan. It reconstructs the surface color from 25 samples of an environment map combined using a Gaussian kernel. These 25 samples are evaluated along a 5×5 grid of normal directions computed from the normal map. The result is gamma corrected and finally displayed.

Our analysis is presented in Figure 8, again showing the same error/performance graph and the decisions made by our algorithm at four different error thresholds. For the lower error thresholds ε_1 and ε_2, our algorithm chooses to cache those samples that contribute the least amount of energy to the final color (i.e., those with lower weights in the Gaussian reconstruction kernel). For ε_1, only

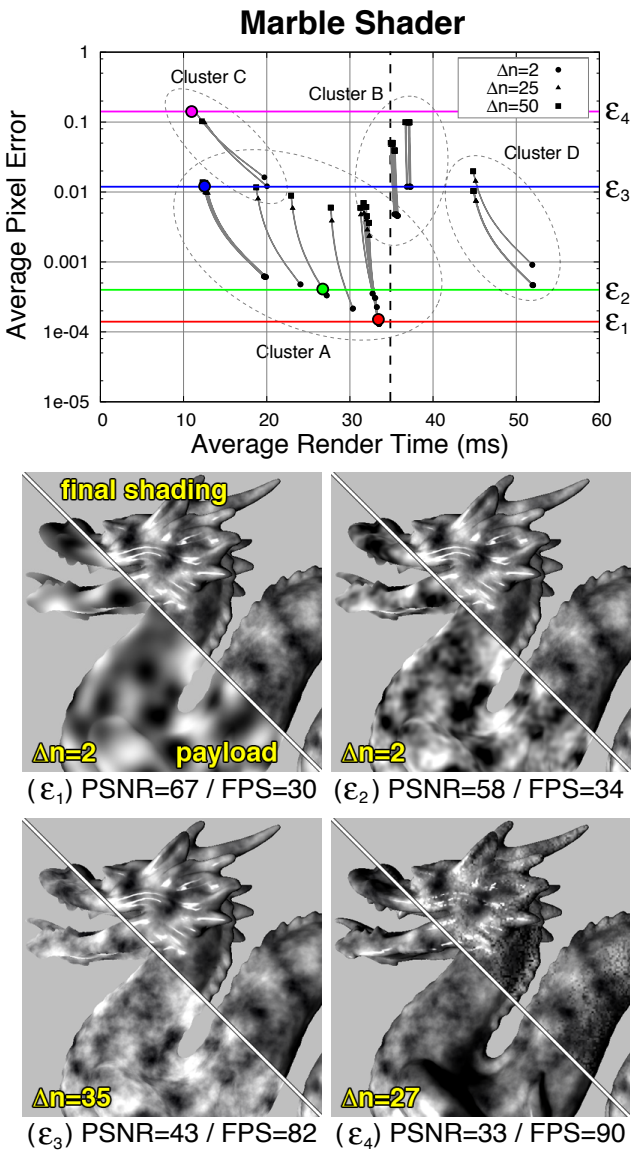

Figure 7: Error/performance paths and four selections made automatically by our system for the Marble shader. The graph at top plots the predicted average pixel error vs. render time over a range of refresh periods for each cacheable node in the AST. The images below visualize reprojection policies at four different error thresholds indicated in the graph. Each image displays the cache payload and chosen refresh period, along with the final shading and measured frame rate. The original shader runs at 29 FPS as indicated by the dashed line.

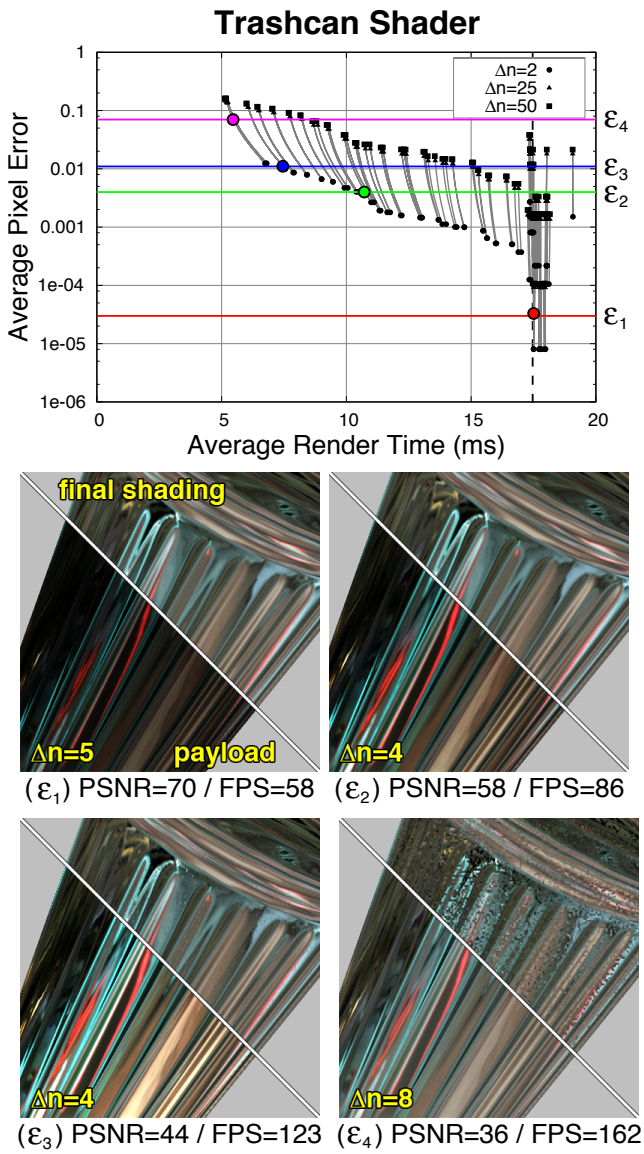

Figure 8: Error/performance paths and four selections made by our system for the Trashcan shader. The original shader runs at 59 FPS.

a single sample is cached; at ε_2, sixteen samples are cached. As before, these examples do not provide huge performance gains, but they do help illustrate the behavior of our system. For the error threshold ε_3, our algorithm selects the sum of 24 samples of the possible 25. In other words, this modified shader evaluates 24 samples every fourth frame (on average) and evaluates the single sample with the greatest reconstruction weight at every frame. This provides a 2.1x performance improvement at an acceptable level of error. These parameters were used in Figure 1. As is always the case, for the largest threshold we consider (ε_4), our algorithm selects the final pixel color with a refresh period of 8. This gives further speed improvements, but injects noticeable artifacts into the shading. As indicated by the absence of paths toward the lower left region of the graph in Figure 8, this shader is not particularly well suited for temporal reprojection. This is because all of the calculations depend strongly on the camera position and cached values quickly become stale. Nevertheless, our system is able to achieve reasonable performance gains even for this challenging case.

5.3 Ambient Occlusion Shader

This shader estimates the ambient occlusion at each pixel using the technique proposed by Bunnell [2005] and later extended by Hoberock and Jia [2007]. The basic idea is to approximate the scene geometry as a collection of discs. These are organized into a hierarchical data structure and stored in a texture map. As each pixel is shaded, this data structure is traversed in order to compute the percentage of the hemisphere that is occluded. This calculation is combined with a diffuse texture and a Blinn-Phong specular calculation to produce the final surface color. We observed render times of around 2 FPS on a Dell XPS with an ATI Radeon HD2900 graphics card at a screen resolution of 640×480. Because most of this

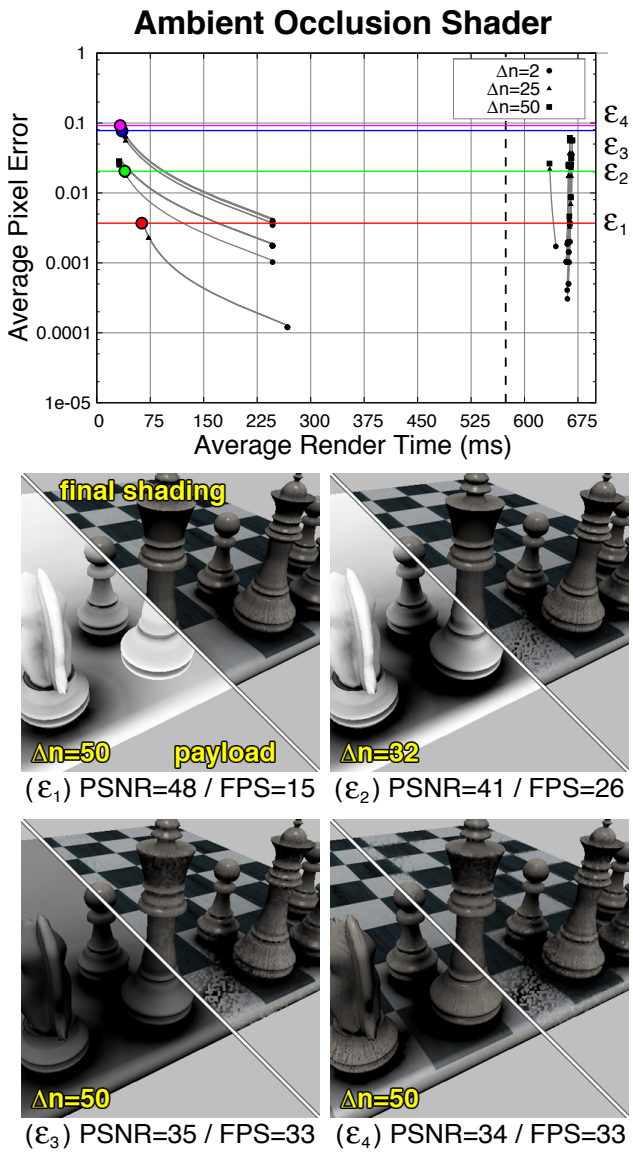

Figure 9: Error/performance paths and four selections made by our system for the Ambient Occlusion shader. The original runs at 1.8 FPS.

shader's effort is spent traversing a hierarchical data structure with a series of nested loops, we found a disproportionately small number of cacheable nodes in the AST (recall that we do not consider nodes which appear inside dynamic blocks of code). Nevertheless, our system successfully identified several calculations that are suitable for temporal reuse.

The ambient occlusion calculation is carried out by summing the contribution of the king chess piece separately from the other pieces. The sample render session shows only the king moving, in addition to the camera and light-source position, while the positions of the other chess pieces and board remain fixed. Therefore, the contribution of these fixed pieces does not change during the animation, while the contribution from the king changes with its position. The analysis for this shader is presented in Figure 9. The presence of nodes in the lower-left region of the error/performance graph indicate a clear opportunity for reprojection. Indeed, for an error threshold of ε_1, our algorithm selects to reuse the portion of the ambient occlusion calculation that accounts for only the static pieces, thus computing the contribution of the moving king and the remaining shading at every frame. This provides a 8x speed-up for a marginal level of error and was used in Figure 1. Note that this image is one frame from a sequence in which both the camera position and king are moving. For the larger error threshold ε_2, our system chooses to cache the entire ambient occlusion calculation. Although this gives a 15x performance gain, it introduces visible artifacts into the shading (easily seen around the base of the king) since cached data become stale at a quicker rate. For a threshold of ε_3, our system chooses the product of the ambient occlusion term and the cosine-falloff for the light-source. This gives only a slight performance improvement over ε_2, even for a larger refresh period, since the added term is relatively inexpensive to compute. Finally, for the largest threshold ε_4, our system selects to reuse the final surface color at the maximum refresh period of 50 frames. This also brings only minor performance improvements over lower thresholds since the majority of the computation is devoted to the ambient occlusion calculation. Furthermore, reusing the final color causes significant artifacts.

5.4 Comparison to Prior Work

We also compare our work to two existing techniques for reducing pixel load. *Dynamic video resizing* [Montrym et al. 1997] decreases rendering latency by evaluating the shading in an off-screen buffer at a reduced resolution and, in a second pass, resampling this buffer at the target resolution. An improved resizing technique was introduced by Yang et al. [2008]. We also implemented the simplification method proposed by [Pellacini 2005]. Figure 10 shows *equal time* comparisons of these three techniques for the *Marble* shader. We selected a buffer size and level of simplification that give improvements comparable to the result obtained with our approach using the threshold ε_3 in Figure 7. The insets show close-up views of the final shading and a visualization of the error. We used the same sequence as in Figure 1 and 7.

As can be seen in the inset, resizing the framebuffer clearly degrades high-frequency features of the shading. This is most noticeable around specular highlights and depth discontinuities. Although automatic code simplification retains these features, important detail in the marble texture is lost as a result of replacing several high-frequency noise calculations with constant expressions. For this example, reprojection provides a significantly better compromise between speed and accuracy due to the wider range of optimization choices available and its ability to better adapt to changes in the scene. While the margin of difference between these methods would not be as wide in every case, reprojection in general has greater flexibility in preserving important spatial and angular details at the expense of temporal reprojection artifacts, but with the overhead of two additional rendering passes.

6 Discussion

Although it is possible to modify each of the shaders in Section 5 by hand to achieve results similar to those we report, it is exactly this type of manual effort that our work aims to alleviate. For example, the static portion of the ambient occlusion in Figure 9 could be precomputed and stored in a static texture map using, for instance, the approach outlined by Jones et al. [2000]. In practice, however, shaders are re-purposed many times and these types of optimizations could be missed by a designer less familiar with the source code. More importantly, it's not uncommon for a shader to be used under many different rendering conditions within a single application. For example, a complete chess game may involve separate animation sequences for the motion of each piece. Our system provides a way to optimize this entire set of shaders automatically, by simply applying different sample sessions. Additionally, the reprojection cache we use obviates the need for an explicit surface parameterization of scene elements, which is a common requirement

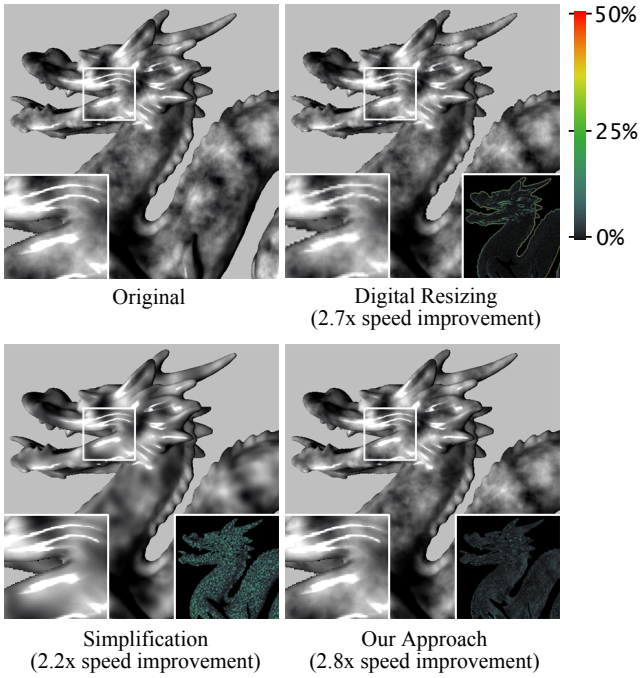

Figure 10: Error comparisons at equal acceleration rates for three methods: dynamic resizing, code simplification, and our approach.

of manual precomputation techniques, since it only reuses data temporarily stored at visible surface locations. Another benefit of our approach is its ability to identify good candidates for reprojection that have little physical meaning and could be easily overlooked by a human. For example, the nodes selected for ε_1 and ε_2 in Figure 7 are buried inside a noise calculation and might not be caught during a manual search. Finally, our system could be used to generate a level-of-detail (LOD) approximation of a pixel shader. Although this paper has focused on exposing and analyzing the underlying error/performance trade-offs involved with data reprojection, these decisions could be made at runtime based on a variety of factors: viewing direction and position, camera motion, etc. These types of optimizations would be tedious to perform by hand.

7 Limitations and Future Work

Although this paper marks an important step in demonstrating the role of data reprojection as a general tool for optimizing procedural shaders, there are many areas that warrant further study. Our exhaustive analysis could be improved with a search over the code's dependency graph and the use of pruning algorithms. Future work should also consider the general problem of caching multiple nodes and investigate compilation methods, including loop unrolling and code reordering, that can improve a shader's suitability for reprojection. Additionally, it should be possible to make cache decisions at runtime based on online estimates of the scene's volatility. These techniques would require less precomputation, and would adapt better to changing inputs. We believe heterogeneous CPU+GPU architectures that have recently emerged [NVIDIA Corporation 2007] provide an excellent platform for these types of algorithms which would display a combination of instruction and data parallelism. It is also important to better understand the trade-offs between static code simplification and reprojection, with the goal of developing unified algorithms that combine the best elements of these complimentary approaches. Additionally, future work should explore alternative cache parameterizations, similar to the work of Miller et al. [1998]. Although our choice to associate cache entries with visible surface locations is justified in many cases, we have found examples such as the *Trashcan* shader in which defining cache coordinates at reflected directions would result in greater temporal uniformity. Finally, novel hardware design could improve the efficiency of data reprojection. In particular, a tighter integration between refresh patterns and the execution granularity imposed by the underlying hardware could significantly reduce overhead.

8 Conclusion

This paper has introduced a set of techniques for automating the use of data reprojection as a general tool for optimizing pixel shaders. Although the general strategy of caching and reusing expensive shading calculations has previously been explored, the critical decision of what partial shading information to cache has remained a fully manual process. Our automated approach relies on parametric models of the performance and fidelity of each of a set of shaders modified to cache subexpressions in the input's abstract syntax tree. These models drive an interactive profiler in which a user may explore the accuracy/speed trade-offs associated with reprojection. We evaluated our models and final selection algorithm using a prototype system to optimize a range of complex pixel shaders and directly compared our approach to current alternatives.

Acknowledgements

The authors thank Natalya Tatarchuk and AMD Corporation for providing the Trashcan shader along with David Luebke and Eugene d'Eon for their help with early experiments. Jason Lawrence acknowledges an NSF CAREER award CCF-0747220 and an NVIDIA Professor Partnership award. Pedro Sander and Lei Yang were partially funded by RGC CERG grant #619207.

References

S. J. Adelson and L. F. Hodges. 1995. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications 15*, 3, 43–52.

Advanced Micro Devices, 2006. ATI toyshop demo.

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*.

D. Aliaga and A. Lastra. 1998. Smooth transitions in texture-based simplification. In *Computers & Graphics*, 22, 71–81.

S. Badt. 1988. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer 4*, 3, 123–132.

K. Bala, J. Dorsey, and S. Teller. 1999. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics 18*, 3, 213–256.

G. Bishop, H. Fuchs, L. McMillan, and E. J. S. Zagier. 1994. Frameless rendering: double buffering considered harmful. In *Proceedings of ACM SIGGRAPH' 94*, ACM, New York, NY, USA, 175–176.

J. F. Blinn. 1977. Models of light reflection for computer synthesized pictures. *Computer Graphics (Proceedings of ACM SIGGRAPH 77) 11*, 2, 192–198.

M. Bunnell. Dynamic ambient occlusion and indirect lighting. In M. Pharr, Ed., *GPU Gems 2*, 223–233. Addison-Wesley.

S. E. Chen and L. Williams. 1993. View interpolation for image synthesis. *Computer Graphics (Proceedings of ACM SIGGRAPH 93)*, 279–288.

R. L. Cook. 1984. Shade trees. *Computer Graphics (Proceedings of ACM SIGGRAPH 84) 18*, 3, 223–231.

R. L. Cook, L. Carpenter, and E. Catmull. 1987. The REYES image rendering architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 87) 21*, 4, 95–102.

A. Dayal, C. Woolley, B. Watson, and D. Luebke. 2005. Adaptive frameless rendering. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*, 265–275.

X. DÉCORET, F. DURAND, F. SILLION, AND J. DORSEY. 2003. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics (Proc. SIGGRAPH) 22*, 3, 689–696.

P. GAUTRON, K. BOUATOUCH, AND S. PATTANAIK. 2007. Temporal radiance caching. *IEEE Transactions on Visualization and Computer Graphics 13*, 5, 891–901.

P. GAUTRON, J. KŘIVÁNEK, K. BOUATOUCH, AND S. PATTANAIK. 2005. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*, 55–64.

S. J. GORTLER, R. GRZESZCZUK, R. SZELISKI, AND M. F. COHEN. 1996. The lumigraph. In *Proceedings of ACM SIGGRAPH 96*, 43–54.

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

J. HASSELGREN AND T. AKENINE-MOLLER. 2006. An efficient multi-view rasterization architecture. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*, 61–72.

V. HAVRAN, C. DAMEZ, K. MYSZKOWSKI, AND H.-P. SEIDEL. 2003. An efficient spatio-temporal architecture for animation rendering. In *Rendering Techniques*, 106–117.

J. HOBEROCK AND Y. JIA. High-quality ambient occlusion. In H. NGUYEN, Ed., *GPU Gems 3*, 257–274. Addison-Wesley.

T. R. JONES, R. N. PERRY, AND M. CALLAHAN. 2000. Shadermaps: A method for accelerating procedural shading. Technical report, Mitsubishi Electric Research Laboratories.

T. B. KNOBLOCK AND E. RUF. 1996. Data specialization. In *Proceedings of SIGPLAN*, ACM, New York, NY, USA, 31, 215–225.

G. W. LARSON AND M. SIMMONS. 1999. The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics 18*, 4, 361–368.

M. LEVOY AND P. HANRAHAN. 1996. Light field rendering. In *Proceedings of ACM SIGGRAPH 96*, 31–42.

P. MACIEL AND P. SHIRLEY. 1995. Visual navigation of large environments using textured clusters. In *Proceedings of the Symposium on Interactive 3D Graphics*, 95–ff.

W. R. MARK, L. MCMILLAN, AND G. BISHOP. 1997. Post-rendering 3D warping. In *Proceedings of the Symposium on Interactive 3D Graphics*, 7–ff.

G. MILLER, M. HALSTEAD, AND M. CLIFTON. 1998. On-the-fly texture computation for real-time surface shading. *IEEE Computer Graphics and Applications 18*, 2, 44–58.

J. S. MONTRYM, D. R. BAUM, D. L. DIGNAM, AND C. J. MIGDAL. 1997. InfiniteReality: A real-time graphics system. In *Proceedings of ACM SIGGRAPH 97*, ACM, 293–302.

D. NEHAB, P. V. SANDER, J. LAWRENCE, N. TATARCHUK, AND J. R. ISIDORO. 2007. Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware*, 25–35.

J. A. NELDER AND R. MEAD. 1965. A simplex method for function minimization. *Computer Journal 7*, 4, 308–313.

NVIDIA CORPORATION, 2007. NVIDIA CUDA Compute Unified Device Architecture programming guide.

M. OLANO, B. KUEHNE, AND M. SIMMONS. 2003. Automatic shader level of detail. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, 7–14.

F. PELLACINI. 2005. User-configurable automatic shader simplification. *ACM Transacations on Graphics 24*, 3, 445–452.

K. PERLIN. 1985. An image synthesizer. *Computer Graphics (Proceedings of ACM SIGGRAPH 85) 19*, 3, 287–296.

J. RAGAN-KELLEY, C. KILPATRICK, B. W. SMITH, D. EPPS, P. GREEN, C. HERY, AND F. DURAND. 2007. The lightspeed automatic interactive lighting preview system. *ACM Transacations on Graphics 26*, 3, 25–36.

M. REGAN AND R. POSE. 1994. Priority rendering with a virtual reality address recalculation pipeline. In *Computer Graphics (Proceedings of ACM SIGGRAPH 94)*, 155–162.

G. SCHAUFLER AND W. STÜRZLINGER. 1996. A three dimensional image cache for virtual reality. *Computer Graphics Forum (Proc. of EUROGRAPHICS) 15*, 3, 227–236.

D. SCHERZER, S. JESCHKE, AND M. WIMMER. 2007. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)*, 45–50.

J. SHADE, D. LISCHINSKI, D. H. SALESIN, T. DEROSE, AND J. SNYDER. 1996. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of ACM SIGGRAPH*, ACM, New York, NY, USA, 75–82.

M. SIMMONS AND C. H. SÉQUIN. 2000. Tapestry: dynamic mesh-based display representation for interactive rendering. In *Eurographics Workshop on Rendering*, Springer-Verlag, London, UK, 329–340.

P. SITTHI-AMORN, J. LAWRENCE, L. YANG, P. V. SANDER, AND D. NEHAB. 2008. An improved shading cache for modern gpus. In *Graphics Hardware*, 95–101.

M. STAMMINGER, J. HABER, H. SCHIRMACHER, AND H.-P. SEIDEL. 2000. Walkthroughs with corrective texturing. In *Rendering Techniques*, Springer-Verlag, London, UK, 377–388.

T. TAWARA, K. MYSZKOWSKI, AND H.-P. SEIDEL. 2004. Exploiting temporal coherence in final gathering for dynamic scenes. In *Proceedings of the Computer Graphics International (CGI)*, IEEE Computer Society, Washington, DC, USA, 110–119.

P. TOLE, F. PELLACINI, B. WALTER, AND D. P. GREENBERG. 2002. Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics 21*, 3, 537–546.

J. TORBORG AND J. T. KAJIYA. 1996. Talisman: commodity real-time 3d graphics for the PC. In *Proceedings of ACM SIGGRAPH 96*, 353–363.

B. WALTER, G. DRETTAKIS, AND S. PARKER. 1999. Interactive rendering using the render cache. In *Rendering Techniques*, Springer-Verlag/Wien, New York, NY, D. Lischinski and G. Larson, Eds., 10, 235–246.

C. WOOLLEY, D. LUEBKE, B. WATSON, AND A. DAYAL. 2003. Interruptible rendering. In *Proceedings of the Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, 143–151.

L. YANG, P. V. SANDER, AND J. LAWRENCE. 2008. Geometry-aware framebuffer level of detail. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering (EGSR)) 27*, 4, 1183–1188.

T. ZHU, R. WANG, AND D. LUEBKE. October, 2005. A GPU accelerated render cache. In *Pacific Graphics*, (Short Paper Session).