Extending the Graphics Pipeline with Adaptive, Multi-Rate Shading

Yong He Yan Gu Kayvon Fatahalian
Carnegie Mellon University

Abstract

Due to complex shaders and high-resolution displays (particularly on mobile graphics platforms), fragment shading often dominates the cost of rendering in games. To improve the efficiency of shading on GPUs, we extend the graphics pipeline to natively support techniques that adaptively sample components of the shading function more sparsely than per-pixel rates. We perform an extensive study of the challenges of integrating adaptive, multi-rate shading into the graphics pipeline, and evaluate two- and three-rate implementations that we believe are practical evolutions of modern GPU designs. We design new shading language abstractions that simplify development of shaders for this system, and design adaptive techniques that use these mechanisms to reduce the number of instructions performed during shading by more than a factor of three while maintaining high image quality.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors

Keywords: GPU architecture, graphics pipelines, shading

Links: DL PDF

1 Introduction

Per-fragment shading computations dominate the cost of rendering in many modern games. For example, high-end "AAA" titles employ expensive fragment shaders that implement complex material and lighting models needed to represent realistic scenes. Conversely, casual 2D and 3D games employ simple shaders, but these applications feature commensurately low geometric complexity and are typically enjoyed on resource-constrained mobile devices with high-resolution displays. Indeed, today's highest-resolution tablets require mobile GPUs to synthesize sharp, four-megapixel images (more pixels than most modern 27-inch desktop displays) using a power budget of only a few watts. Under these conditions, shading computations constitute nearly 95% of the processing cost of modern OpenGL ES applications [Shebanow 2013].

Traditionally, real-time graphics systems have favored simple, "brute-force" techniques amenable to parallelization or acceleration via specialized hardware. This approach was justified by the benefits of performance predictability and continual performance improvement through additional GPU processing resources. However, energy constraints (now omnipresent for both high-end and mobile devices) make it increasingly difficult to rely on rapid growth in compute capability as a primary mechanism for improving the quality of real-time graphics. Simply put, to scale to more advanced rendering effects and to high-resolution outputs, future GPUs must adopt techniques that perform shading calculations more efficiently than the brute-force approaches used today.

In this paper, we enable high-quality shading at reduced cost on GPUs by extending the graphics pipeline's fragment shading stage to natively support techniques that adaptively sample aspects of the shading function more sparsely than per-pixel rates. Specifically, our extensions allow different components of the pipeline's shading function to be evaluated at different screen-space rates and provide mechanisms for shader programs to dynamically determine (at fine screen granularity) which computations to perform at which rates. We perform an extensive study of the challenges of integrating *adaptive, multi-rate shading* into the graphics pipeline by evaluating two- and three-rate implementations that we believe are plausible evolutions of modern GPU designs. We then present new shading language abstractions that simplify development of shaders for this system. Last, we describe adaptive shading techniques that utilize the proposed mechanisms to reduce the number of instructions performed by the graphics pipeline during shading by more than a factor of three while also maintaining high visual quality.

2 Background

Limiting the cost of shading in real-time rendering applications is a central challenge for both graphics application developers and GPU architects. One way to reduce the cost of shading is to simplify the shading function itself, for example, through manual design of efficient approximations to material and lighting models, by precomputing common sub-computations (often involving prefiltering of values in precomputed structures), sharing sampling results within a quad fragment via differential instructions [Penner 2011], or via automatic compiler optimization [Guenter et al. 1995; Olano et al. 2003; Pellacini 2005; Sitthi-Amorn et al. 2011].

A complementary strategy is to reduce the number of times the fragment shading function is invoked during rendering. For example, all modern GPUs attempt to eliminate shader invocations for occluded surfaces via early-Z cull mechanisms. GPUs also reduce the number of shader invocations by reusing a single shading result for multiple coverage samples in a pixel (multi-sampling anti-aliasing) [Akeley 1993] and by sharing shader intermediate results across neighboring fragments to compute derivatives. Techniques such as quad-fragment merging exploit spatial coherence even more aggressively by sharing shading computations between adjacent mesh triangles [Fatahalian et al. 2010]. In addition, an increasing number of games also exploit temporal coherence of shading by reusing shading results from the previous frame via reprojection [Scherzer et al. 2012; NVI 2013]. Our work also seeks to reduce shading costs by evaluating the shading function less often, but we pursue this goal by architecting first-class pipeline support for adaptively sampling components of the shading function (e.g., terms with low screen-space variation) at lower than per-fragment rates. Thus our contributions are orthogonal and complimentary to the above approaches.

Multi-rate (or "mixed-resolution") rendering techniques are already employed by today's high-end game engines to reduce shading

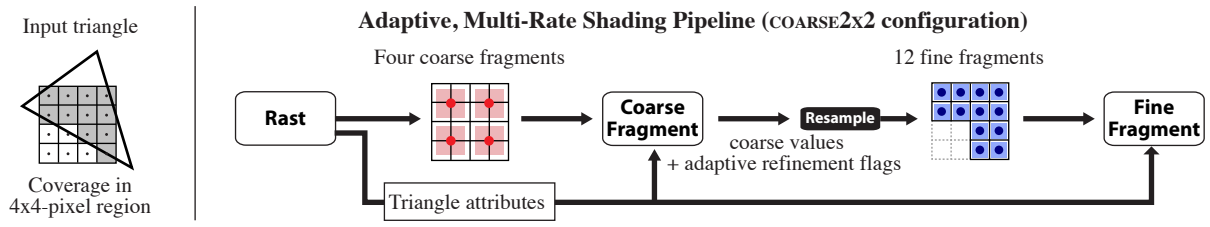


Figure 1: *The multi-rate shading pipeline rasterizes triangles into coarse fragments that correspond to multiple pixels of coverage (here: 2×2 pixels). Coarse fragments are shaded, then partitioned into fine fragments for subsequent per-pixel shading. If the coarse fragment shader determines an effect should not be evaluated at low sampling rates, these computations are performed in the fine shading stage.*

costs on current GPUs, but they are implemented as an application software layer above the graphics pipeline using multi-pass rendering. While many variants exist [Yang et al. 2008], the most common implementations are based on deferred shading [Kircher and Lawrance 2009; Tatarchuk et al. 2013]. The basic process is to use the traditional GPU pipeline to generate a low-resolution G-buffer (e.g., 1/4 screen resolution) as input for a low-resolution deferred shading pass. This pass evaluates low-frequency terms of the shading function for all G-buffer samples, resulting in a sparse screen-space sampling these terms. Many lighting computations are attractive to perform in this low-resolution pass due to their high cost and slow screen-space variation. Then, a second shading pass upsamples the low-resolution shading results and uses them as input (along with a traditional, full-screen resolution G-buffer) for final per-pixel shading computations.

Although implementing multi-pass, mixed resolution rendering on top of the existing GPU pipeline can be an effective way to lower shading costs, the approach suffers from a number of drawbacks: First, to avoid artifacts caused by resampling coarse shading results onto the wrong surface, upsampling must respect geometric discontinuities. Implementations typically employ a cross-bilateral upsampling filter (e.g., using surface ids, depth, normals as inputs) to minimize interpolation across discontinuities [Yang et al. 2008], but artifacts still remain. Errors due to resampling can be particularly troublesome when rendering fine-scale geometry, which may not be sampled at all in the low-resolution buffer. A second drawback is that implementations built upon current GPU capabilities statically partition the shading function into low and high-frequency terms. They do not attempt to adapt shading rate based on need. As a result, implementations either accept loss of quality due to undersampling (e.g., low-resolution lighting calculations fail to capture sharp shadows), or must be conservative in their choice of which shading terms to evaluate sparsely. Building adaptive shading on top of existing pipeline mechanisms [Nichols et al. 2010] introduces significant rendering engine complexity and incurs multi-pass overhead. We do not view it as a practical, efficient solution and to our knowledge it is not commonly implemented in games.

Thus, we believe there are strong reasons to integrate multi-rate shading more tightly into the GPU graphics pipeline. They include:

- Multi-rate shading in a single forward rendering pass. Forward rendering remains an important rendering strategy and would benefit from multi-rate shading optimizations. Moreover, single-pass forward rendering removes the need for heuristic edge-finding during upsampling.
- Adaptivity. Our experiences show that value of multi-rate shading is far greater if shaders can adaptively choose sampling rates at runtime.
- Convenience, portability, and performance. Existing games provide ample evidence that multi-rate shading is an effective way to reduce shading cost with minimal impact on image quality. First-class pipeline and shading language support systematizes this technique, simplifying its use in both forward and deferred rendering contexts and provides opportunities for underlying GPU optimization.

3 Multi-Rate Pipeline Architecture

In this section, we introduce extensions to the graphics pipeline that provide support for performing adaptive, multi-rate shading in a single draw operation. We first describe the basic system architecture as well as key details of several prototype implementations. We provide examples of the system's use in Section 4, and defer description of how to target the features of this architecture in a high-level shading language to Section 5.

Our proposed multi-rate shading pipeline, shown in Figure 1, partitions execution of post-rasterization (and optionally post-z-cull) screen-space shading operations into two stages: **fine fragment** operations, like traditional fragment processing, are executed once per covered pixel; and **coarse fragment** operations, which occur prior to fine fragment shading and are executed at a lower rate. To simplify explanation, Figure 1 illustrates a specific pipeline configuration where the coarse fragment shading stage samples shading once per 2×2-pixel region. We will describe more complex configurations of coarse fragment processing in Section 3.1.

As shown in the figure, the rasterizer first computes triangle-screen coverage then generates coarse fragments for shading (coverage for a 4×4-pixel region is shown in gray). To support derivative computations during coarse shading computations, coarse fragments are always shaded in blocks of four. As a result, if any visibility sample in a 4×4-pixel region is covered by the triangle, a block of four coarse fragments will be emitted for shading (shown in red). Following coarse fragment processing, coarse fragments are partitioned into fine fragments. Blocks of fine fragments are only generated if coverage exists for the corresponding 2×2-pixel region. Thus, the multi-rate pipeline never shades more fine fragments than a traditional pipeline. In the figure, the block of four coarse fragments is broken into three blocks of coarse fragments (twelve fine fragments in total—shown in blue) because the triangle does not cover the bottom-left 2×2-pixel region.

Figure 2 provides HLSL-like pseudocode for a coarse fragment shader (`coarse_shader`) that computes diffuse and specular lighting and a fine fragment shader (`fine_shader`) that modulates the coarsely computed lighting results with the results of per-pixel samples from an albedo texture. Notice that in the multi-rate shading pipeline, there are two sources of varying inputs to the fine fragment shader program: values computed by coarse fragment shading and then resampled to fine fragment shading sample positions (`coarse`), and input triangle attributes (`attribs`) that are evaluated using triangle attribute equations computed during rasterization. This design is similar to how the pipeline's domain shader stage receives surface attributes from the hull shader

```
struct Coarse_in {
  float3 norm, vert;
};

struct Coarse_out {
  float diff, spec;
  bool  spec_refine; : SV_REFINE_FLAG_0;
};

struct Fine_in {
  float3 norm, vert;
  float2 uv;
};

uniform float3    lightPos;
uniform sampler2D albedo;

Course_out coarse_shader(Coarse_in in)
{
  Course_out result;
  result.spec_refine = false;
  result.diff = diff_lighting(in.norm, in.vert, lightPos);
  result.spec = spec_lighting(in.norm, in.vert, lightPos);

  // determine if refinement is necessary
  if (fwidth(result.spec) > THRESHOLD)
    result.spec_refine = true;
  return result;
}

vec4 fine_shader(Fine_in attribs, Coarse_out coarse)
{
  // use coarsely sampled specular lighting or
  // recompute once per pixel
  float specular = coarse.spec_refine ?
      spec_lighting(attribs.norm, attribs.vert, lightPos) :
      coarse.spec;
  float lighting = coarse.diff + specular;
  return texture(albedo, attribs.uv) * lighting;
}
```

Figure 2: *HLSL-like pseudocode for coarse and fine fragment shaders that cooperate to evaluate diffuse lighting coarsely, surface albedo once per pixel, and specular lighting adaptively based on need. (Subroutines* `diff_lighting` *and* `spec_lighting` *evaluate diffuse and specular lighting terms accordingly.)*

and vertex parametric locations from the tessellator. It avoids the need for applications to "plumb" fine attributes through the coarse shading stage. More importantly, the design ensures all triangle attributes are available at full precision to the fine fragment processing stage regardless of coarse stage operation. (They do not incur coarse-to-fine resampling error.)

In the pseudocode, `coarse_shader` and `fine_shader` cooperate to dynamically adjust the rate at which specular lighting is evaluated based on local properties of the surface. In this example, the coarse shader evaluates whether specular lighting variation is sufficiently high within the current coarse shading block. If so, it sets a *refinement flag* (`spec_refine`) to signal the fine shader that specular lighting should be performed at a per-pixel rate (i.e., resampled coarse lighting results may yield artifacts and should be ignored). Adaptive refinement flags are exposed to the pipeline as system-interpreted values to allow for optimization, which is described in Section 3.1 and Section 3.3.

In the following three sections we provide details of how coarse stage shading sample locations are determined and how our prototype multi-stage shading pipeline is scheduled for efficiency under wide SIMD execution.

3.1 Coarse Fragment Stage Configurations

Our prototype pipeline supports three configurations of the coarse fragment processing stage. The configurations result in different sampling densities for coarse shading.

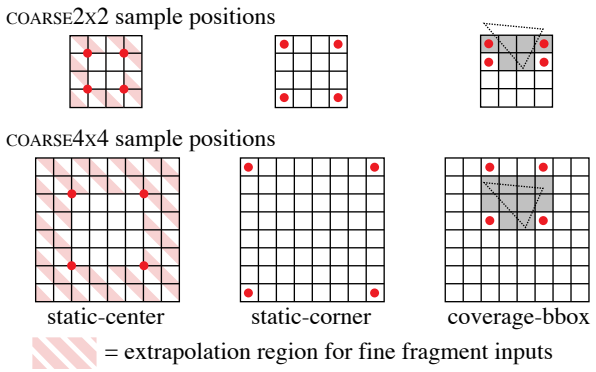


Figure 3: *Three coarse sampling modes implemented in our multi-rate shading pipeline. Sampling positions impact the quality of resampling, the likelihood of sampling shader input attributes outside triangle boundaries, and the robustness of refinement decisions.*

COARSE2X2. Coarse fragments correspond to 2×2-pixel regions of triangle-screen coverage. The minimum granularity of coarse fragment shading in the pipeline is an 4×4-pixel block. This is the configuration illustrated in Figure 1.

COARSE4X4. The same as above, but coarse fragments correspond to 4×4-pixel regions of coverage, yielding a sparser coarse stage sampling. The minimum granularity of coarse fragment shading is an 8×8-pixel block.

DYNAMIC4X4. The pipeline dynamically chooses between COARSE2X2 and COARSE4X4 configuration coarse shading for each covered 8×8-pixel block. The pipeline first generates coarse fragments for the COARSE4X4 configuration. If coarse processing sets block refinement flags, the coarse shading outputs are ignored and each COARSE4X4 coarse fragment is partitioned into COARSE2X2 coarse fragments. Then the coarse shader is re-executed on these fragments. The DYNAMIC4X4 configuration makes the most aggressive attempts to employ coarse shading, but incurs the overhead of repeating evaluation of the coarse stage shader in situations where per-pixel shading is necessary.

3.2 Coarse Fragment Stage Sampling

A straightforward implementation of coarse fragment shading evaluates input attributes at the center of each coarse fragment, regardless of triangle coverage (Figure 3-left). Coarse shading results evaluated at these locations can then be smoothly resampled to fine fragment stage sampling positions using bilinear filtering.

While this approach is simple, it presents two possible image-quality issues for a multi-rate system. Resampling coarse outputs to pixels outside the hull of the sample points requires extrapolation of the block's four coarse shading results, risking discontinuities at block boundaries. (The pipeline shades coarse fragment blocks independently so coarse shading results from adjacent blocks are not available as inputs during resampling.) Second, the static uniform sample pattern may sample coarse shader inputs outside triangle boundaries in cases where the triangle does not fully cover a coarse shading block. This problem exists with traditional fragment shading, but is exacerbated by the larger extent of coarse fragments.

To understand these concerns, we implemented and evaluated two additional approaches to selecting shading sample locations for coarse fragments. The first places the four coarse sample locations at the center of the corner pixels of the coarse block (static-corner: Figure 3-center). It eliminates the need for extrapolation during

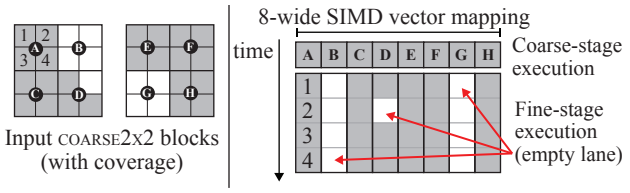


Figure 4: *Scheduling all shading work (coarse and fine) for a coarse block of pixels to a single logical-thread of GPU control can result in inefficient SIMD-vector execution when coarse fragments are not completely covered by a triangle (left, coverage shown in gray). The resulting SIMD lane utilization is shown at right.*

resampling at the cost of non-uniform sampling of coarse shading values (decreasing the effective sampling rate of coarse shading).

The final scheme, sample locations are placed in the corners of a bounding box of the triangle (coverage-bbox: Figure 3-right). This coverage-dependent pattern is an efficient approximation to centroid sampling patterns already present in GPUs. It reduces input attribute extrapolation error (unlike centroid sampling, it does not completely eliminate it), but facilitates simple resampling via bilinear interpolation from the axis-aligned sample positions.

Small-triangle optimization. In all three schemes, if triangle coverage is contained within a 2×2-pixel region of a coarse block, the pipeline evaluates coarse shading at pixel centers of the covered 2×2 block, just as in traditional quad-fragment shading. Coarse shaders should not set refinement flags in this case, as coarse shading results are the same as those of traditional fragment shading.

3.3 Pipeline Scheduling

An evolutionary approach to scheduling multi-rate shading computations is to implement all multi-rate shading logic (both coarse and fine) in a single logical GPU thread (vector lane) of control. In such a design, a GPU would execute logic for a coarse fragment then immediately compute fine stage shading for all corresponding fine fragments in sequence. (This implementation is similar to efforts to implement multi-sample anti-aliasing in deferred renderers [Lauritzen 2010].) While conceptually simple, this approach can suffer from inefficient execution on a wide SIMD processing architecture for two reasons. First, as illustrated in Figure 4, executing a SIMD-vector sized group of coarse fragments together will suffer execution divergence in situations of partial coverage, since coarse fragments will partition into different numbers of fine fragments. (The effective granularity of fine shading is now the coarse block, not 2×2-block of pixels.) Second, since more fine shader instances (up to 16 times more in COARSE4X4 mode) now map to each SIMD group, the likelihood of execution divergence (e.g., due to differences in refinement decisions in this group) is greater.

We seek hardware-optimized implementations of adaptive, multi-rate shading, so our prototype instead separates scheduling of coarse and fine shading tasks, and uses inter-stage buffering to coalesce work in SIMD-vector sized groups for efficient execution. Specifically, for a SIMD width of size N, our implementation schedules a batch of $N/4$ coarse shading blocks in a SIMD group. When execution is complete, the coarse blocks are partitioned into fine fragment blocks and buffered for fine fragment stage processing. The system then dispatches fine shading work in groups of $N/4$ blocks from this buffer. When fine fragments in a SIMD group receive different refinement flags, divergent execution will occur since different execution sequences are necessary in the fine fragment shader. To reduce divergence due to adaptive shading, prior to dispatch, an optional optimization is to inspect the refinement flags of each coarse block and sort buffered fine fragment blocks according to these flags. (We note that an integer sort of elements stored in fixed-sized, on-chip buffers is also proposed by Clarberg et al. [2013] to improve pipeline efficiency.)

Scheduling DYNAMIC4X4 mode execution is more complex. The system maintains buffers for both fine fragments (as done above) and COARSE2X2 mode coarse fragments (coarse stage work to be re-executed). After initial COARSE4X4-mode shading is complete, coarse fragment blocks with no refinement flags set are partitioned into fine fragment blocks and enqueued as described above. Blocks with refinement flags set are broken into COARSE2X2 fragment blocks and buffered for COARSE2X2 mode shading.

The ability to efficiently execute multi-rate shaders on a highly parallel architecture is primarily limited by the storage costs of buffering. The system must maintain visibility coverage information for entire coarse blocks (64 pixels in COARSE4X4 operation) and must allocate on-chip storage to buffer coarse-to-fine stage intermediate values. However, in our experiments, inter-stage intermediates are often compact (three floats for the shaders evaluated in Section 6). Second, rather than directly store resampled intermediates at fine fragment granularity, optimized implementations (particularly COARSE4X4 configurations) should instead buffer the coarse fragment outputs and perform resampling to fine sample locations on demand in the fine shader (similar to how modern GPUs interpolate triangle attributes on-demand from per-triangle data).

4 Adaptive Shader Examples

To enable use of the multi-rate pipeline mechanisms discussed in Section 3, we now describe techniques for robustly and inexpensively determining when shading effects can be accurately evaluated in the coarse fragment stage.

4.1 Diffuse and Specular Lighting

Diffuse Lighting. Diffuse lighting can sometimes be evaluated with sufficient accuracy in the coarse shader without the need for adaptive shading (an approximation made by current games). However, near the silhouettes of curved objects there can be wide variation in interpolated normals across a triangle (particularly when coarse shading sample positions lie outside projected triangle boundaries), leading to large variation in diffuse lighting. Large screen-space derivative of $\mathbf{N} \cdot \mathbf{L}$ is a sufficient signal to detect this case, and we use it to make a refinement decision. Since $\mathbf{N} \cdot \mathbf{L}$ must be computed as part of lighting evaluation, the overhead of this adaptivity check is minimal.

Specular Lighting. Reflectance from specular surfaces may change rapidly with small variations in normal, so a simple way to implement an adaptivity decision is to use a surface flatness test (normal difference) with a low refinement threshold. The problem with this approach is that the flatness threshold must be conservatively set avoid undersampling near highlight regions. This results in a decision to evaluate specular lighting in the fine shader for many regions of the surface, even if these regions are located far from highlights and feature low screen-space variation in specular reflectance. For example, specular lighting is unnecessarily evaluated at per-pixel rates on the chin region of the Ogre character in Figure 5 when a flatness heuristic is used (see heat map, top-center). To reduce unnecessary refinement due to specular lighting, the adaptive shader must determine if a surface region also lies near a specular highlight. We assume a specular BRDF whose magnitude varies slowly when the lighting vector is oriented at a large angle to the view reflection direction (a typical condition for specular BRDFs), making coarse specular lighting evaluation sufficient in these situations. We detect these cases by estimating a bound

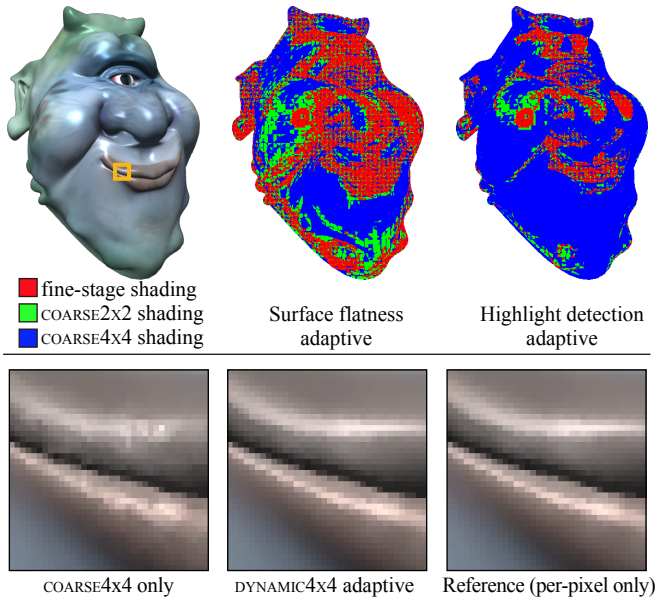


Figure 5: *Bottom: Near highlight regions, it is insufficient to evaluate specular lighting at coarse rates (left). Adaptive techniques (center) are necessary to closely approximate the reference shading result (right). Top: Accounting for the specular highlight position in refinement decisions (right) can avoid avoid unnecessary refinements triggered by a basic flatness heuristic (center). Heat maps visualize specular refinements during* DYNAMIC4X4 *mode shading.*

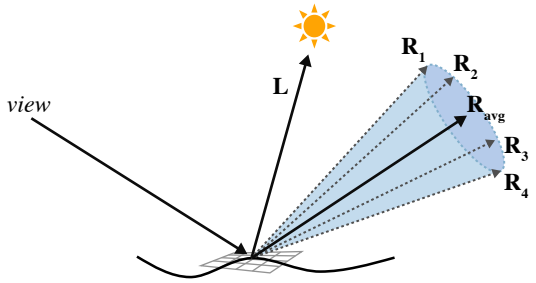


Figure 6: *Specular reflectance may vary significantly in a coarse block if the orientation of the lighting vector* $\mathbf{L}$ *falls in or near the bounding cone of directions formed by a coarse block's reflection vectors* $\mathbf{R}_i$*.*

for the reflection vectors for a coarse block, and then compute the minimum angle of the lighting vector $\mathbf{L}$ with this bound.

As shown in Figure 6 we assume that the light source is sufficiently far from the surface that the lighting vector $\mathbf{L}$ can be treated as a constant for the block, reducing the problem to finding the distance between $\mathbf{L}$ and the bounding cone of the coarse block's four reflection vectors $\mathbf{R}_i$. If the bounding cone contains $\mathbf{L}$, the coarse shading block samples near the peak of specular highlight. We approximate this condition by testing if, for any coarse sample:

$$\mathbf{L} \cdot \mathbf{R}_{avg} \leq \lambda \, (\mathbf{R}_i \cdot \mathbf{R}_{avg})$$

$\mathbf{R}_{avg}$ is the axis of the bounding cone computed as the average of the four $\mathbf{R}_i$. The parameter λ is chosen according to the known spread of the BRDF, effectively enlarging the cone so that refinement will occur when the shading the coarse block samples the BRDF in regions of high variation. The cost of this predicate is computing the average reflection direction $\mathbf{R}_{avg}$ for a block and

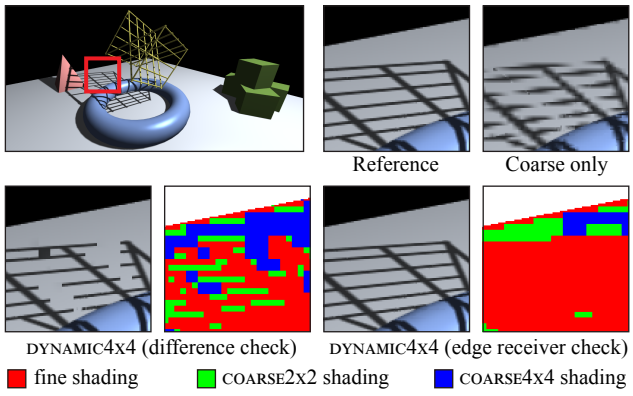
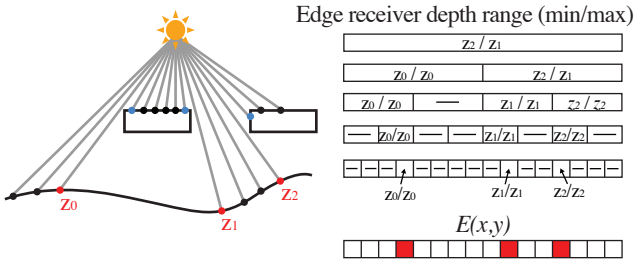


Figure 7: *By precomputing the location of shadow edge receivers in the scene, we are able to robustly determine when a shadow edge may fall between coarse shading samples, avoiding artifacts.*



Pseudocode: edge-receiver aware adaptive shadow shader

```
// compute light-space depth range of block
block_range = [min(lightspace_frag_z), max(lightspace_frag_z)];
refinement_range = texture(edge_receiver_map, shadow_coord);
if (block_range overlaps refinement_range)
    refine;

// evaluate shadow effect
shadow_factor = compute_shadow(shadow_map, shadow_coord);
// final refinement check
if (fwidth(shadow_factor) > THRESHOLD)
   refine;
```

Figure 8: *Top: red dots indicate surface points upon which a shadow edge may fall. The resulting receiver min-max hierarchy represents these regions for rapid lookup in shaders. Bottom: pseudocode for the complete adaptive shadow shader. The subroutine* `compute_shadow` *performs the actual occlusion computation.*

two dot products. The reflection directions $\mathbf{R}_i$ are already required to evaluate lighting.

4.2 Shadow Mapping

Shadows are a compelling candidate for adaptive rate shading because high frequencies are only present near shadow boundaries. Also, popular shadowing techniques (e.g., percentage closer filtering [Reeves et al. 1987]) often require multiple samples from a shadow map and have high compute and bandwidth cost. Thus, the main challenge of efficiently evaluating shadows in an adaptive pipeline is determining whether the surface region spanned by a coarse block receives a shadow edge. Sampling light occlusion at each coarse sample location, then checking for large differences in occlusion results is insufficient since coarse sampling may miss shadow detail falling in between coarse samples (Figure 7, bottom-left). This problem is particularly severe when coarse shading is evaluated once per 4×4-pixel block.

To robustly detect cases where shadow detail lies between coarse samples we analyze the shadow map to find scene regions that can *potentially receive shadow edges*. Figure 8 illustrates a simple geometry setup yielding a shadow map with depth samples indicated by dots. We detect edge pixels in the shadow map by computing the second derivative of depth at each pixel, and then identify pixels with large *negative* secondary depth derivative values. These shadow map pixels, highlighted in red in the figure, correspond to scene surfaces that are shadow edge receivers. Pixels with large *positive* second derivative (blue dots) lie on the other side of the shadow map edge and correspond to surfaces that will not receive shadows. Specifically, we encode edge receiving regions in a binary "edge receiver map" defined as:

$$E(x,y) = \max\left(-\frac{\partial^2}{\partial x^2}Z(x,y), -\frac{\partial^2}{\partial y^2}Z(x,y)\right) \geq \delta$$

Where $Z(x,y)$ gives the depth value at shadow map pixel (x,y). Note that the edge receiver map is not a representation of shadow map edges, it is a representation of scene regions that these edges will cast shadows on. Using the edge receiver map we build a min-max hierarchy that maintains, for each shadow map ray, a conservative depth range of scene regions that may receive a shadow edge (Figure 8 top-right). To avoid artifacts due to undersampling shadow edges, we will not perform shadow calculations at a coarse rate for any surface lying within these regions. We initialize the base level (Z^0_{min} and Z^0_{max}) of the receiver min-max hierarchy from the shadow map and edge receiver map as given below, then generate the remaining levels of the min-max hierarchy.

$$Z^0_{min}(x,y) = \begin{cases} Z(x,y) & : E(x,y) = 1 \\ +\infty & : E(x,y) = 0 \end{cases}$$

$$Z^0_{max}(x,y) = \begin{cases} Z(x,y) & : E(x,y) = 1 \\ -\infty & : E(x,y) = 0 \end{cases}$$

As given by the pseudocode in the bottom half of Figure 8, the coarse shader accesses the receiver min-max map to determine if the light-space depth bounds of the current coarse block overlap with the range of potential shadow edge receivers. If overlap exists, a shadow edge may fall on the surface within this block and so shadow computations should be performed in the fine fragment shader. If the block does not fall in an edge-receiving region, the coarse shader proceeds to compute light occlusion using the shadow map. Since this predicate is based on the receiver min-max hierarchy, it is applicable to many forms of shadow mapping techniques, such as variance shadow maps [Donnelly and Lauritzen 2006] and exponential shadow maps [Annen et al. 2008]. Since the shadowing method itself may introduce (smooth) variation across the block, we complete the adaptive coarse shadow effect with a difference check of the coarse block's four occlusion values.

The overhead of constructing the edge receiver hierarchy is small since a shadow map prefiltering pass is commonly performed by shadowing techniques. Moreover, since Z_{min} and Z_{max} define a conservative range, to reduce the bandwidth overhead of predicate evaluation, these values can be stored at low precision and packed into a 32-bit word. One further optimization (which we omit from the Figure 8 pseudocode for clarity) is to detect when the shadow map texture footprint for an entire coarse block is contained within a 2×2-texel region of the base-level shadow map. In this case, all shadow map lookups during coarse shading access the same shadow texels. Thus, no shadow edges fall between these samples and it is unnecessary to use the receiver min-max map for refinement. This optimization can significantly reduce refinements in situations where shadow maps are undersized for the current view.

4.3 Motion Blur

Recent work has considered adaptively decreasing shading rate under conditions of motion or defocus blur since the blurring effect removes frequency content from the image. While most prior efforts anticipated future pipelines with stochastic rasterization support [Ragan-Kelley et al. 2011; Vaidyanathan et al. 2012; Liktor and Dachsbacher 2012], adaptive multi-rate shading provides a mechanism to perform similar optimizations when blur effects are approximated by popular screen-space post processing techniques (common in current games). For example, it is simple to implement predicates that inspect the motion vectors of a surface and then conditionally execute large parts of the shading function in the coarse shader when blur is estimated to be large. We report on our experiences with adaptive shading under conditions of post-process motion blur in Section 6.4.

5 Adaptive Shading Language

Multi-rate shading increases the complexity of authoring shaders. Defining shading effects for the multi-rate pipeline described in Section 3 using a one-program-per-stage programming model (as in Figure 2) is possible, but it requires programmers to produce both coarse and fine fragment shaders and also marshal intermediate values between these stages. Adaptivity uniquely adds further complexity because a single logical shading effect (e.g., specular lighting) may be executed in either the coarse or fine fragment shading stage. This logic must be duplicated in shader programs bound to both stages and the interface between stages must be modified to hold intermediates and flags. This duplication can make programming tedious and make shaders harder to read and maintain.

While it may seem reasonable to manage complexity by encapsulating an adaptive shading effect's logic in a subroutine called by each stage, this is not a straightforward solution. The code in the top half of Figure 9 shows a more complex adaptive shader that implements the diffuse and specular lighting predicates described in Section 4.1. The code is now dominated by managing adaptivity (e.g., setting/checking flags) and although we have attempted to confine specular lighting logic to the subroutine `spec_lighting` (implementation not shown), the specular effect's refinement flag must still be maintained in `Coarse_Out`. Also, since it is difficult to separate evaluation of the specular adaptive predicate from the computation of specular lighting itself (the predicate depends on intermediate value **R**), the subroutine must evaluate the predicate and return the result to the caller (parameter `refine_flag`). This complicates the subroutine interface in the fine fragment shader when predicate evaluation is not required. Adding additional adaptive effects (e.g., shadows) would yield a shader that is even harder to read and maintain.

5.1 Language Abstractions

To simplify the task of authoring efficient adaptive, multi-rate shaders we believe it is desirable to raise the level of abstraction in shader programming. Specifically, we seek to support encapsulation of entire adaptive effects in simple modules and to simplify management of adaptive behaviors. While adding rate qualifiers (e.g., per-coarse fragment) to existing GLSL/HLSL-style shading languages may seem like a logical way to achieve the above goals, the interaction of rate qualifiers and data-dependent control flow in modern shading languages creates ambiguous semantics that leads to unpredictable compiler behavior (and thus, unpredictable shader performance) [Foley and Hanrahan 2011]. We judged this to be a non-starter for our needs since the sole goal of multi-rate shading is to optimize shader performance. Instead, we pursue a declarative

```
struct Vertex_In {
  float3 vertex, normal;
  float3 view, light_dir;
  float2 uv;
}
struct Coarse_Out {
  float diffuse, specular;
  bool  diffuse_flag  : SV_REFINE_FLAG_0;
  bool  specular_flag : SV_REFINE_FLAG_1;
}
float spec_lighting(Vertex_In vin, out bool refine_flag)
{   ... // may set refine_flag to true ...  }
Coarse_Out coarse_shader(Vertex_In in) {
  Coarse_Out rs;
  rs.diffuse_flag = rs.specular_flag = false;
  float nDotL = dot(in.normal, in.light_dir);
  if (fwidth(nDotL) > DIFFUSE_THRESHOLD) {
    rs.diffuse_flag = true;
  } else {
    rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    rs.specular = spec_lighting(in, rs.specular_flag);
  }
  return rs;
}
float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
  if (cin.diffuse_flag) {
    float nDotL = dot(vin.normal, vin.light_dir);
    cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
  }
  if (cin.specular_flag) {
    bool tmp_flag;  // unused flag, but needed by call below
    cin.specular = spec_lighting(vin, tmp_flag);
  }
  // not shown: modulate lighting by albedo from texture ...
}
```

```
in float3 vertex, normal;
in float3 view, light_dir;
in float2 uv;
uniform sampler2d texAlbedo;
coarse effect float diffuse {
  float nDotL = dot(light_dir, normal);
  if (fwidth(nDotL) > DIFFUSE_THRESHOLD) refine;
  diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
}
coarse effect float specular {
  if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
    refine;
  float3 R = reflect(view, normal);
  float rDotL = dot(R, light_dir);
  specular = pow(rDotL, k);
  float3 Ravg = average(R);
  if (dot(Ravg, light_dir) < lambda * dot(Ravg, R))
    refine;
}
fine effect float4 albedo = texture(texAlbedo, uv);
out float4 color = albedo * (diffuse + specular);
```

Figure 9: *Top: A shader implementing adaptive diffuse and specular lighting effects. Managing refinement flags and maintaining logic across two pipeline stages can make adaptive shader programming laborious and shaders difficult to read. Bottom: The same shader written in our proposed declarative, effect-centric programming language.*

shading language design that draws ideas from RTSL [Proudfoot et al. 2001] and Foley et al.'s Spark language [2011] which was motivated by many of the same cross-pipeline-stage code maintenance issues highlighted above. However, our language describes only programs for the fragment shading stages of our pipeline (rather than the entire pipeline) and it includes first-class primitives for managing adaptivity.

The main idea of our language is to structure complex shaders as DAGs of *effects*. Effects encapsulate all the logic necessary to compute the value of a logical feature of a shader (e.g., specular lighting or a shadow term). The bottom half of Figure 9 shows the previous lighting example in our effect-based language. The shader is composed of four effects: `diffuse` (evaluates diffuse lighting), `specular` (evaluates specular lighting), `albedo` (the surface albedo, obtained from texture), and `color` (the output color of the shader). `color` depends on the other three effects.

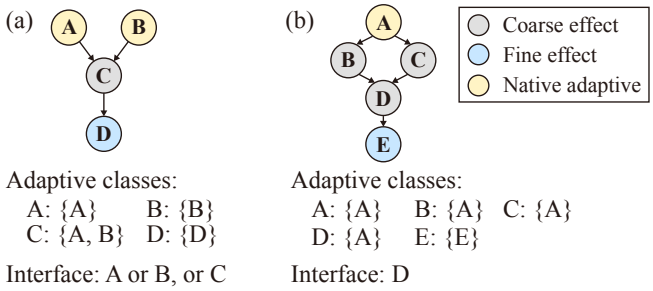


Figure 10: *Two shader DAGs. Each effect's adaptive class, and the resulting coarse-to-fine-stage interface requirements are given.*

Each effect (but not individual language statements) is annotated with a rate qualifier that indicates which stage the effect should be evaluated in. In our example, `albedo` and `color` are fine fragment stage effects. (The `out` keyword indicates that `color` is the final output of shading, and must be evaluated in the fine stage.) `diffuse` and `specular` are coarse stage effects, so their evaluation will first be attempted in the coarse stage.

Coarse effects can optionally determine when their evaluation at coarse rate is insufficient for high image quality and indicate this condition using the language's `refine` statement. Notice that both `diffuse` and `specular` reach their `refine` statements when refinement conditions are met. This indicates that the effect will be re-evaluated in the fine stage before its value can be used.

5.2 Compilation to a Multi-Stage Pipeline

Given a shader program written in this effect-based language, the challenge is to generate optimized code for our multi-rate pipeline. Specifically, while it is the programmer's responsibility to partition shading into effects and design adaptive predicates, it is the compiler's responsibility to synthesize optimized coarse and fine fragment shaders, manage refinement flags, and determine the required interface between the two pipeline stages. Compact interface generation is particularly important since it has direct impact on the size of inter-stage buffers. Our compilation process is described below.

Phase 1: adaptive class assignment. In order to determine the interface between coarse and fine pipeline stages, compilation must determine what effects will be executed in each stage. Compared to prior work scheduling declarative shading programs, a unique challenge of our language is that the `refine` construct creates situations where it is not known until runtime what stage an effect may execute in. This is particularly complicated because other effects (even ones without `refine` statements) can depend on this effect, and thus the stage of their execution is also not statically known. We refer to all effects whose stage of execution cannot be statically determined as *adaptive effects*. For clarity, we refer to adaptive effects that contain a `refine` statement as *native adaptive effects*.

The first step is to statically figure out the conditions that require an effect to be evaluated in the fine fragment stage. The answer is trivial for effects annotated with the `fine` rate qualifier, since by definition they are always executed in the fine fragment stage. For all other effects, fine fragment stage execution is required if the effect's execution reaches a `refine` statement or if at least

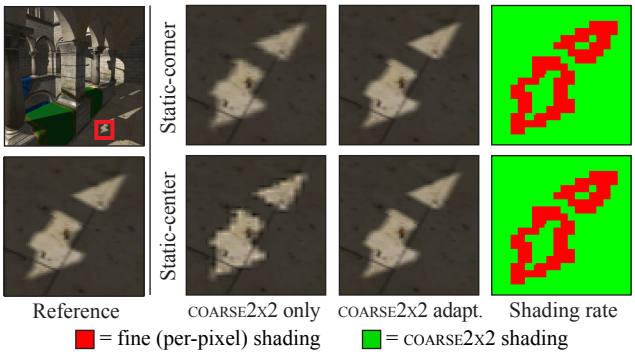


Figure 11: *The shadow edge is severely undersampled by the static-corner scheme (column 2-top) and error due to extrapolation during resampling is clearly noticeable (see hard edge artifacts) when using static-center positions (column 2-bottom). When adaptive logic is enabled, the adaptive shadow predicate triggers refinement, yielding a high-quality result.*

one of the effect's dependencies needs to be executed in the fine stage. For example, in Figure 10-a, the coarse effect C depends on native adaptive effects A and B. Therefore, if either A or B require refinement, C will execute in the fine fragment stage.

For each effect, we represent the conditions for its fine fragment stage execution in a set referred to as the effect's **adaptive class**. The adaptive class for an effect with a `fine` rate qualifier is the set containing the effect. The adaptive class for all other effects is the set of native adaptive effects upon which it directly or indirectly depends. Intuitively, for these effects, the adaptive class enumerates all potential `refine` statements that may cause the effect to require execution in the fine fragment stage. The adaptive classes of all effects are given in Figure 10.

Phase 2: Interface generation. Given the adaptive class of all effects, we then generate the coarse-to-fine stage interface. Intuitively, values that must be passed from the coarse fragment to fine fragment stage (and thus must be represented in the interface) are effects that are computed in the coarse stage but consumed by effects executed in the fine fragment stage. Thus, the compiler must statically determine if there is *any* dynamic execution sequence that results in an effect and its dependencies being evaluated in different stages.

This condition is determined by comparing the adaptive class of an effect with that of its dependencies. Specifically, if effect A references effect B (i.e. there is an edge in the DAG that goes from B to A) and the adaptive class of A contains effects *not* in the adaptive class of B, then the result of B must be passed over the interface. In other words, if the conditions for B executing in the fine fragment stage are *less strict* than the conditions for A, then it is possible for A to execute in the fine fragment stage and B to execute in the coarse stage. Communication of B across the coarse-to-fine stage boundary is necessary in this case, and so space must be reserved for B in the interface. Figure 10 illustrates two example shader DAGs, the adaptive class for all effects, and the resulting interface definitions. In example (a) even though any of A, B, or C may be passed over the interface, only one of these values is passed in any one shader invocation. Interface storage is optimized accordingly using register allocation techniques (similar behavior is true in example (b), where either A or D may be communicated in any one instance).

In addition to determining values that must cross the interface, we also assign control flag slots to each native adaptive effect.

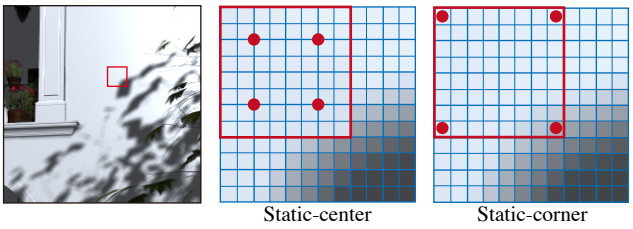


Figure 12: *Corner sample positions yield a more robust difference predicate in our adaptive shadow effect. The edge receiver predicate detects the presence of shadow variation between samples, so it is preferable to take differences of occlusion values sampled from locations spanning the entire block. (See algorithm in Figure 8)*

Phase 3: Code generation and optimization. Once the interface is defined, code generation is straightforward. The compiler emits code for `fine` rate effects in the fine fragment stage, code for adaptive effects in both stages, and code for all remaining effects in the coarse fragment stage in DAG topology order. Code is also emitted to marshal data in and out of the interface and to set and check refinement flags. Applying standard compiler optimizations (e.g., dead-code elimination) to the resulting code removes unnecessary predicate evaluation from the version of adaptive effects emitted to the fine shader.

6 Evaluation

To evaluate the shading work reduction possible using adaptive, multi-rate shading as well as the corresponding output image quality, we ran experiments on the six scenes shown in Figure 13. Our shading model features diffuse and specular lighting terms and all scenes, with the exception of WAREHOUSE, feature one shadowed light source (5×5-tap PCF). (WAREHOUSE features 6 additional unshadowed sources to add lighting complexity.) The diffuse, specular, and shadowing effects may trigger refinement using the predicates discussed in Section 4. Surface albedo textures are always sampled at per-pixel rates. This baseline shader is simple, yet up to 91% of the instructions may potentially be executed in the coarse stage. More advanced lighting effects, such as global illumination lighting, soft shadows, etc. are candidates for coarse sampling and would drive this number even higher.

We performed all experiments using a CPU-based graphics pipeline simulator that models wide SIMD execution. All renderings were performed at 2560×1440 (modeling the native display resolution of modern high-end 10-inch tablets) unless otherwise specified. 8× MSAA is enabled to minimize geometric edge aliasing.

6.1 Image Quality

We rendered animations of all six scenes and inspected the quality of the resulting images. Specifically, we looked for aliasing due to undersampling of adaptive effects or popping due to an adaptive effects transitioning between coarse and fine stage sampling in consecutive frames. When using the adaptive predicates described in Section 4 we find it difficult to notice *any differences* between multi-rate images and corresponding full-rate renderings. Inspection of magnified difference images shows that, except for isolated pixels typically on silhouettes, most pixels are almost numerically identical. (It is possible to nearly eliminate these numerical differences by using more aggressive adaptive thresholds, but in our opinion, the imperceptible gain in image quality did not justify the increase in shading work.) We refer the reader to the video accompanying this paper to compare the outputs of adaptive multi-rate and traditional GPU shading (as well as difference images).

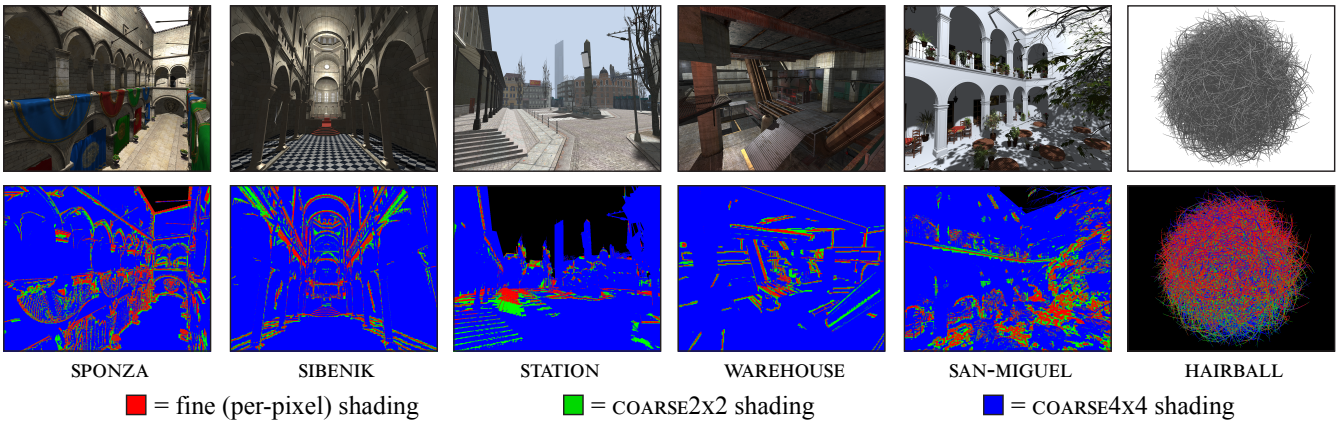


Figure 13: *Top: Six test scenes used to evaluate multi-rate shading pipeline configurations. Bottom: Visualization of the shading frequencies that result from adaptive logic during* DYNAMIC4X4 *mode shading.*

In our experiments, we found that in the presence of adaptive shading, corner sampling is often the preferred sampling strategy for two reasons. The first is that even though center sampling effectively provides twice the sampling density during coarse fragment shading, when a shader's frequency content is sufficiently high to justify the higher sampling rate, extrapolation error introduced when resampling from the center sampling positions to block edge pixels is noticeable. In other words, in situations where corner sampling requires refinement due to undersampling, center sampling will require refinement to avoid extrapolation error. Thus, regardless of sampling pattern, refinement is necessary to achieve high image quality. Figure 11 shows one example of this case by highlighting a shadow edge from SPONZA. The second column displays rendered results when adaptivity is disabled. The image generated using corner sampling (top) has undersampling artifacts. In contrast, the center-sampled result (bottom) has sharp edges due to extrapolation during resampling. However, when adaptivity is enabled, when using either sampling technique, adaptive predicates detect high frequency change and trigger refinement yielding an image that closely matches the reference (column 3).

A second reason to use corner sampling is that positioning sample points to span the entire coarse block can improve the robustness of refinement decisions. For example, Figure 12 shows a situation where an entire coarse sampling block (COARSE4X4 configuration) falls within one pixel of the active shadow map. As described in Section 4, our shadow predicate uses only a value difference check in this case (since no shadow edge may lie in between the samples). While the center sample positions (center panel) detect no variation of shadowing in the block, the more conservative corner positions (right) do, triggering refinement. (Recall the edge receiver predicate serves to handle cases where detail falls in between the sample points.) For these reasons, we use coverage-bbox sampling in all of our results, since it provides the corner sampling property in cases of full-block coverage, and minimizes attribute extrapolation error in situations of partial coverage. Other adaptive shading effects may prefer alternative sample positions.

6.2 Performance

Figure 16 plots the total number of screen-space shading instructions executed by the pipeline (totaled across coarse + fine fragment processing) for all three coarse-stage configurations and all test scenes (camera angles were manually chosen to be difficult candidates for multi-rate shading). We separate vector arithmetic instruction counts (shown in blue) from texture fetch requests (shown

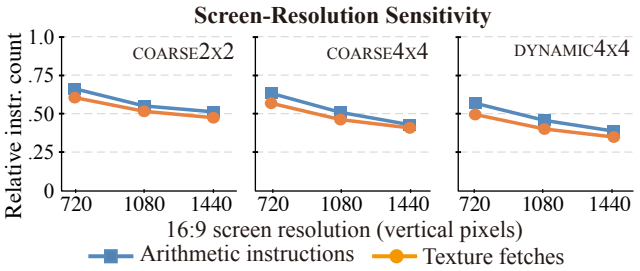


Figure 14: *The performance benefit of multi-rate shading grows with screen resolution (values averaged over all scenes).*

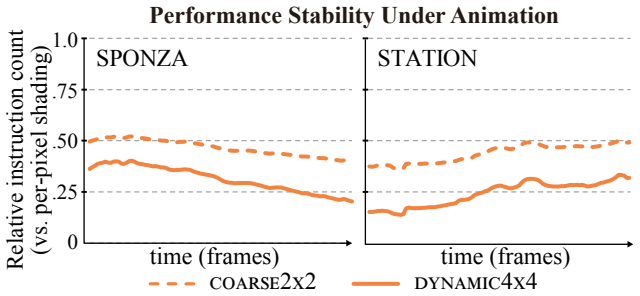


Figure 15: *The performance benefit of adaptive, multi-rate shading costs is relatively stable from frame-to-frame. Relative instruction counts achieved using multi-rate shading over 240-frame animations are shown.*

in orange), and normalize values to counts produced by a traditional GPU pipeline. Therefore, lower values indicate a greater reduction in shading work. The simulated pipeline is configured to execute 16-wide SIMD instructions so these total instruction counts reflect both the overhead of SIMD-divergence as well as additional work performed by adaptive predicates. (16-wide SIMD execution is representative of current Intel and AMD GPUs. A parameter study of different SIMD widths is discussed in Section 6.3.)

In many configurations, the adaptive, multi-rate pipelines execute significantly fewer shading instructions. COARSE2X2 performs 38 to 82% as much shading as a traditional pipeline. Arithmetic instruction counts are consistently under 50% for the larger triangle scenes (recall the minimum amount of shading, if all shading work was performed in the COARSE2X2 configuration's coarse stage, is

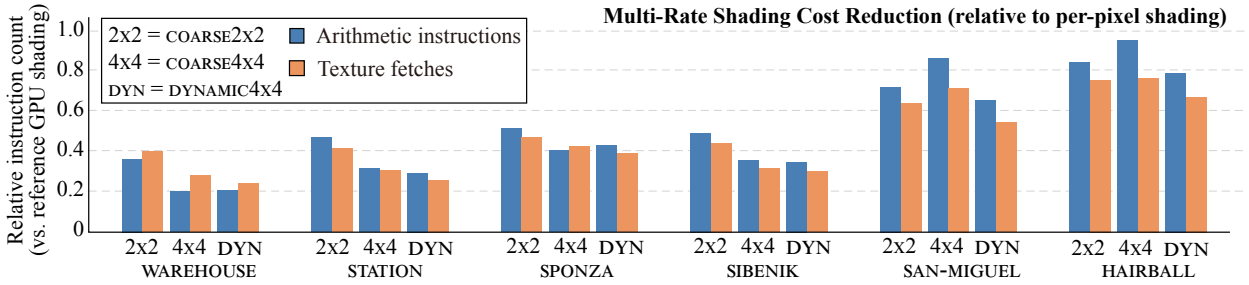


Figure 16: *Multi-rate shading decreases the total number of screen-space shading instructions (arithmetic and texture) as compared to a traditional, non-adaptive GPU pipeline. Arithmetic instruction counts decrease by up to a factor of five in the* WAREHOUSE *scene. The benefit of multi-rate shading decreases in scenes with very small triangles (*HAIRBALL*). All scenes are rendered at 2560×1440 resolution with 8× MSAA and 16-wide SIMD execution.*

25%). COARSE4X4 reduces shading costs further, often executing under 30% as many arithmetic instructions for large-triangle scenes (as low as 18%–over a factor of five reduction–on WAREHOUSE).

As expected, scenes containing more high frequency detail require more shading. For example, SANMIGUEL generates higher relative instruction counts because the tree's detailed shadows require a higher percentage of refinements. Since triangles are shaded independently in the pipeline, the benefit of multi-rate shading also decreases if scene triangles are small and thus generate sparse coarse-block coverage. For this reason, all multi-rate schemes shade nearly 80% as much as the baseline in the HAIRBALL small-triangle stress test, which has an average triangle area of 5.7 pixels. For similar reasons, the benefit of multi-rate shading diminishes slowly with decreasing screen resolution as shown in Figure 14.

Dynamically adapting the rate of coarse shading (DYNAMIC4X4) provides benefit in SANMIGUEL and HAIRBALL where intricate shadows are not adequately captured by sampling once per 4×4 pixel block, but are often captured sampling once per 2×2. However, in most other situations, we find that refinement in the COARSE4X4 configuration often implies need for refinement of the corresponding blocks in the COARSE2X2 configuration. Thus the benefits provided by the extra complexity of the DYNAMIC4X4 scheme are marginal. In the specific case of our shaders, DYNAMIC4X4 does reduce executions of shadow computations (it reduces overall texture fetches compared to COARSE4X4).

A major concern in interactive applications is not just high performance, but predictability of performance. To assess the performance stability of multi-rate execution, Figure 15 plots arithmetic instruction count reduction over 240-frame animation paths through the SPONZA and STATION scenes (see video for these paths). While the relative benefit of multi-rate shading clearly varies over the duration of the animation, frame-to-frame variation in performance is low. (These graphs also confirm our performance evaluation in Figure 16 utilized near-worst case frames.)

6.3 SIMD Scheduling

To evaluate SIMD efficiency resulting from various pipeline scheduling strategies, we ran simulations using 8 to 64-wide SIMD vectors. For each SIMD configuration we evaluated the (1) single-thread-of-control, (2) two-phase with inter-stage buffering, and (3) two phase with buffering and fine-fragment sorting strategies discussed in Section 3.3. Results are shown in Figure 18.

There are significant efficiency benefits from repacking shading work in between the coarse and fine stages when executing under the COARSE4X4 configuration. For example, COARSE4X4 shading using single-thread-of-control scheduling on a 32-wide SIMD machine results in nearly 1.8 times more instructions executed than with two-phase scheduling. As predicted, this is due to divergence resulting from empty vector lanes arising from partially covered 8×8-pixel blocks. Modest benefits are observed from inter-stage buffering when executing the COARSE2X2 configuration on a wide SIMD machine. In a 32-wide COARSE2X2 configuration, work repacking via inter-stage buffering reduces instruction count by approximately 10% compared to single-thread-of-control scheduling. Surprisingly, sorting fragments based on adaptive flags (green bars) has minor performance benefit in COARSE2X2 configurations (at most 4%). The benefit raises to 10% in the COARSE4X4 pipeline. We note that the relative benefits of inter-stage buffering/repacking and fragment sorting execution increase if screen resolution is lowered from our 2560×1440 test resolution.

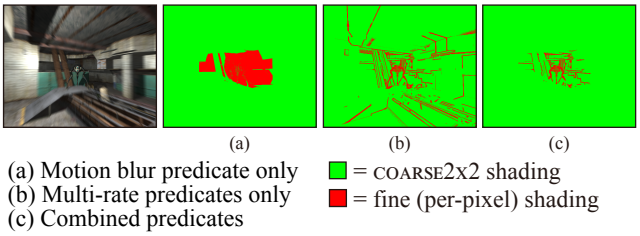


Figure 17: *Both the motion blur predicate (a) and our blur-agnostic multi-rate predicates (b) reduce the cost of shading on their own, however the benefit of combining the two methods only reduces instruction count by 3.2% when compared to the multi-rate optimizations alone (c).*

Overall, we conclude that many performance benefits of COARSE2X2 adaptive, multi-rate shading can likely be realized with simple hardware implementations. However, realizing the shading cost reductions possible with COARSE4X4 shading requires full inter-stage buffering to utilize vector hardware efficiently.

6.4 Motion Blur

In our experiments, we also studied the interaction between post-process motion blur and multi-rate shading. An unanticipated result is that for our scenes, the shading cost reductions realized from multi-rate shading (in the absence of blur) are sufficiently large that the additional benefit of also considering blur in refinement calculations is marginal. In fact, the cost reductions described in the prior sections are similar in magnitude to those of prior work that adapts shading rate based on blur [Vaidyanathan et al. 2012].

Figure 17 shows a frame from the WAREHOUSE scene with a rapidly forward-moving camera and compares the benefits of evaluating the

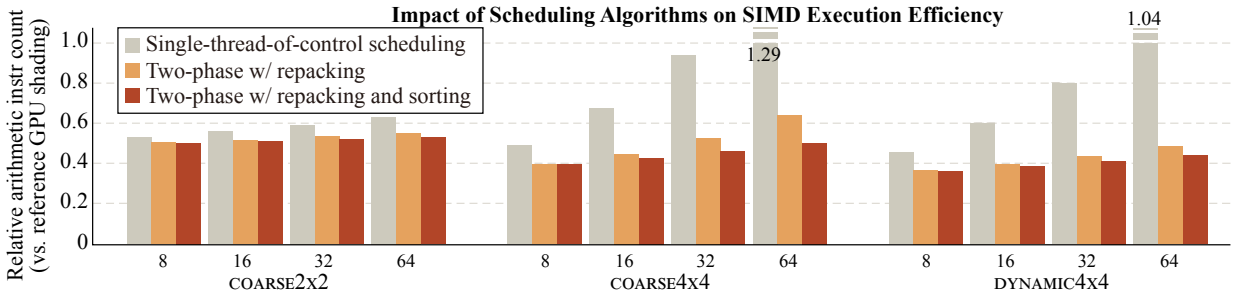


Figure 18: *Two-phase scheduling of multi-rate shading computations has only modest efficiency benefits in a* COARSE2X2 *pipeline configuration, but the optimization is critical for maintaining high efficiency under* COARSE4X4 *or* DYNAMIC4X4 *execution, particularly when the machine's SIMD width is large. Results are averaged over all scenes.*

entire shading function at coarse rate in regions of high blur (a) with the results of adaptive, multi-rate shading in the absence of blur (b). Notice in (c) that combining the two predicates only results in shading instruction counts decreasing by 3.2% in a COARSE2X2 configuration. (A drop of 45% (panel B) to 42% (panel C) of full per-pixel shading costs.) Certainly, considering blur in shading rate calculations allows refinement thresholds in multi-rate predicates to be relaxed (or may allow new effects, such as bump-mapped surfaces, to become candidates coarse-rate sampling). However, we find that using multi-rate shading mechanisms to decrease sampling of low frequency shading effects, regardless of blur conditions, is in general a more effective (more significant cost reductions, and more predictable–it is applicable to all frames) way to reduce shading costs during rendering.

7 Discussion

In anticipation of a need for substantially increased shading efficiency in future GPUs due to high-resolution displays, mobile graphics processing, and increasingly complex shaders, we have conducted an extensive study of adaptive, multi-rate shading in the context of improving modern GPUs. We find that simple pipeline mechanisms used in conjunction with our adaptive techniques can reduce the cost of shading during rendering by at least a factor of two (COARSE2X2). More complex pipeline scheduling (COARSE4X4 or DYNAMIC4X4) can reduce shading costs, on average, to more than three and sometimes up to a factor of five. In this paper we primarily demonstrated these benefits for scenes with smoothly varying lighting, but native GPU support for multi-rate shading mechanisms also stands to be valuable in situations where sparse sampling is acceptable due to significant camera motion blur or precise tracking of viewer gaze [Guenter et al. 2012].

While our initial motivation was to investigate pipeline extensions for multi-rate rendering in a forward rendering setting, the proposed two-stage pipeline shading mechanisms, adaptive shaders, and the effect-based language used to conveniently express them are equally valuable when performing multi-rate shading in a deferred rendering context. Further, although presented in the context of two-stage, multi-rate shading, our proposed language extensions and compilation techniques will generalize to pipelines with additional shading stages, for example, if shading effects were also performed at per-multi-sample granularity.

Of course, shading more sparsely than per-pixel rates is only viable for effects that exhibit slow screen variation. High-frequency phenomena, such as detailed displacement, normal, or bump mapped surfaces, are not good candidates for coarse shading. However, the use of bump or normal maps does not imply high surface detail exists throughout a surface, and one area of future work involves the design of adaptive predicates that detect low-screen variation in the appearance of bump-mapped surfaces. Still, many effects driving up the cost of shading are low-frequency effects, such as global illumination. These effects are strong candidates for coarse rate shading and the design of efficient and robust adaptive logic for them will be important to consider in the future.

8 Acknowledgments

Support for this research was provided by the National Science Foundation (IIS-1253530) and by gifts from NVIDIA, Intel, and Qualcomm. The Ogre model is available from the Keenan Crane model archive. The WAREHOUSE and STATION scenes were created by Valve Software. We would like to thank Anjul Patney, Eric Lum, and Henry Moreton for valuable conversations.

References

AKELEY, K. 1993. RealityEngine graphics. In *Proceedings of SIGGRAPH 93*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 109–116.

ANNEN, T., MERTENS, T., SEIDEL, H.-P., FLERACKERS, E., AND KAUTZ, J. 2008. Exponential shadow maps. In *Proceedings of Graphics Interface 2008*, GI '08, 155–161.

CLARBERG, P., TOTH, R., AND MUNKBERG, J. 2013. A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. Graph. 32*, 4 (July), 141:1–141:10.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, 161–165.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph. 29*, 4 (July), 67:1–67:8.

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 the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, SIGGRAPH '95, 343–350.

GUENTER, B., FINCH, M., DRUCKER, S., TAN, D., AND SNYDER, J. 2012. Foveated 3d graphics. *ACM Trans. Graph. 31*, 6 (Nov.), 164:1–164:10.

KIRCHER, S., AND LAWRANCE, A. 2009. Inferred lighting: Fast dynamic lighting and shadows for opaque and translucent objects. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, ACM, Sandbox '09, 39–45.

LAURITZEN, A. 2010. Deferred rendering for current and future rendering pipelines. In *ACM SIGGRAPH 2010 Courses: Beyond Programmable Shading II*. Available at `http://bps10.idav.ucdavis.edu`.

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, I3D '12, 143–150.

NICHOLS, G., PENMATSA, R., AND WYMAN, C. 2010. Interactive, multiresolution image-space rendering for dynamic area lighting. In *Proceedings of the 21st Eurographics Conference on Rendering*, Eurographics Association, EGSR'10, 1279–1288.

NVIDIA CORPORATION. 2013. *TXAA - Temporal Anti-Aliasing Technology*. Available at `http://www.geforce.com/landing-page/txaa/technology`.

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

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

PENNER, E. 2011. Shader amortization using pixel quad message passing. In *GPU Pro2 : advanced rendering techniques*. A K Peters, Ltd., 349–367.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, SIGGRAPH '01, 159–170.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled sampling for graphics pipelines. *ACM Trans. Graph. 30*, 3 (May), 17:1–17:17.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph. 21*, 4 (Aug.), 283–291.

SCHERZER, D., YANG, L., MATTAUSCH, O., NEHAB, D., SANDER, P. V., WIMMER, M., AND EISEMANN, E. 2012. Temporal coherence methods in real-time rendering. *Computer Graphics Forum 31*, 8, 2378–2408.

SHEBANOW, M., 2013. An evolution of mobile graphics. High Performance Graphics 2013 Keynote Address.

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.

TATARCHUK, N., TCHOU, C., AND VENZON, J. 2013. Destiny: From mythic science fiction to rendering in real-time. In *ACM SIGGRAPH 2013 Courses: Advanced in Real-Time Rendering in Games*.

VAIDYANATHAN, K., TOTH, R., SALVI, M., BOULOS, S., AND LEFOHN, A. 2012. Adaptive image space shading for motion and defocus blur. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, Eurographics Association, EGGH-HPG'12, 13–21.

YANG, L., SANDER, P. V., AND LAWRENCE, J. 2008. Geometry-aware framebuffer level of detail. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, Eurographics Association, EGSR'08, 1183–1188.