Efficient Adaptive Deferred Shading with Hardware Scatter Tiles

AGATHA MALLETT, University of Utah
CEM YUKSEL, University of Utah
LARRY SEILER, Facebook

Adaptive shading is an effective mechanism for reducing the number of shaded pixels to a subset of the image resolution with minimal impact on final rendering quality. We present a new scheduling method based on on-chip tiles that, along with relatively minor modifications to the GPU architecture, provide efficient hardware support. As compared to software implementations on current hardware using compute shaders, our approach dramatically reduces memory bandwidth requirements, thereby significantly improving performance and energy use. We also introduce the concept of a *fragment pre-shader* for programmatically controlling when a fragment shader is invoked, and describe advanced techniques for utilizing our approach to further reduce the number of shaded pixels via temporal filtering, or to adjust rendering quality to maintain stable framerates.

CCS Concepts: • **Computing methodologies** → **Graphics processors**; *Rasterization.*

Additional Key Words and Phrases: deferred shading, adaptive shading, graphics hardware

ACM Reference Format:
Agatha Mallett, Cem Yuksel, and Larry Seiler. 2020. Efficient Adaptive Deferred Shading with Hardware Scatter Tiles. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 11 (August 2020), 17 pages. https://doi.org/10.1145/ 3406184

1 INTRODUCTION

The cost of fragment shader evaluations can be a bottleneck for realtime rendering applications, particularly when using complex shading models and expensive lighting evaluations. This cost is exacerbated due to overdraw with the standard forward rasterization that the GPU rendering pipeline is designed to do. Deferred shading [Saito and Takahashi 1990] provides an alternative rendering pipeline that eliminates overdraw by making sure that the (potentially expensive) fragment shader is executed only once per pixel sample. Although there is no specific hardware support for deferred shading, it can be implemented on existing GPUs by using multi-pass forward rasterization: the first pass generates a *G-buffer* that stores the necessary geometry information and the second pass computes fragment shading. Today, deferred shading can be considered an industry standard and most game engines are designed for or only support deferred shading.

However, the cost of fragment shading can be a bottleneck with deferred shading as well, particularly using high-resolution displays that are commonplace today. Among various solutions to this crucial problem, *Deferred Adaptive Compute Shading* (DACS) [Mallett and Yuksel 2018] provides unique advantages. In particular, DACS can produce visually indistinguishable results by executing the fragment shading operations only for a small fraction of the pixels. DACS achieves this by adaptively deciding which pixels should be shaded, without relying on previously-rendered images.

Authors' addresses: Agatha Mallett, University of Utah, agatha@geometrian.com; Cem Yuksel, University of Utah, cem@cemyuksel. com; Larry Seiler, Facebook, larryseiler@fb.com, larry.seiler@aranfell.com.

While DACS provides a net performance gain by adaptively reducing the fragment shading computations, the reported render-time reductions are nowhere near the substantial reduction in the fragment-shading operations it achieves. This is because its software implementation on current GPUs is burdened by inefficiencies within its memory-access patterns. More specifically, DACS significantly inflates the memory bandwidth used for accessing the G-buffer, provides poor cache utilization for the G-buffer, and requires random read/write access to the output framebuffer.

In this paper, we propose a dedicated deferred shading pipeline and relatively minor changes to the GPU hardware for achieving an efficient implementation of adaptive deferred shading. These changes completely eliminate these inefficiencies of DACS. More specifically,

- We present a new swizzling order for G-buffer storage to ensure that adaptive deferred shading leads to a reduction in memory bandwidth use for the G-buffer and achieves optimal cache utilization.
- We introduce a new work-scheduling approach based on *hardware scatter tiles*, a new tile structure that optimizes framebuffer accesses and minimizes corresponding data movement.

We analyze the potential improvements provided by our methods and present performance and quality comparisons to hardware variable-rate shading. Our tests also show that compute shader implementations of deferred shading can lead to unreliable performance due to the complexities of work scheduling.

Finally, we describe possible API changes that would simplify the implementation of adaptive deferred shading by introducing the concept of a *fragment pre-shader* that allows programmatically controlling when the expensive portion of a fragment shader should be computed. We discuss possible use-cases of adaptive deferred shading with hardware support to achieve further reduction in fragment shading computations via temporal filtering and stable framerates by adaptively throttling the rendering quality when needed.

2 RELATED WORK

Deferred shading [Saito and Takahashi 1990] has become nearly ubiquitous in video games, as these move toward more-complex and -computationally-expensive shading algorithms involving realistic, physically-based materials and a large number of light sources. Although deferred shading addresses the additional cost of overdraw in forward rendering, executing the fragment shader for each pixel is often still the performance bottleneck.

2.1 Reducing Fragment Shader Invocations

The number of fragment shader invocations can be reduced by rendering a lower-resolution image and then upscaling it during post-processing. Another common approach is *checkerboard rendering* [Mansouri 2016; Vlachos 2016; Wihlidal 2017]. Checkerboard rendering separates the pixels into two groups using a checkerboard pattern of various sizes: commonly MSAA subpixels are (ab)used to make a subpixel pattern. The samples in one group are shaded and the skipped samples of the other group are later filled in using a temporal reconstruction filter. The group that is shaded is alternated each frame.

The concept of multi-sample antialiasing can be generalized by decoupled shading [Ragan-Kelley et al. 2011], which separates shading rate from visibility. It can be implemented on current GPUs as a cache that simply reuses the shading result of "similar" visibility samples [Liktor and Dachsbacher 2012], although better performance can be achieved via dedicated hardware support [Clarberg et al. 2014, 2013]. In general, any method that separates shading and visibility can be considered a form of decoupled shading [Akenine-Möller et al. 2007; Crassin et al. 2015; Kerzner and Salvi 2014; McGuire et al. 2010].

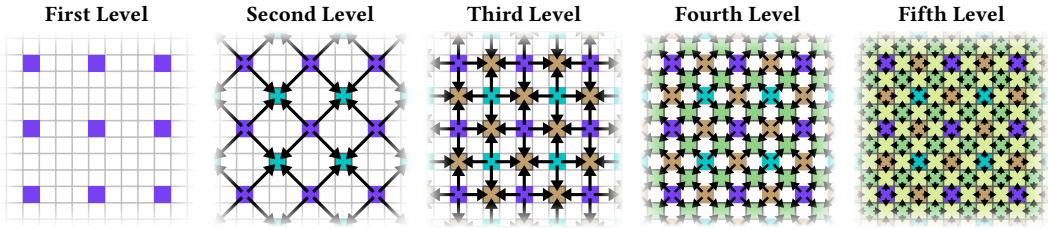


Fig. 1. *Subdivision levels of DACS [Mallett and Yuksel 2018] for an initial grid step size of* 4*. For each additional pixel added in each level, DACS considers "neighbor" pixels from the previous levels, as indicated by the arrows.*

He et al. [2014] proposed modifications to the GPU rendering pipeline to support multi-rate shading, which first shades a set of coarse fragments and then adaptively refines them, as needed. This allows dynamically adjusting the shading rate based on the content. Vaidyanathan et al. [2014] presented an alternative approach that extends the concept of multi-sample antialiasing to achieve a lower-resolution shading rate than the target image resolution.

The *variable-rate shading* (VRS) feature in NVIDIA's Turing architecture [Burgess 2020] can be considered a variant of this approach. VRS allows the user to specify a single shading rate for separate tiles of pixels. In the most-recent version, the tiles can be 8, 16, or 32 pixels large (16 is typical). The available shading rates range from 8× supersampling to 1 shade per 4×4 block ($1/16\times$ undersampling). VRS requires the developer to specify the desired shading rate for each tile in advance; in a deferred context, this information could be generated by an additional pre-pass on the G-buffer. Unfortunately, the adaptivity provided by VRS is limited, since the shading rate is controlled per-tile (as opposed to making decisions individually for each pixel). It does not interpolate fragment shading output (and simply copies the result to as many pixels as specified), it cannot use the fragment shading output of the currently rendered frame, and it requires the programmer to pick the desired shading rate prior to rendering. VRS can also specify the shading rate per-triangle, but this does not directly apply when shading in a deferred context. On the other hand, VRS is flexible enough to work with forward, as well as deferred, shading.

Stengel et al. [2016] discusses an algorithm that shades a percentage of pixels for foveated rendering in the context of deferred shading. The colors of the pixels that are not shaded are approximated by interpolating the shaded pixel colors. Unfortunately, a practical implementation on an existing or speculative hardware is not described.

2.2 Deferred Adaptive Compute Shading (DACS)

We use the adaptive subdivision strategy of DACS [Mallett and Yuksel 2018], which is an algorithm for shading less than one sample per pixel (undersampling) when shading in a deferred context on existing GPU hardware. This method divides the pixels into different *shading level*s based upon their positions in a subdivision pattern (Figure 1). All pixels of the first (coarsest) level are shaded unconditionally. Each pixel of the following (finer) levels are either shaded or else their shading results are approximated by interpolating the four *neighbor* pixels from the previous level. The decision to shade or interpolate is made dynamically at per-pixel granularity, allowing for fine-grained and on-the-fly subdivision.

The subdivision order for processing the pixels is achieved by handling the fragment shading operations within a compute shader. First, all pixels of the first level are shaded (in a scanline order) and then the pixels of the subsequent levels are either shaded or interpolated from previously computed pixels of the previous levels. To maintain warp coherence and achieve efficiency within

SIMD groups of threads, pixels that must be shaded are first enqueued into a warp-local ring buffer; pixels are only shaded when enough accumulate into this buffer. This ensures that, even though many pixels are skipped, the warp always shades pixels fully-filled.

The decision to shade or interpolate a pixel is handled based upon an arbitrary user-defined *criterion*. This allows using detailed and content-specific information, such as material ID or 3D position of the pixel in question, as well as data from the four neighbor pixels of the previous level. Though, it has been shown that simply relying on the color variation of the four neighbor pixels can provide acceptable quality, even in complex scenes.

3 INEFFICIENCIES OF DEFERRED ADAPTIVE COMPUTE SHADING

DACS can produce visually similar or indistinguishable results to brute-force shading, in many cases, by executing the fragment shading operations only for a relatively small fraction of the pixels. However, its implementation on current GPUs has some important drawbacks that hinder its performance improvements.

First of all, executing fragment shader operations within a compute shader that reorders the computation following a special subdivision order leads to additional shader code complexity. Much of the implementation is provided by the authors, but care must still be taken e.g. for derivatives when computing mipmaps.

More significantly, the order in which the pixels are processed leads to an ineffective use of GPU caches for accessing the G-buffer (for shading) and the framebuffer (for the user criterion). While GPUs excel at hiding memory latency, the order in which the pixels are processed inflates the memory bandwidth usage. This is a substantial drawback, considering that memory bandwidth can be a limiting factor for performance and a significant contributor to power consumption.

3.1 Analysis of DACS G-Buffer Accesses

Using a compute shader to generate DACS results requires reading the G-buffer up to five times, depending upon how it is stored. This is an inefficiency that our hardware-assisted method eliminates, so that the G-buffer need only be accessed once.

The following analysis assumes for simplicity that all pixels need to be rendered. It also assumes that the G-buffer is large enough so that by the time we have reached the end, the first accesses have fallen out of the cache.

Consider a G-buffer that is stored in classic tiled order, so that each 64B (a typical cacheline size) of 32-bit pixels covers a 4×4 aligned tile. We see in Figure 2a that each 4×4 tile contains pixels from all five DACS levels. Therefore, each of the five passes ends up reading the entire G-buffer. The total memory bandwidth is therefore 5× the size of the G-buffer. It would be even larger if the read cache cannot hold multiple scanlines of G-buffer data.

The read bandwidth is lower if the G-buffer is stored in scanline order, so that each read returns 16×1 pixels. With scanline order, pixels of the first and second level are in just one quarter of the scanlines, third and fourth level are in one half of the scanlines, and fifth level are in all of the scanlines. So the five passes access those fractions of the G-buffer for a total memory bandwidth use that is 2.5× the size of the G-buffer.

3.2 Analysis of DACS Framebuffer Accesses

Using a compute shader to write DACS results to the framebuffer is worse than the G-buffer read case, as it requires accessing the framebuffer up to nine times, depending upon how it is stored. Again, this is an inefficiency that our hardware-assisted method eliminates, replacing this with a single write to the framebuffer.

Making the same swizzled storage assumptions as for the G-buffer and 64B cachelines, processing the first level must write a result to each cacheline in the framebuffer, since each swizzled cacheline contains a first-level result. Unless the whole framebuffer fits into the on-chip cache, processing the second through fifth levels must each read the framebuffer, insert their results, and then write it back again. The result is that the entire framebuffer must be accessed nine times: four reads and five writes.

These numbers would be reduced if the framebuffer were stored in scanline order; however, this would likely result in significantly slower performance for all other accesses to the framebuffer.

3.3 Analysis of Work-Group Occupancy

One major inefficiency in DACS is the overhead of doing the pixel scheduling in software. The logic itself is fairly involved (though still relatively small compared to a typical shading operation). More problematically is that execution in software creates a subtle implementation issue with regards to work-group allocation.

The DACS shader loads from the G-buffer to execute the user function, and also loads other textures during shader evaluation. For these texture reads, ideally the multiprocessor could switch to a different warp in order to avoid stalling. Unfortunately, in the simple case, there is exactly one warp per multiprocessor because the DACS shader's work-group size is the same size as a warp; it is intended to encapsulate all shading operations done at a multiprocessor level. Thus, the multiprocessor ends up being unable to hide the latency.

The obvious approach is to simply allocate more instances of the DACS shader to each multiprocessor: the multiprocessor switches among them according to the GPU's scheduling logic whenever a stall happens, effectively emulating additional multiprocessors. The original DACS paper took this approach. Unfortunately, issues can arise where some warps get starved, and lag far behind others, leading to image artifacts or unstable shading rates (which were reflected in their results).

Another approach to attempt to address these issues is to make the work-group-local pixel ring buffer in DACS into a global object, but in practice this introduces significant synchronization overhead. Multiple global objects amounting to some partition of the framebuffer might fare better, but this still increases memory traffic outside the multiprocessor.

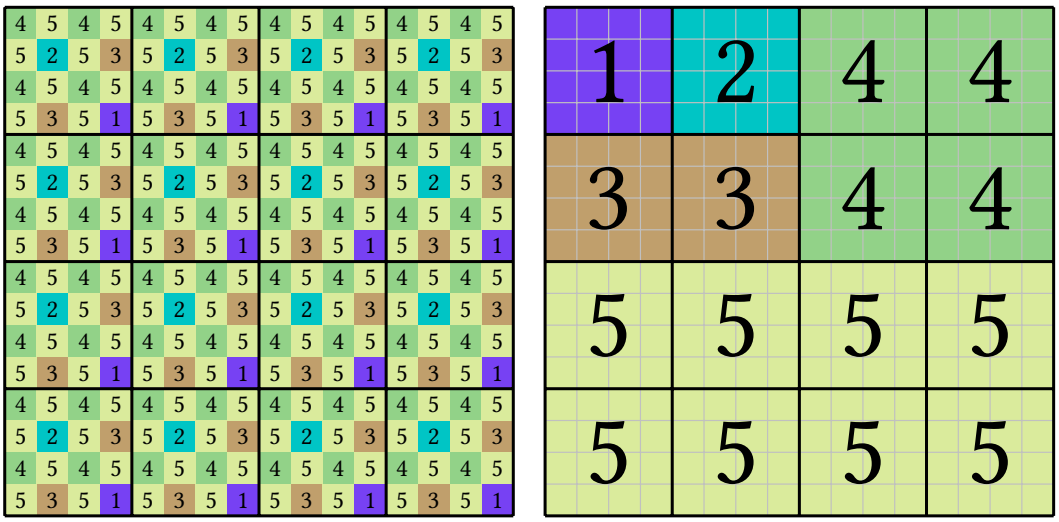
The best approach is likely to increase the compute shader's work-group size and readjust the DACS scheduling logic, but this increases complexity—and although barriers would allow the warps within the work-group to stay relatively in-sync, there is no guarantee the work group will execute on a single multiprocessor, leading to unintentional memory traffic between multiprocessors.

4 HARDWARE SUPPORT FOR EFFICIENT ADAPTIVE DEFERRED SHADING

The adaptive subdivision pattern in Figure 1 provides an effective mechanism that minimizes the full fragment shader evaluations while maintaining the final image quality. Most of the inefficiencies of its implementation on existing GPUs are related to the additional data movement it generates, needing access to both the G-buffer and the framebuffer. For optimizing its behavior and minimizing the corresponding data movement we propose a swizzling scheme that automatically reorders the data in the G-buffer (Section 4.1) and a new computation order and on-chip structure for distributing the adaptive fragment shading work to multiple warps (Section 4.2). We also discuss related issues regarding warp synchronization (Section 4.3) and determining the tile size (Section 4.4).

4.1 G-Buffer Swizzling

Our solution to the G-buffer bandwidth problem is to change the way G-buffer data is stored in memory. Typically, pixel data is stored in 2D footprints of increasing sizes, an approach referred to as *swizzling* or *tiling*. Given 32-bit pixel values, each 4×4 box stores 64B of G-buffer data, a typical



(a) G-buffer pixels and their levels **(b) Our swizzled G-buffer storage**

Fig. 2. *Swizzle pattern for optimized G-buffer reads:* ***(a)*** *a* 16×16 *array of G-buffer pixels, marked with their level numbers in the adaptive subdivision order, and* ***(b)*** *the G-buffer pixels swizzled so that each* 4×4 *block contains pixels from a single level. As a result, a single G-buffer read accesses only the required DACS shading level.*

cacheline size on modern hardware. Sixteen such accesses read a 16×16 swizzled array of pixels (Figure 2a). With this swizzle pattern, the width and height of a pixel array, such as a framebuffer, are padded to multiples of 16 so that the 2D array contains an integral number of these swizzled 16×16 tiles. Many other swizzle patterns are possible, but all share this general structure.

Swizzle patterns like this are used for rendering because they are more efficient when accessing a 2D shape, such as a triangle or rectangle, as opposed to storing pixels in a scanline order. However, this pattern is highly inefficient for G-buffer reads when used with the computation order of our adaptive subdivision pattern (see Section 3.1).

Our solution is to write the G-buffer using the pattern in Figure 2b, such that each block of 16×16 pixels is swizzled as shown. In this pattern, each 4×4 box contains pixels from just one level. E.g. the upper left box contains pixels from the first level, the box to its right contains pixels from the second level, and the two boxes below them contain third-level pixels from the left and right sides of the 16×16 tile. The result is that the processing for each level is able to read only pixels from the desired level, thus reducing the G-buffer read memory bandwidth by up to 5×.

The remaining issue is to define how to write to a G-buffer using this swizzle format. In this pattern, each 2×2 of computed G-buffer pixels is written into three different 64B memory cachelines and each 4×4 of computed G-buffer pixels is written into five different 64B memory cachelines. That would be a bottleneck for a simple pass-through fragment shader that produces a 2×2 pixel result every one or two clocks, but not for a fragment shader at least three instructions long. One to two instruction shaders can be supported, if necessary, by placing a merge buffer in front of the pixel cache to reduce write bandwidth into the cache.

4.2 Scatter Tiles

The key challenge with mapping deferred adaptive shading onto hardware is determining an efficient way of splitting the framebuffer into tiles, which are distributed to warps sequentially

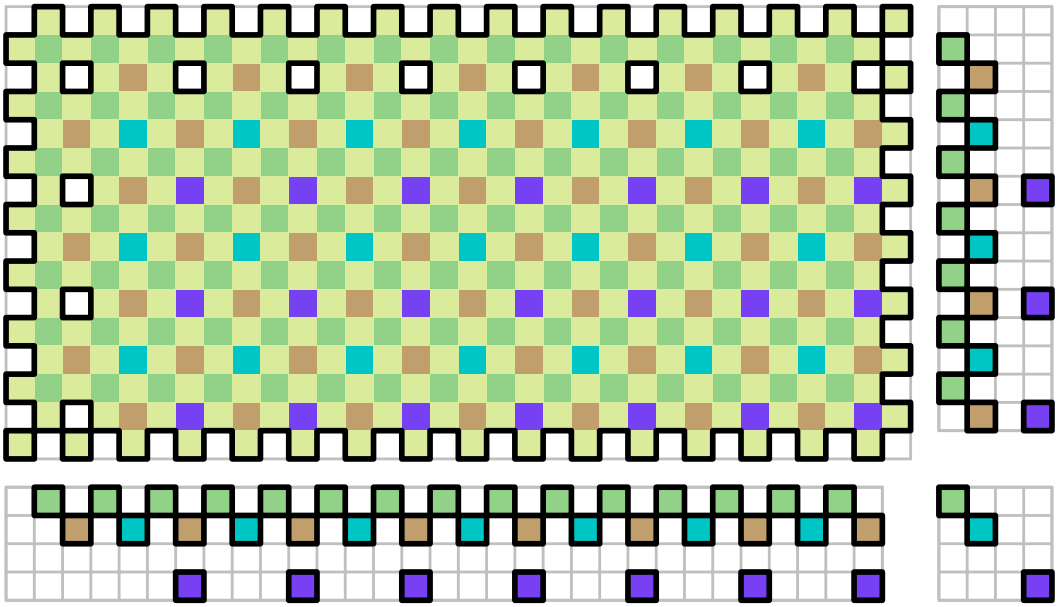


Fig. 3. *Diagram of the spatial meaning of a single "8 × 4" scatter-tile. Notice the irregular, perforated shape. The main region of the tile scales quadratically with tile size, and can be freed after being streamed out to framebuffer memory. The edge regions scale linearly with tile size, and are retained on-chip for use by no more than three neighbor tiles, interlocking into the top and left.*

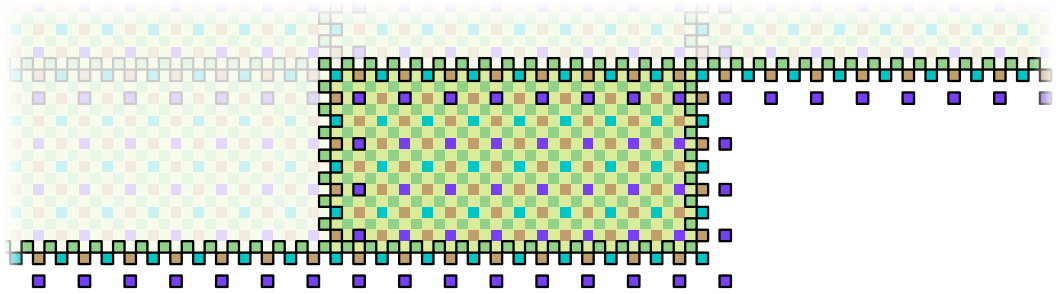


Fig. 4. *Memory residency while shading interlocking tiles; the non-faded regions would be stored on-chip. Notice that the boundary regions from previous tiles are retained for the use of future tiles. Only a single tile is being shaded here for clarity; in an actual system with multiple warps, many tiles would be shaded at the same time.*

from left-to-right and then top-to-bottom order[1]. Warps are responsible for assigning colors to all pixels within a tile. Within each tile, the adaptive subdivision algorithm is executed to completion, looping through each level sequentially.

The adaptive deferred shading algorithm requires passing data between tiles. To handle this efficiently, we propose *hardware scatter tiles*, an irregularly-shaped, self-interlocking tile with four regions: one main and three border, as shown in Figure 3. The main region of a scatter tile lives entirely within a single warp's local memory; the three border regions along the bottom, right, and

[1]Any order is possible with appropriate changes; we present it this way since it will be most familiar.

corner are also made accessible to other warps. The border and corner pieces fit into corresponding gaps in the neighbor scatter tiles to the bottom and right, so that the tiles tessellate to fill space. This scatter tile structure is designed to minimize the shared data between warps and to give each warp sufficient time to compute the shared data by limiting the shared pixels to the earlier levels of the subdivision order.

As the warp shades its tile, the pixels in the border regions become progressively filled, and are used by neighboring warps processing their tiles. Once the tile has been shaded to completion, all of its pixels are streamed out to the framebuffer memory and the warp acquires a new tile to shade. The main region is immediately deallocated on-chip, but the boundary regions can persist if the neighboring tiles have not yet been fully shaded (Figure 4). It is feasible to do this, even for the entire framebuffer, because the boundary regions are smaller, by a square root, than the main region of the tile.

4.3 Warp Synchronization

Processing multiple scatter tiles in parallel requires some simple communication between warps. In particular, when a warp needs to access the output of another warp (i.e. the border regions of the scatter tile), it needs to check whether the data is available. This requires only one-way communication and can be implemented similar to typical caches with a single bit per pixel, indicating whether the warp that owns the border region produced the data.

Ideally, when a warp accesses the output of another warp, the requested data should be available. Yet, this cannot be guaranteed. If the data is not available, the thread requesting the data can generate the data itself, stall, or process another pixel. Generating the requested data by unconditionally shading the neighboring pixel would cause extra fragment shading work. Also, in all likelihood, the other warp would produce the data earlier. Therefore, stalling appears to be a better alternative here. The best alternative, however, is shading another pixel instead. Note that only pixels near the edges of a tile need to access the border pixels of other warps, so, in most cases, there should be other work for the thread, while it waits for the data from the other warp. Also, a warp can be assigned multiple tiles at a time and switch between them to avoid stalls. Existing GPUs already operate in a similar fashion for latency hiding.

We also need to know when it is safe to deallocate the on-chip storage for the border regions. This can be implemented using a flag per border region. Note that the bottom and right border regions adjoin two warps and the corner region adjoins four. Therefore, just a few bits per border region is sufficient to indicate whether there are other warps that still need the data. The last warp that flips these bits can signal the eviction of the data from on-chip storage.

4.4 Determination of Tile Size

Using our scatter tiles, it at first seems beneficial to assign a warp a larger tile, as opposed to assigning multiple smaller tiles, because larger tiles reduce the data movement between warps, both proportionally and absolutely. However, we would also like to minimize the tile size, so that the framebuffer can be split into more tiles, providing better granularity for work distribution to multiple warps. Smaller tiles also implies a smaller warp-local storage requirement. Therefore, determining the optimal scatter tile size involves a more detailed analysis than simply considering the warp SIMD width k.

We measure the scatter tile size using (roughly) the number of 4×4 pixel blocks along its width and height, w and h, respectively. More precisely, w and h are the number of pixels in the first level along the width and height of the scatter tile (e.g. in Figure 3 $w = 8$ and $h = 4$). The number of scatter tile pixels for each shading level is shown in Table 1.

Table 1. *Number of scatter tile pixels of size $w \times h$ at each shading level.*

	Number of Pixels	Number Shared
Level 1:	hw	$w + h - 1$
Level 2:	hw	$w + h - 1$
Level 3:	$2hw$	$w + h$
Level 4:	$4hw$	$2w + 2h - 1$
Level 5:	$8hw$	0
Total:	$16hw$	$5w + 5h - 3$

Ideally, we would like to fully hide latency from the memory subsystem. The boundary between the second and third shading levels represents a worst-case. The first shading level is shaded unconditionally, and so all G-buffer accesses are perfectly predictable and can be streamed in-advance. The second shading level is shaded conditionally based on a user-defined criterion. Therefore, its memory access pattern is unpredictable. One could simply load the G-buffer unconditionally (indeed, some portion of it must be loaded regardless to compute the user criterion for the next shading level), but reducing this bandwidth is still highly desirable. The second shading level is also one of the smallest shading levels, meaning that there is the least amount of time between the completion of its first pixel, and requiring G-buffer data for the first pixel of the next level.

The criterion for a given pixel can be evaluated as soon as that pixel's neighbor pixels from the previous shading level are completed (and potentially partially or completely earlier, depending on the criterion). It is expected that the user criterion is of negligible complexity compared to the shader. Assuming no additional tiles for context switching are allowed, given an average shading rate r for the second level, latency hiding implies that no pixel in the next level could be shaded before the memory latency has elapsed. Let L_s and L_m indicate the expected fragment shading time and memory latency, respectively. For perfect latency hiding, we must satisfy the inequality

$$\left\lfloor \frac{hwr}{k} \right\rfloor L_s \geq L_m \ . \tag{1}$$

Therefore, the desirable tile size must satisfy

$$hw \geq \frac{k}{r} \left\lceil \frac{L_m}{L_s} \right\rceil \ . \tag{2}$$

Notice that data passed from neighboring tiles will also need to be available at this rate. However, this is clearly easier to attain, given that this data is entirely on-chip.

If $L_s > L_m$, which is likely given the complexity of shaders (and the fact that they themselves usually do memory accesses also), then $hw = k/r$. For example, if $k = 32$ and $r = 25\%$, then a 16×8 tile would be a desirable size for scatter tiles to hide latency.

On-chip memory and area is expensive, and it is important to check that the addition of an on-chip tile is feasible. At 3 bytes per sRGB pixel or 12 bytes per pixel (for HDR), the $16hw$ pixels in a tile take $48hw$ or $192hw$ bytes, respectively. For the above example ($w = 16$, $h = 8$), we have 6.1–25KB of on-chip storage required per tile.

We would like to accommodate multiple tiles per warp for context-switching, latency hiding for multiple memory accesses, and simple pessimism. However, even multiplying this amount by a modest factor results in a feasible on-chip storage on current GPUs. One expects that some of this on-chip memory may need to be reallocated from existing resources.

5 EVALUATION

We evaluate our methods for providing hardware support for adaptive deferred shading by comparing it to DACS (i.e. its software implementation on current GPUs), NVIDIA variable-rate shading (VRS), and brute-force shading of all pixels.

5.1 Memory Bandwidth

As we explain in Section 3.2, DACS can significantly inflate the bandwidth usage for accessing the G-buffer (up to 5×), as compared to brute-force fragment shading. Also, it requires accessing the framebuffer for reading the fragment-shading output of previous levels, thereby substantially increasing the bandwidth use for the framebuffer as well (up to 9×).

Deferred shading with VRS, on the other hand, does not have a similar bandwidth overhead. Using typical swizzle patterns with cachelines of 4×4 pixels and efficient cache utilization, VRS would read all G-buffer texels exactly once (even when shading only a single sample for a 4×4 block) and write the framebuffer only once without reading it back.

Our swizzle pattern (Figure 2b) avoids repeated G-buffer reads from the main memory for adaptive deferred shading. In fact, this swizzle pattern alone can be directly used with DACS as well to minimize its G-buffer bandwidth. This swizzle pattern ensures that the G-buffer is read at most once and it can even reduce the G-buffer bandwidth further. The part of the G-buffer that corresponds to the first level (1/16 of the G-buffer size) is always read. As for the following levels, for every group of pixels within a cacheline, if the result of the user criterion is such that they are not shaded, the cacheline is not read at all. For example, if fragment shading is not computed for the last (5^{th}) level, half of the G-buffer bandwidth can be eliminated. Similar savings can happen at any shading level (except for the first).

Our hardware scatter tiles are designed to avoid introducing extra framebuffer read and write operations for implementing adaptive deferred shading. By caching the output of the fragment shading on-chip, we avoid introducing additional bandwidth for the framebuffer. Also, each cacheline of the framebuffer is written only once. Thus, similarly to VRS and brute-force shading, we write the framebuffer output exactly once.

5.2 Quality

In Figure 5, we compare adaptive deferred shading and variable-rate shading on a synthetic example involving gradients and a step function. In the gradient regions, VRS is unable to interpolate, and so its undersampling leads to blocky artifacts. In the diagonal region, the discontinuity must be densely sampled. VRS is only able to achieve this by increasing the sampling rate for an entire tile. This leads to shading effort being expended in the off-diagonal regions and, given the constrained shading rate, this still is not sufficient to resolve the diagonal adequately. The result is a blocky image with quantitively inferior quality to that produced by adaptive deferred shading.

One important practical concern with VRS is that the user must specify the shading rate of each block (i.e. 16×16 pixel region on the tested hardware) prior to rendering. In practice, this is done using various perceptual heuristics, which can produce acceptable results [Yang and Zhdan 2019]. In our tests, however, we precomputed the *optimal* block shading levels (in the sense of achieving epsilon-factor globally-minimal per-pixel RMSE errors) using an offline algorithm based on the multiple-choice knapsack problem (MCKP). Thus all VRS results we present are the "best-possible" quality for a given shading rate.

Because shading effort is allocated at the granularity of tiles, in order to shade high-frequency features at the required high shading rate, VRS ends up shading any surrounding low-frequency features at a high rate as well. Furthermore, since VRS cannot interpolate, even low-frequency

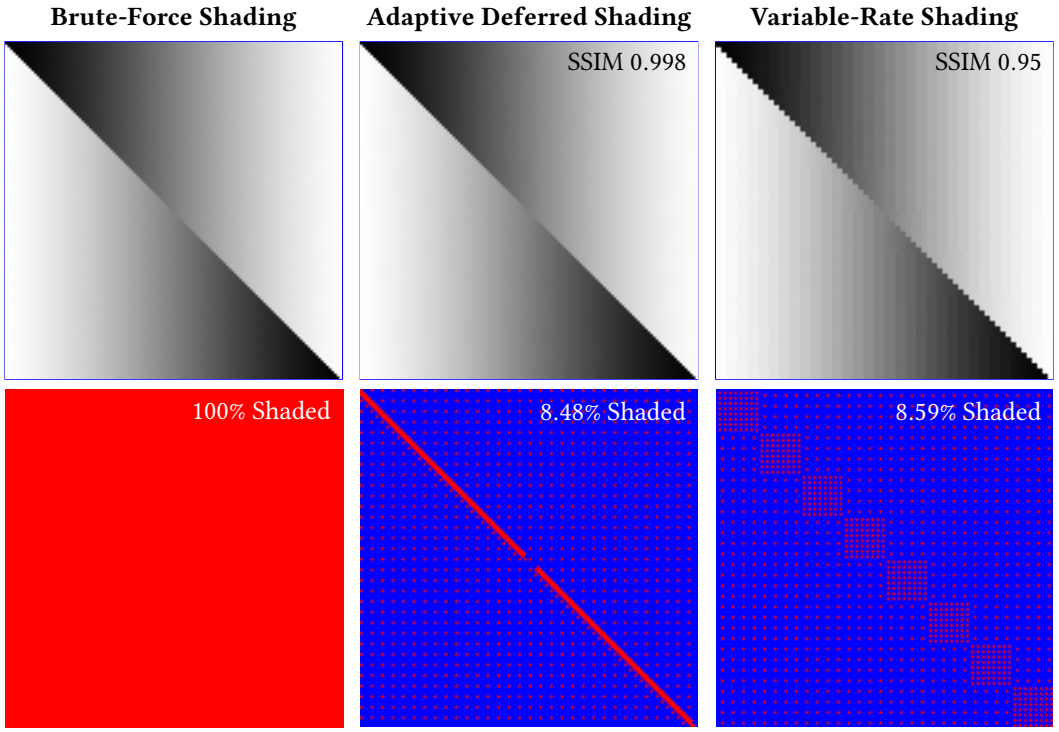


Fig. 5. *Synthetic example rendered with both adaptive deferred shading and variable-rate shading. Despite shading at the same rate, VRS undersamples the diagonal edge and renders the gradient with blocky artifacts, leading to an overall worse image.*

regions must be shaded at a high rate if they vary from a flat color, which is common. These two effects mean that VRS's overall shading rate must be higher to capture the same detail; equivalently, if the shading rates are constrained to be the same, the visual quality of adaptive deferred shading is superior.

In Figure 6, we show a rendering of the Crytek Sponza and Lumberyard Bistro scenes. The insets show how adaptive deferred shading and VRS, with both algorithms constrained to the same shading rate, handle various regions. Adaptive deferred shading can be seen to capture fine detail and silhouette edges, because it is largely shading *only* those perceptually important features. Whereas, VRS is not really designed for such low shading rates. VRS, with appropriate coding, can detect these features, but because it overshades in tiles, simply does not have enough shading budget to expend to shade them properly. As a consequence, the shading rate for these features is lower, leading to perceptual loss: block artifacts can clearly be seen in shadows and silhouette edges. Note that, although VRS can be used with deferred shading, it is primarily designed for forward shading. It can properly handle geometric discontinuities in forward shading, but not during deferred shading, resulting in noticeable artifacts.

5.3 Performance

We present our performance analysis in this section. All results are obtained on an NVIDIA RTX 2080 MaxQ GPU using a custom OpenGL-based renderer. Our test application computes *percentage closer soft shadows* [Fernando 2005] sampling an $8K \times 8K$ shadow map.

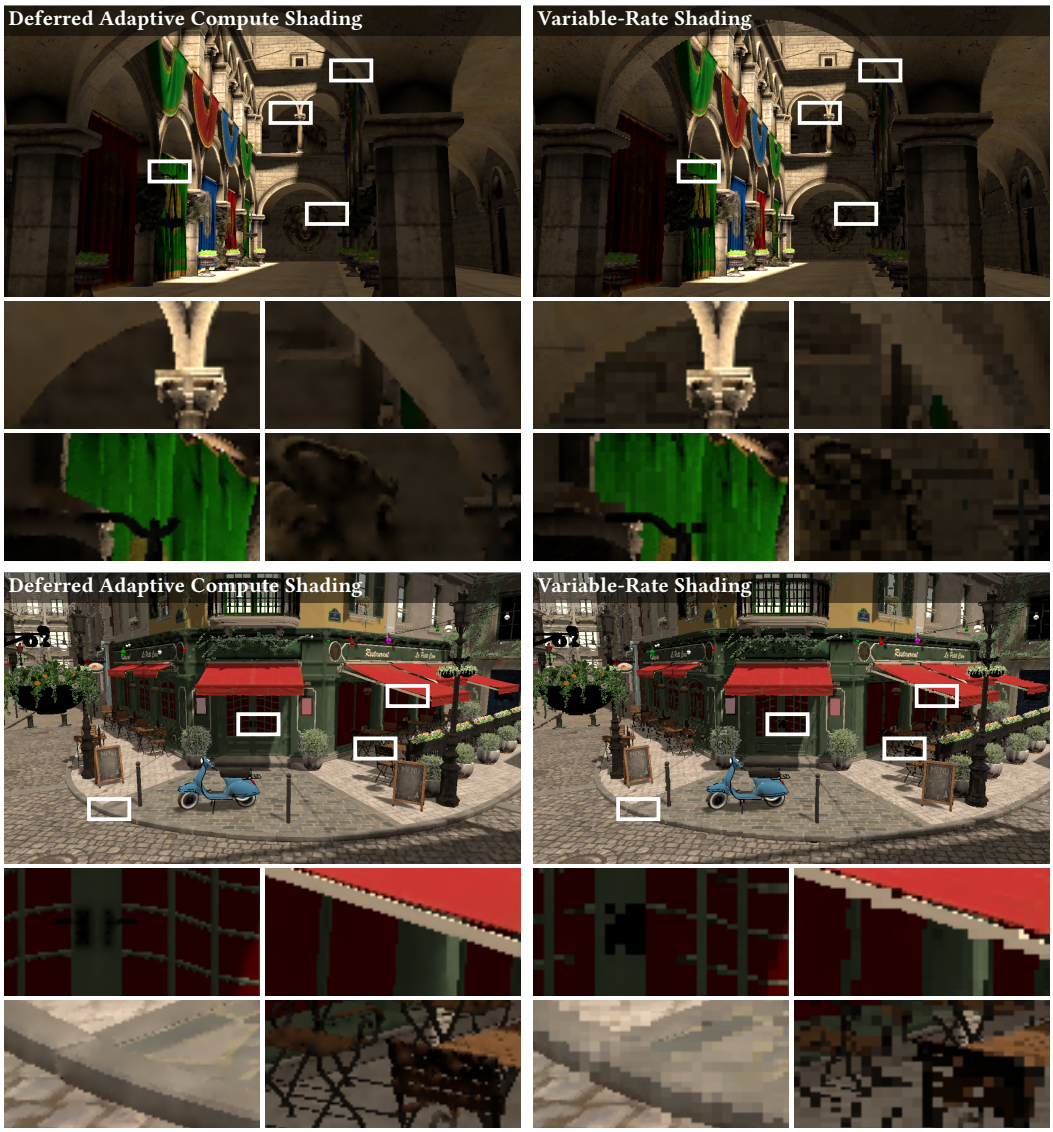


Fig. 6. *We compare adaptive deferred shading and deferred VRS in the Crytek Sponza and Lumberyard Bistro scenes at 30% and 50% shading rates, respectively. VRS is too coarse to allocate its shader invocations to the fine details in the scene (insets), resulting in blocky artifacts. VRS could partially address these artifacts near silhouettes by using forward mode instead of deferred, but then there's overdraw.*

We use DACS for providing a performance analysis of our G-buffer swizzling. We run DACS first by using a regular G-buffer texture, then by using a texture swizzled using our method. Swizzling and unswizzling is performed in software using additional passes. The computation time of software swizzling passes are not included in the results. However, DACS accessing the swizzled G-buffer has the additional cost of computing the swizzled texture coordinates, which is included.

Table 2. *Shading latency of DACS with and without our swizzling.*

	Fragment Shading Rate	@ 720p		@ 2160p	
		Crytek Sponza	Lumberyard Bistro	Crytek Sponza	Lumberyard Bistro
Regular G-buffer	100%	77.27 ms	33.77 ms	644.96 ms	281.22 ms
Our Swizzled G-buffer		76.36 ms	32.93 ms	636.25 ms	274.03 ms
Difference		**0.91 ms**	**0.84 ms**	**8.71 ms**	**7.19 ms**
Regular G-buffer	70%	63.84 ms	26.87 ms	479.14 ms	193.48 ms
Our Swizzled G-buffer		62.72 ms	26.03 ms	471.20 ms	189.51 ms
Difference		**1.12 ms**	**0.84 ms**	**7.94 ms**	**3.97 ms**
Regular G-buffer	30%	31.35 ms	15.10 ms	221.73 ms	95.06 ms
Our Swizzled G-buffer		30.55 ms	14.43 ms	214.90 ms	89.19 ms
Difference		**0.80 ms**	**0.67 ms**	**6.82 ms**	**5.87 ms**

The render times with and without a swizzled G-buffer using a variety of shading rates in two different scenes are shown in Table 2. Notice that, in these tests, using our G-buffer swizzling the render times are consistently reduced by about 0.9 millisecond at 720p resolution and up to almost 9 milliseconds at 2160p resolution, as compared to a regular G-buffer texture. The performance improvement of our G-buffer swizzling does not increase with reduced shading rate, because our user criterion involves reading the G-buffer, so the G-buffer is unconditionally accessed for all pixels.

Unfortunately, evaluating the expected performance of our hardware scatter tiles is more challenging, requiring a full GPU simulation. Moreover, the work scheduling policies for the GPU warps can be a major factor determining the performance, particularly when the shader is memory-bound, such as when a shader requires a substantial amount of texture accesses.

We demonstrate this complexity using different computation methods that run the same shading code for all pixels, without any reduction in shading rate. The results are shown in Table 3.

Table 3. *Performance of different scheduling algorithms for fragment shading at 100% shading rate.*

	@ 720p		@ 2160p	
	Crytek Sponza	Lumberyard Bistro	Crytek Sponza	Lumberyard Bistro
Forward Shading	188.44 ms	124.74 ms	935.35 ms	629.17 ms
Forward Shading (VRS)	189.50 ms	138.83 ms	940.74 ms	738.30 ms
Deferred Fragment Shader	33.36 ms	12.62 ms	278.60 ms	100.87 ms
Deferred Fragment Shader (VRS)	33.81 ms	12.53 ms	277.36 ms	101.75 ms
Deferred Compute Shader	68.32 ms	30.53 ms	585.20 ms	267.16 ms
Deferred Compute Shader (Tiles)	21.96 ms	15.13 ms	143.91 ms	140.60 ms

The first method we test is forward shading with and without VRS. In these tests, VRS is set to shade one sample per pixel; therefore, it provides no reduction in shading, but incurs a relatively small proportional overhead. As expected, when the fragment shader is expensive, which is typical in most modern graphics applications, forward shading delivers poor performance, due to overdraw. Deferred shading using a fragment shader for the shading pass with and without VRS can significantly improve the performance, as compared to forward shading. Unlike in the forward-shading case, VRS with one sample per pixel leads to negligible overhead. Obviously, reducing the shading rate with VRS improves the performance at the cost of some reduction in

quality. This quality reduction can be mostly hidden, if the shading rate is chosen carefully and/or postprocessing tricks are used.

Interestingly, running the very same shading code inside a compute shader can lead to significantly different performance characteristics. For testing the performance of compute shaders for handling fragment shading, we implemented a brute-force compute shader that evaluates every pixel. This compute shader implementation splits the framebuffer area into work-groups and simply shades each pixel. Choosing the size of the region is nontrivial. A work-group size of 8×4, for example, matches the warp size of the GPU hardware. However, more or larger work-groups are a better choice, as they allow the GPU to hide latency encountered. However, in our tests, although we tried many configurations, the end result does not matter much in practice: with the best configuration, the deferred compute shader more than doubled the render time in the Crytek Sponza scene and almost tripled it in the Lumberyard scene. Since we use the same shading code and the same hardware, the primary difference is the order in which the work is scheduled internally by the GPU. Indeed, analyzing the code with NVIDIA's NSight software revealed that the bottleneck was memory incoherency during shading.

However, this apparent inefficiency is not an *inherent* problem of using compute shaders. Our alternative brute-force compute shader implementation uses tiles of 32×32 pixels. Instead of generating as many work-groups as the number of tiles (like the compute shader implementation discussed above), this compute shader uses work-groups of size 32×1 and spawns as many of them as the number of shading multiprocessors available on the GPU, times some small latency-hiding factor. During execution, each work-group obtains the indices of the pixels to shade from a global index, atomically incrementing it. The pixels are shaded within a loop, following scanline order within a 32×32 tile, with the tiles themselves arranged in scanline order, until all pixels are shaded. Surprisingly, this unusual software-managed scheduler provides significantly better performance than the other compute shader implementation above. In fact, it even performs significantly faster *even than deferred shading using an fragment shader* in the Crytek Sponza scene. In the Lumberyard Bistro scene, it trails behind the fragment shader implementation at 720p and performs slower than the other computer shader implementation at 2160p.

The performance of DACS running within a compute shader at 100% shading rate is included in Table 2. As expected, DACS incurs an additional overhead on top of our deferred shading implementation with a compute shader (Table 3).

These observations lead to two salient points. First, great caution must be exercised when discussing or analyzing performance numbers for fragment shading methods that use software scheduling. Second, this unpredictability of performance is a further argument that hardware scheduling support is essential: a hardware implementation maintained by the vendor could be more stable and future-proof.

6 EXTENSIONS

Achieving a high-performance implementation of adaptive deferred shading, as we explain in this paper, requires changes to the GPU hardware for supporting the new G-buffer swizzling order and the new warp scheduler based on scatter tiles. We also need some extensions for the graphics API.

First of all, the new swizzling order should be exposed in the API, allowing textures to be stored with this order. Obviously, this is particularly useful for textures that will be used as a part of the G-buffer.

Also, we need a new rendering mode that would invoke warp scheduling with scatter tiles. Since this is targeted for deferred shading, the part of the pipeline prior to the fragment shader, including the rasterizer, can be completely skipped. In that respect, this render mode would appear more like compute shader evaluations, but using a different warp scheduler.

6.1 Fragment Pre-Shader

It is important to permit software control for determining which pixels should evaluate the fragment shader. This is because the user-defined criterion can be completely different for different applications. In fact, different applications may favor different interpolation methods for the four neighbor pixel values from the previous shading level, used when the criterion decides to skip fragment shading for the pixel.

Therefore, we propose the *fragment pre-shader* stage, a specialized, light-weight fragment shader that is responsible for determining whether the fragment shader should be evaluated and perform the interpolation of the four neighbor pixel values. This fragment pre-shader would be identical to a fragment shader with a few minor differences. First, it would be required to output a binary value indicating whether the actual fragment shader should be evaluated. Second, it will have access to the output of the four neighbor pixels that were previously computed, as a part of the computations for the previous shading levels.

While we envision the fragment pre-shader to be relatively light-weight, it can still perform expensive operations like accessing the G-buffer. It can also be used for interpolating a part of the fragment-shading computations, while deciding to call the fragment shader for performing the remaining computation. To facilitate this, the fragment shader can output more data than simply the colors that will be written to the framebuffer, which would allow sending arbitrary computation data to the pixels of the following levels.

This fragment pre-shader is not needed for the first level, which is unconditionally shaded. Normally, the pre-shader is needed only when all neighbor data is available.

On the other hand, in some cases, the pre-shader may compute values that the fragment shader would need to re-compute if the pre-shader is not executed. This can be solved by executing the pre-shader for *all* pixels, even level 1, with an input indicating which neighbor data is available (if any). This is also a good way to handle border pixels, where levels 2-4 may not have all neighbor data available.

6.2 Framerate Stabilization

An important missing feature of the GPU rendering pipeline is controlling the render time (i.e. the framerate) on-the-fly. Once a render command is issued, it runs to completion, even if parts of the rendering task, such as fragment shading, take longer than expected. This results in variations in framerate that can lead to discomfort and negatively impact the quality of the interactive experience.

The adaptive deferred shading approach we describe could be easily extended to provide a form of framerate stabilization. For example, the user-defined criterion can be adjusted depending upon the time it took to render the previous levels. This can be handled within the fragment pre-shader by providing a progress indicator. This way, the fragment pre-shader can adjust its criterion. Alternatively, if the deferred shading time exceeds a certain user-defined limit, the GPU can simply ignore the decision of the fragment pre-shader and simply skip fragment shading (and only run the pre-shader) for the remaining pixels.

Obviously, regardless of how framerate stabilization is implemented, reducing the shading rate negatively impacts the quality of the rendered image. Nonetheless, some reduction in momentary render quality might be preferable to framerate fluctuations. Indeed, such reductions in image quality are commonplace with video streaming applications.

6.3 Temporal Filtering with Adaptive Deferred Shading

With additional hardware support, adaptive deferred shading can be extended to implement a form of checkerboard rendering with pixel-level granularity. Note that the last level of the subdivision

order includes half of the pixels in the framebuffer. Therefore, skipping the fragment shader evaluation for this last level alone would have a significant performance impact. Unconditionally skipping the last level can be considered a form of checkerboard rendering.

Checkerboard rendering results in much better image quality when it alternates between which pixels are shaded and which ones are reconstructed, with the reconstruction being computed by temporal reconstruction filtering using the previous frame. For achieving a similar improvement with checkerboard rendering using adaptive deferred shading, the scatter tiles may be simply flipped. For example, by simply mirroring the scatter tiles (flipping horizontally or vertically), we can alternate which pixels appear in the last shading level.

A similar checkerboard rendering approach could also be implemented without direct hardware support. By simply shifting the clip-space coordinates by one pixel while generating the G-buffer, we can alternate the pixel content that appears in the last shading level. This way, even if the pixels of the last level remain the same, we can get the benefits of temporal filtering with checkerboard-style rendering.

7 CONCLUSION

We have presented changes to the GPU rendering pipeline for supporting an efficient implementation of adaptive deferred shading. In particular, we have described a new swizzling order that optimizes the G-buffer accesses and a new warp scheduling approach to minimize the data movement for the fragment shading output. We have also presented test results comparing adaptive deferred shading to variable-rate shading to indicate the potential performance and quality improvements that could be achieved by providing hardware support for adaptive deferred shading. Finally, we have discussed potential extensions to the graphics API for supporting adaptive deferred shading with a fragment pre-shader and additional extensions that can achieve further reduction in fragment shading operations and stable framerates.

ACKNOWLEDGMENTS

This project was supported in part by a grant from Facebook Reality Labs.

REFERENCES

Tomas Akenine-Möller, Jacob Munkberg, and Jon Hasselgren. 2007. Stochastic Rasterization Using Time-continuous Triangles. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '07)*. 7–16.

John Burgess. 2020. RTX on—The NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44.

Petrik Clarberg, Robert Toth, Jon Hasselgren, Jim Nilsson, and Tomas Akenine-Möller. 2014. AMFS: adaptive multi-frequency shading for future graphics processors. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 141.

Petrik Clarberg, Robert Toth, and Jacob Munkberg. 2013. A Sort-based Deferred Shading Architecture for Decoupled Sampling. *ACM Trans. Graph.* 32, 4, Article 141 (July 2013), 10 pages.

Cyril Crassin, Morgan McGuire, Kayvon Fatahalian, and Aaron Lefohn. 2015. Aggregate G-buffer Anti-aliasing. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games (i3D '15)*. 109–119.

Randima Fernando. 2005. Percentage-closer Soft Shadows. In *ACM SIGGRAPH 2005 Sketches (SIGGRAPH '05)*. ACM, New York, NY, USA, Article 35.

Yong He, Yan Gu, and Kayvon Fatahalian. 2014. Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Transactions on Graphics* 33, 4 (2014), Article–142.

Ethan Kerzner and Marco Salvi. 2014. Streaming G-Buffer Compression for Multi-Sample Anti-Aliasing. In *Proceedings of High Performance Graphics*. Eurographics Association, 1–7.

Gábor Liktor and Carsten Dachsbacher. 2012. Decoupled Deferred Shading for Hardware Rasterization. In *Proc. of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 143–150.

Agatha Mallett and Cem Yuksel. 2018. Deferred Adaptive Compute Shading. In *Proceedings of the Conference on High-Performance Graphics (HPG '18)*. ACM, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3231578.3232160

Jalal Eddine El Mansouri. 2016. Rendering 'Rainbow Six | Siege'. Game Developers Conference (GDC).

Morgan McGuire, Eric Enderton, Peter Shirley, and David Luebke. 2010. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *Proceedings of High Performance Graphics 2010*. Eurographics Association, 173–182.

Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. 2011. Decoupled Sampling for Graphics Pipelines. *ACM Trans. Graph.* 30, 3, Article 17 (May 2011), 17 pages.

Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. *SIGGRAPH Comput. Graph.* 24, 4 (Sept. 1990), 197–206.

Michael Stengel, Steve Grogorick, Martin Eisemann, and Marcus Magnor. 2016. Adaptive Image-Space Sampling for Gaze-Contingent Real-time Rendering. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 129–139.

Karthik Vaidyanathan, Marco Salvi, Robert Toth, Tim Foley, Tomas Akenine-Möller, Jim Nilsson, Jacob Munkberg, Jon Hasselgren, Masamichi Sugihara, Petrik Clarberg, Tomasz Janczak, and Aaron Lefohn. 2014. Coarse Pixel Shading. In *Proceedings of High Performance Graphics*. Eurographics Association, 9–18.

Alex Vlachos. 2016. Advanced VR Rendering Performance. Game Developers Conference (GDC).

Graham Wihlidal. 2017. 4K Checkerboard in Battlefield 1 and Mass Effect Andromeda. Game Developers Conference (GDC).

Lei Yang and Dmitry Zhdan. 2019. NVIDIA Adaptive Shading Overview. (2019). http://www.leiy.cc/publications/nas/nas-gdc19.pdf Game Developers Conference.