High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Coarse Pixel Shading

K. Vaidyanathan[1], M. Salvi[1], R. Toth[1], T. Foley[1], T. Akenine-Möller[1,2], J. Nilsson[1],

J. Munkberg[1], J. Hasselgren[1], M. Sugihara[1], P. Clarberg[1], T. Janczak[1], A. Lefohn

[1]Intel Corporation, [2]Lund University

Figure 1: *The* CITADEL 1 *scene, rendered at 2560×1440 with pixel-rate shading (PS) on the left and coarse pixel shading (CPS) on the right, using a coarse pixel size of 2×2. CPS almost halves the number of shader invocations, yet shows few perceivable differences on a high pixel density display, with a structural similarity index (SSIM) of 90.6%. In contrast, an image rendered at 1280×720 and upscaled exhibits blurring at silhouette edges, and lower overall quality, with an SSIM of 86.9%.*

Abstract

We present a novel architecture for flexible control of shading rates in a GPU pipeline, and demonstrate substantially reduced shading costs for various applications. We decouple shading and visibility by restricting and quantizing shading rates to a finite set of screen-aligned grids, leading to simpler and fewer changes to the GPU pipeline compared to alternative approaches. Our architecture introduces different mechanisms for programmable control of the shading rate, which enables efficient shading in several scenarios, e.g., rendering for high pixel density displays, foveated rendering, and adaptive shading for motion and defocus blur. We also support shading at multiple rates in a single pass, which allows the user to compute different shading terms at rates better matching their frequency content.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

DOI: 10.2312/hpg.20141089

1. Introduction

A significant portion of the power cost in a 3D pipeline is due to pixel shading [Poo12] and current trends towards higher display resolutions and graphics quality, especially in handheld devices, are at odds with the need to minimize power dissipation. Running pixel shaders more efficiently and without compromising image quality would therefore be particularly advantageous.

Approaches for *decoupled shading* take advantage of the fact that geometric occlusion typically introduces higher-frequency details compared to surface shading, and perform shading at a lower rate than visibility testing. Figure 1 shows that decoupled shading is well suited for high pixel density displays, where the effects of reduced shading rates are barely discernible from normal viewing distances. Shading rates may be further reduced in regions of the screen that are blurred or otherwise less perceivable to the user, for instance regions affected by motion or defocus blur, or regions outside the user's foveal vision.

It is difficult to take advantage of these opportunities on current graphics architectures, where shaders are restricted to execute at per-pixel or per-sample rate. Shading rates may be coarsely controlled by varying frame buffer resolution, but this does not allow more fine grained variation of shading rates, e.g., per object, per triangle, or per image region, which are still coupled to the visibility sampling rate.

In this paper, we introduce *coarse pixel shading* (CPS), which is an architecture for varying shading rates in a rasterization pipeline, while keeping the visibility sampling rate constant. Our work differs substantially from other decoupled shading schemes, such as the ones proposed by Ragan-Kelly et al. [RKLC*11] and Burns et al. [BFM10]. We greatly simplify and constrain our approach so that it is amenable to efficient implementation in current rasterization pipelines. Our approach can be seen as a generalization of multi-sample anti-aliasing (MSAA) [Ake93]. Where MSAA uses a fixed number of visibility samples, and one shading sample per pixel, CPS allows the number of shading samples to be varied, and further reduced. To support CPS, we also introduce *multi-rate shading*, which allows a single rendering pass to execute shading code at one or more different rates: per group of pixels, per pixel, and per sample.

2. Background and Related Work

Many approaches have been proposed to improve shading efficiency by sampling shading at a lower rate than visibility. *Multi-sample anti-aliasing* (MSAA) [Ake93] is one such technique, often supported by GPU hardware. With MSAA, multiple *coverage samples* (also called visibility samples) are stored per pixel, but pixel shaders are only executed once for each pixel covered by a primitive. This is in contrast to *super-sampling*, in which shaders are executed once per covered sample.

In a stochastic renderer, MSAA cannot be employed efficiently [MCH*11], so other techniques have been developed. Ragan-Kelley et al. [RKLC*11] propose decoupled sampling (DS), where shading and visibility are sampled in a decoupled manner. Pixel shading is evaluated lazily over an image-space shading grid and temporarily stored in a memoization cache for reuse during *stochastic rasterization* (SR) [CCC87, AMMH07]. Burns et al. [BFM10] use another decoupled sampling approach, with shading sampled uniformly in parametric patch space in an optimized Reyes architecture [CCC87]. These techniques allow shading grids with arbitrary grid spacing, which enables more flexible control of the shading rate.

Liktor and Dachsbacher [LD12] implement deferred shading for SR with a software based memoization cache on a GPU. Andersson et al. [AHTAM14] present another GPU implementation of DS for SR, where shading is computed in object space and stored in a sparse mipmap hierarchy [GBAM11] using conservative rasterization. In a second pass, the scene is stochastically rasterized and shading results are looked up in the mipmap. Clarberg et al. [CTM13] propose a hardware architecture to reduce shading rates for SR in the context of tiled deferred shading, where coherence is extracted by sorting shading point identifiers.

While our focus is on shading for rasterization, similar approaches have also been applied to ray tracing. For example, Stoll et al.'s Razor architecture [SMD*06] shades grids in object-space and caches them for later reuse. They also propose computing some shading terms at different rates, i.e., view-independent computations are performed per vertex, and view-dependent ones per sample.

Shading rates may also be varied non-uniformly, so detail can be reduced in areas where it is unlikely to be noticed. Ragan-Kelley et al. [RKLC*11] propose to scale the spacing of the shading grid based on the absolute value of the circle of confusion, reducing the shading rate in out-of-focus regions. Vaidyanathan et al. [VTS*12] use an analysis of the frequencies involved in light transport [DHS*05], along with a set of tailored approximations, to derive suitable grid spacings for motion and defocus blur. Hasselgren and Akenine-Möller [HAM07] present the programmable culling unit (PCU), which uses an interval-based culling program per tile to avoid work for a tile of fragments. In addition, fragment shader switching is used in order to take a high-level decision per tile and select a fragment shader based on the tile shader output.

Similarly, *foveated rendering* [GFD*12] exploits the fact that a human observer only sees fine detail in a 5° circle around their gaze direction to reduce shading costs without perceivable error. Eye tracking is used to detect the viewer's gaze, and several layers are rendered with decreasing resolution around this point using multiple passes. These layers are then composited to produce the final image.

3. Algorithm

To allow shading at a lower rate than once per pixel, we introduce the notion of a *coarse pixel* (CP). A CP is a group of $N_x \times N_y$ pixels, which will share the result of a single *coarse pixel shader* (CPS) evaluation. This is similar to how multiple visibility samples share a single pixel shader evaluation with MSAA, with the difference that we can vary the shading rate by varying the size of the CP. Similar to current GPUs, we shade in groups of 2×2 CPs, called *coarse quads* (CQs), to facilitate computing derivatives using finite differences.

As we will show in Section 6, many applications can benefit from the ability to vary shading rates across different regions of the screen. To enable such variation in the shading rate, we divide the screen into tiles of size $T_x \times T_y$ and allow a different value of the CP size for each tile. Each tile maps to a shading grid of CQs, with the selected CP size.

The key to our simplified decoupled sampling technique is avoiding overlapping shading grids, which ensures that each pixel unambiguously belongs to only one CQ. This can be achieved by restricting the CP sizes to a finite set of values that ensure that the shading grid is perfectly aligned with the tile boundaries. With this assumption, our shading technique can be summarized as follows:

Algorithm 1 Simplified decoupled sampling with CPS.

```
for each primitive:
    for each covered tile on screen:
        Rasterize tile and store visible fragments
        Determine Nx, Ny for tile
        Divide tile into CQs of size 2Nx × 2Ny
        for each CQ with visible fragments:
            Shade CQ
            Write output color to all covered pixels
```

In order to ensure a perfectly aligned grid of CQs inside a tile, we restrict the CP sizes such that the tile size is a common multiple (e.g., the LCM) of all allowed CQ sizes.

3.1. Controlling Coarse Pixel Size

We have identified a small number of modes for controlling the CP size that are easy to use, yet powerful enough to support a range of applications, as shown in Section 6. In each case, the user does not directly specify a CP size, but rather a pair of *CP parameters* (s_x, s_y) that specifies the desired CP size. The CP parameters are then quantized to the closest available CP size that meets or exceeds the requested shading rate. Based on the selected mode, the CP parameters (s_x, s_y) may be:

- interpolated from per-vertex shader outputs,
- set to a constant value using render state, or
- expressed as a radial function of screen coordinates.

Controlling the CP parameters with a shader output is highly flexible, and allows many use cases to be expressed.

No Compensation Compensated LOD

Figure 2: *A zoomed-in image, rendered with smoothly varying CP parameters having a smaller value towards the top right and a larger value towards the bottom left. Quantizing the CP parameters can produce visible discontinuities in the CP size (top), which can be masked by compensating the LOD calculation in the texture sampler (bottom).*

The ability to set constant CP parameters is provided for simplicity. It is the least invasive method for adding CPS to an existing application. The ability to use a radial function is included specifically for foveated rendering [GFD*12]. As a radial function cannot be robustly expressed using linear interpolation of per-vertex attributes, we included a separate mode for this special case.

In order to always meet the required shading rate, the CP size, $N_x \times N_y$, is determined by computing conservative lower bounds for $|s_x|$ and $|s_y|$ within the tile, and rounding down to the nearest available CP size. The use of absolute values here enables use of negative CP parameters, which we take advantage of in the case of motion and defocus blur as discussed in Section 6.4.

3.2. Texture LOD Compensation

There are two sources of quantization of CP sizes. First, the CP size is evaluated only once per tile. Second, the CP size is quantized to one of the finite available CP sizes. Both of these sources cause discontinuities in the CP sizes as we move from tile to tile, which may result in visible tile transitions as seen in Figure 2. To compensate for these discontinuities, we augment the texture sampler level of detail (LOD) calculation to reflect the requested (un-quantized) CP size. This can be done by scaling the finite differences of texture coordinates that are used to compute the texture LOD:

$$\delta_x^{new} = \delta_x \frac{|s_x|}{N_x}, \quad \delta_y^{new} = \delta_y \frac{|s_y|}{N_y},$$

where δ_x and δ_y are the finite differences of the texture coordinates along the x- and y-axes respectively. The values of (s_x, s_y) are evaluated for every CP, unlike the CP size which is evaluated once per tile. Compensating the texture LOD creates a smooth variation in image detail, which masks the discontinuities in the CP size, as shown in Figure 2.

Although LOD compensation can be effective in many cases, there are some scenarios where it might not be applicable, for example, with procedurally generated textures. In such cases, LOD compensation techniques can possibly

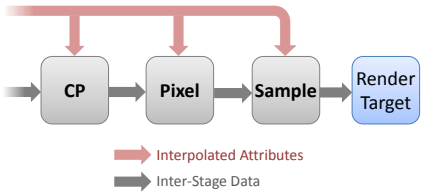

Figure 3: *Pipeline abstraction of the post rasterization stages showing the different phases of shading and the dataflow between the phases.*

be applied in user space, based on the CP size and CP parameters, which are available as shader inputs.

3.3. Multi-Rate Shading

One of the key features of our proposed architecture is the ability to execute shading at up to three different rates within the same rendering pass. Some of the shader computations can, e.g., be moved to a lower rate than once per pixel, while certain high-frequency effects can be evaluated per pixel, or even per visibility sample to reduce aliasing. Note that this is not possible in current graphics APIs, as the pixel shader can be configured to run at pixel *or* sample rate, but the two are mutually exclusive. Conceptually, we divide the single pixel shader stage of the graphics pipeline into three distinct *phases*, one for each rate (see Figure 3). Hence, after a tile is divided into coarse quads, each quad is shaded at one or more different rates: per-CP, per-pixel, and per-sample.

4. Pipeline Description

Figure 4 illustrates how CPS can be integrated into an existing graphics pipeline. With our simplified decoupled sampling technique, we can buffer rasterized samples for a given tile and primitive in a *tile buffer*, and then invoke the pixel shader on complete CQs. This solution closely resembles the buffering performed for quad-based shading in current architectures. The tile buffer retains the screen coordinates and all the necessary information to resolve visibility, such as depth values and coverage information. When a new tile or a new primitive is rasterized, the tile buffer is flushed, i.e., only a single tile's worth of data needs to be buffered.

Note that a memoization cache framework [RKLC*11] could also be used to implement CPS, where samples produced by the rasterizer could trigger shading of CQs. However, such an approach is likely to require significant changes to current hardware architectures.

5. User Abstraction

Figure 3 illustrates how CPS and multi-rate shading are integrated into the pipeline abstraction of an existing graphics architecture like Direct3D 11 [Mic12]. Each shading phase has access to interpolated per-vertex attributes and a small

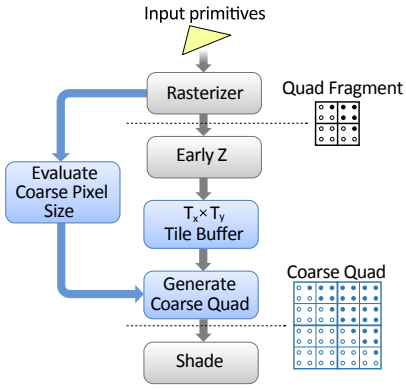

Figure 4: *A portion of the GPU pipeline with modifications for coarse pixel shading (in blue). The rasterizer tests input primitives to generate quad fragments, which are buffered for every $T_x \times T_y$ tile on the screen. The buffered fragments are mapped to coarse pixels and coarse quads, which are shaded and then resolved into the output buffers.*

```
struct VS_OUT { // Output attributes
    float2 cpsize : SV_CoarsePixelSize;
    ...
};
VS_OUT VertexShader(VS_IN In) {
    VS_OUT Out;
    Out.cpsize = ComputeCPSize(In)
    ...
    return Out;
}

[shadingphase("coarse-pixel")]
[nextshaderfunc("PixelShader")]
float4 CoarseShader(VS_OUT In) {
    return ComputeDiffuse(In)
}

[shadingphase("pixel")]
float4 PixelShader(VS_OUT VSIn, float4 CPIn) {
    return ComputeSpecular(VSIn, CPIn);
}
```

Figure 5: *An example of multi-rate shading, where a low frequency diffuse term is computed per coarse pixel and a high frequency specular term is computed per pixel. The CP size is controlled through the vertex shader.*

amount of data may be communicated from one phase to the next through user-defined shader outputs; the amount of data allowed is an implementation-specific limit. Any of the phases may also write outputs that will be consumed in later pipeline stages (e.g., framebuffer blending), and we will discuss constraints on this below.

We considered many alternatives for exposing multi-rate shading. Perhaps the simplest would be to expose each of the phases of shading as a distinct pipeline stage, to which shaders and resources can be bound through the graphics API. We dismissed this option since a driver might be forced to compile code behind the scenes to maintain the illusion of logically distinct pipeline stages on architectures that would execute the shading phases in the same shader thread. Another family of options would allow a single shader function to contain code that runs at different rates, either sepa-

rated by syntactic *phase markers*, or tagged with *rate qualifiers* [PMTH01,FH11]. Such approaches make the plumbing of data between rates implicit, and could obscure the amount of inter-phase data being used. We prefer to make such resource usage, which could impact performance, more immediately visible to shader writers.

The solution we ultimately arrived at is a single conceptual pipeline stage, running a single shader compiled from up to three different entry points, one for each rate as shown in Figure 5. This approach provides programmers with a high degree of visibility into, and control over, what code runs at each rate, as well as what data flows between phases. The simpler models described above may still be implemented as layered abstractions in cases where control can be sacrificed for ease of use.

5.1. System-Defined Shader Parameters

When CP size is being controlled by shader code (see Section 3.1), an additional system-interpreted value, `SV_CoarsePixelSize`, is made available to the last shading stage before the rasterizer. This output is a two-component vector, and corresponds to the CP parameters (s_x, s_y) defined in Section 3.1. Typically this output will be set in a vertex shader, but could also be defined in a hull, domain, or geometry shader, if these are used.

In addition, shader code running at CP rate has access to inputs for both the interpolated values of the CP parameters before quantization, and the actual coarse pixel size: `SV_RequestedCoarsePixelSize` and `SV_CoarsePixelSize`, respectively. These two values together are sufficient for a shader to compute its own LOD compensation (see Section 3.2), for use in computations that do not involve the texture sampler (e.g., prefiltering a procedural texture).

5.2. Shader Outputs

We initially expected that users would often want to compute shader outputs at different rates, e.g., evaluating specular albedo at a lower rate than diffuse when outputting a G-buffer. However, allowing outputs to be written from any phase would lead to thorny semantic questions in the presence of operations like fragment discard. For example, a shader could write one output at coarse rate, and then perform a conditional `discard` at pixel rate. A naïve interpretation might say that the write in the coarse-rate code "already happened" and values should be written (and blended) into the framebuffer before the `discard`. However, this contradicts the mental model of the 3D pipeline in OpenGL and Direct3D, where all pixel shading logically occurs *before* framebuffer blending.

To avoid such ill-defined semantics, we opt for a simple restriction: a shader with multiple phases may only write framebuffer outputs from the last phase. Restricting outputs to the last phase could lead to worse performance, but in practice we expect a compiler to be able to *hoist* writes from a higher to a lower rate when it can prove that it is safe (e.g., when there are no `discard` operations). The same kind of hoisting optimization can apply to shaders that conditionally compute their output at per-pixel or per-sample rate.

6. Applications

In this section, we will discuss several important use cases, where CPS can be used to significantly reduce the amount of pixel shading work. This is by no means an exhaustive list, as we expect many more use cases to be found and explored.

6.1. High-DPI Rendering

Rendering to the native resolution of high-DPI displays is often a task too demanding for the GPU. The typical remedy is to lower the rendered resolution and upsample the image, which results in perceivable quality degradation along object silhouettes, while changes in surface interiors are not as apparent. Instead, by using CPS and setting the CP parameters to a constant value, such as 2×2 pixels, we can achieve a dramatic reduction of shading rate while retaining most detail. An example is shown in Figure 1, with hardly any perceivable change in image quality at normal viewing distances.

With more flexible control over the shading rate, we can enable a wider range of applications, as discussed below.

6.2. Shading Low-Detail Surfaces

Some materials have lots of surface detail, while others do not. By choosing a CP parameter depending on the material type, computations can be saved where the visual impact is minimal. For instance, a particle system for rendering smoke may be rather homogeneous and shaded at a low rate, while a sign with text may warrant high resolution shading. Similarly, objects in full shadow may possibly be shaded at a lower rate than objects in bright sunlight.

6.3. Foveated Rendering

CPS makes it easier to shade efficiently with foveated rendering [GFD*12], since we can avoid resending geometry over multiple rendering passes. We use a configurable radial function to control the shading rate with a few parameters: the point that corresponds to the center of the gaze, c, aspect ratio, a, inner and outer minor radii, r^{i} and r^{o}, and inner and outer CP parameters, $s^{\min}$ and $s^{\max}$. These quantities are illustrated in Figure 6. For foveated rendering, r^{i} should be set to a size representing a view angle of about 5°, and $a = 1$ for a circular falloff function.

Although our shading system supports arbitrary positions for the high resolution region, we have observed that merely

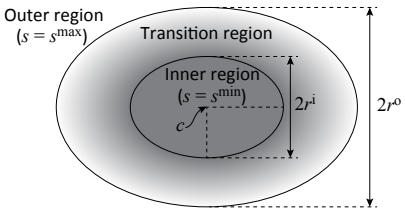

Figure 6: *Foveated rendering with different regions having different shading rates. The inner region has the highest shading rate, i.e., smallest value of (s_x, s_y), which decreases smoothly over the transition region to reach the minimum shading rate outside the outer ellipse. These rates, as well as the ellipses that demarcate the boundaries of these regions, are specified by the user.*

fixing c at the center of the screen and using a wider aspect also produces good results, most notably when rendering from a first person perspective. We call this technique *peripheral CPS rendering*, in contrast to proper foveated rendering, which is only possible with gaze tracking. Figure 8 (top) shows an example of peripheral CPS rendering.

6.4. Adaptive Shading for Camera Effects

Regions of the screen with motion or defocus blur typically have a narrow frequency response and can be shaded at a lower rate [RKLC*11, LD12, VTS*12]. With CPS, we can control the shading rate in such regions by setting CP parameters in the vertex shader that are proportional to the screen space velocity or circle of confusion at that vertex. Since the vertex shader is evaluated before clipping, there may be vertices behind the camera or at zero depth. For such cases, we refrain from reducing shading rate to ensure robustness.

The CP parameters can be determined separately for the x- and y-axes in order to generate anisotropic shading rates for motion blur. By assigning negative CP parameters for vertices in front of the focal plane and positive CP parameters for those behind, we can ensure that the CP parameters will interpolate to zero at the focal plane. Similarly, assigning signed CP parameters for velocity ensures zero values at stationary points inside a moving primitive. Since the screen space circle of confusion radius and velocity are linear functions in screen space, perspective-correct interpolation should be disabled for the CP parameter in such cases. Figure 8 (middle) shows a scene with defocus blur, rendered with a lower shading rate in blurred regions using CPS.

6.5. Multi-Rate Shading

We have explored several different applications where multi-rate shading provides a valuable tool for scaling quality vs. performance. Figure 8 (bottom) shows an example application, where a high quality ambient occlusion term is computed every 2×2 pixels using voxel cone tracing [CNS*11], while diffuse texture lookups are evaluated at a pixel rate

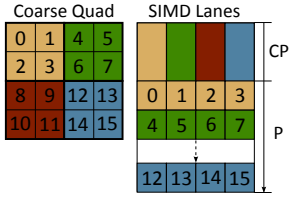

Figure 7: *Multi-rate shading with a coarse quad having a CP size of 2×2, scheduled in a single thread on a 4-wide SIMD processor. The coarse pixel (CP) phase executes first, followed by the pixel phase (P) which loops over all covered 2×2 pixels in the coarse quad.*

to retain most of the surface detail. Similarly, complex low-frequency lighting computations, such as indirect lighting, can also be evaluated at a lower rate.

Another example is locally increasing the shading rate in difficult regions. This may be done to compute shading at a pixel or sample rate only around specular features, and lower elsewhere. We have seen that it is fairly common for today's real-time workloads to implement a type of multi-rate shading using a two-pass approach. In this case, the first pass runs at pixel rate (MSAA), but discards difficult pixels which are marked in a stencil buffer. The second pass then runs shading per sample, but only for pixels marked in the stencil buffer. Using CPS, such algorithms can be converted to a single pass, which selectively computes the result in the pixel- and sample-rate shaders. We additionally gain the benefit of executing results at even lower rates, where possible.

CPS multi-rate shading can also be used to perform culling on a per CP [HAM07] (e.g., 4×4 pixels) basis. We show a proof-of-concept example in Figure 9.

7. Implementation

We evaluate CPS using a software implementation of the pipeline in a CPU based functional Direct3D 11 simulator. For our implementation, we use a tile size of 16×16 pixels and allow CP size widths and heights of 1, 2, or 4. In order to support the CPS programmer abstractions described in Section 5, we extend the DirectX High Level Shading Language and introduce new API functions. Our simulator is instrumented to measure the dynamic instruction count, in order to give an indication of the cost of shading. We also track all memory accesses to the color buffer for measuring color bandwidth.

Scheduling: As discussed in Section 5, in addition to pixel shading, we can introduce a coarse pixel (CP) as well as a sample rate shading phase in the pipeline. There are several potential implementation strategies for multi-rate shading depending on how these additional phases are scheduled across multiple processors and threads. We choose a scheduling strategy, where the CP, pixel, and sample rate shading phases are executed consecutively on the same

thread. Restricting the schedule in this manner enables a simple implementation, which requires significantly fewer changes to the pipeline. It eliminates the need to transport data across phases since the data can reside in the same registers. It also avoids complex flow control mechanisms for throttling inter-phase data.

Figure 7 shows a coarse quad with a CP size of 2×2 scheduled on a 4-wide SIMD processor, which we use for our measurements. First, the four CPs in the coarse quad are executed concurrently across the SIMD lanes which facilitates computing of finite differences. Following the coarse phase, the processor concurrently schedules 2×2 pixel quads inside the coarse quad, looping over all covered quads. Since our scheduling scheme requires movement of data across SIMD lanes when transitioning across phases, we assume the availability of processor instructions to facilitate this in an efficient manner.

For SIMD widths greater than 4 (say 8 or 12), we can schedule multiple coarse quads concurrently. However, since each coarse quad can have a different number of covered pixels, a higher SIMD width can also lead to a reduced utilization of some SIMD lanes as some pixel phase loops terminate early. For improved efficiency with large SIMD widths, one may choose a more optimal scheduling scheme that distributes the pixel phase work more evenly across SIMD lanes or even separate threads.

Color Compression: Shading at a lower rate than once per pixel has implications for how well color buffer compression works to reduce the memory bandwidth. A higher degree of uniformity in color values within 2×2 or larger pixel blocks generally reduces the entropy and makes compression more efficient. There are many existing methods for color compression [RHAM07, SWR*08, RSAM10, PLS12] that we expect to benefit from CPS. To exemplify that this works in practice, we have selected the scheme by Pool et al. [PLS12], which computes color differences between pixels, and then applies entropy coding over the differences using a Fibonacci encoder [FK96].

We have modified Pool et al.'s scheme so that the pixels are always visited in a hierarchical manner using a predefined Hilbert curve. As a consequence, the differences will first be computed inside a 2×2 quad, and then inside a 4×4 region, and so on. When CPS is enabled, several colors inside some of these regions are likely to be the same, producing zero differences, which are efficiently encoded using a Fibonacci encoder. We use a cache line size of 128 bytes, equivalent to 8×4 pixels for an RGBA8 render target. When a line is evicted from the color cache, it is compressed, and if the resulting size is less than or equal to 64 bytes, compression is successful, and we mark the line as compressed in a separate control buffer and send only one 64 byte transaction to memory. Otherwise, we send the uncompressed data to memory in two 64 byte transactions. We model a 64 kB color cache with an LRU replacement policy. Compression results are reported in Section 8.2.

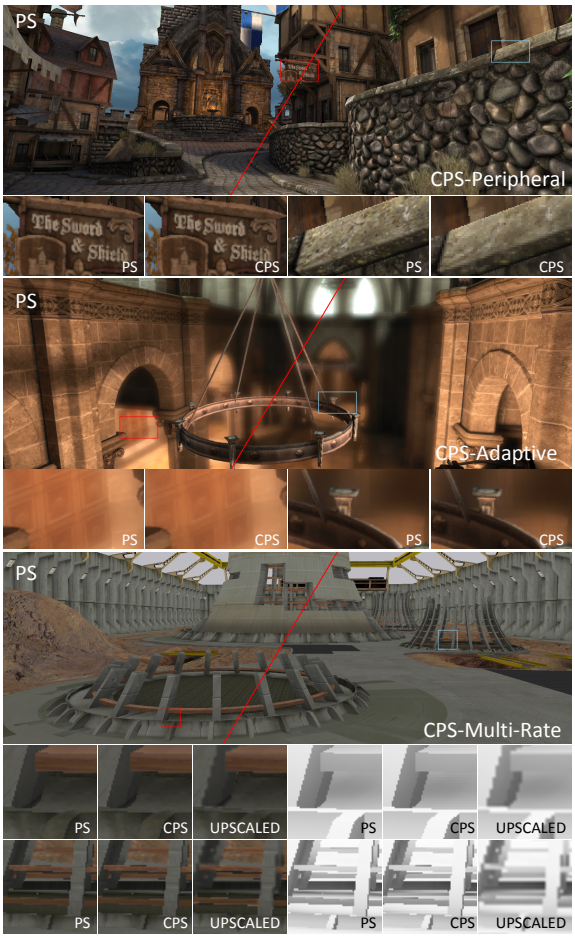

Figure 8: *A comparison of per-pixel shading (PS) and coarse pixel shading (CPS) for three different applications. Top row: the* CITADEL 1 *scene rendered with peripheral CPS with a coarse pixel (CP) size of* 1×1 *in the middle and* 2×2 *towards the screen border. Middle row: the* CITADEL 2 *scene rendered with a post process depth of field effect using a maximum CP size of* 2×2 *pixels in the blurred regions. Bottom row: the* POWER PLANT *scene rendered with high quality ambient occlusion computed with a CP size of* 2×2 *pixels and diffuse textures evaluated per pixel.*

8. Results

We evaluate four scenes for image quality and performance with CPS. Each scene corresponds to a specific application as listed in Table 1. The CITADEL scenes are from the Epic Citadel demo included in the Unreal Development Kit 2014-02 and the POWER PLANT scene is from the Microsoft DirectX SDK. For all scenes, we enable CPS only in the forward rendering passes. Post processing effects are shaded at a per-pixel rate to avoid blurring high frequency regions.

Scene	Application	SSIM (%)	PSNR (dB)
CITADEL 1	High-DPI displays	90.6	32.80
CITADEL 2	Peripheral CPS	89.6	31.96
CITADEL 3	Adaptive shading	98.0	39.91
POWER PLANT	Multi-Rate shading	99.8	51.46

Table 1: *Image quality metrics with CPS.*

The CITADEL 1 scene represents a typical use case for high-DPI displays where the scene is rendered with a uniform CP size of 2×2 pixels. For this scene, we render at a resolution of 2560×1440. All the other scenes are rendered at a resolution of 1920×1080. The CITADEL 2 scene shows an example application with peripheral CPS rendering. As shown in Figure 6, we select a shading rate of once every pixel inside an inner ellipse, 445×250 pixels large. The transition region is defined by an outer ellipse, 623×350 pixels large. Outside the transition region, we select a lower shading rate with a CP size of 2×2 pixels. The CITADEL 3 scene includes a post process depth of field effect, which is a feature available in the Unreal Engine. The CP size is adaptively controlled by assigning CP parameters proportional to the amount of blur on a per-vertex basis.

The POWER PLANT scene showcases the potential of CPS and multi-rate shading to enable complex low frequency effects with a significantly reduced computational overhead. In this scene, ambient occlusion is computed using voxel cone tracing [CNS*11], at a rate of approximately once every 2×2 pixels in the coarse pixel shader, while the rest of the shading is evaluated at a pixel rate in the pixel shader.

8.1. Image Quality

Table 1 shows the structural similarity index (SSIM) [WBSS04] and PSNR image quality metrics with CPS as compared to per-pixel shading for the four different scenes. We measure SSIM on the grayscale image without downsampling. With the suggested downsampling process for SSIM, we observed that the result was close to 99% for all scenes. The measured image quality scores are consistently high, indicating very little degradation in image quality. In the following paragraphs, we present a more detailed analysis of image quality for each of the four applications.

Uniform Downsampling: Figure 1 shows a comparison of the CITADEL 1 scene rendered with per-pixel shading and a fixed CP size of 2×2 pixels. We also evaluate an alternative approach (UPSCALING) where the scene is rendered at a quarter resolution and then up-scaled to the target display using a cubic filter [MN88]. Such a technique is often used to render to high-DPI displays.

The difference between the images rendered with CPS and per-pixel shading is hard to perceive when viewed on a high-DPI display. With UPSCALING however, there is a visible degradation in quality along silhouette edges due to poor sampling. This can be especially seen in the zoomed in regions which have sharp silhouette edges. For regions with high frequency textures, such as the sign board in Figure 1, a higher shading rate can still be selected if desired, by controlling the CP size.

Outside the silhouette regions, UPSCALING tends to produce smoother results than CPS. This is due to the simple approach used by CPS to reconstruct the high-DPI image from sub-sampled shading, where the shaded results for a CP are merely copied to the pixels inside it, instead of applying a reconstruction filter. However, we find that these differences are not perceptible on a high-DPI display.

Peripheral CPS rendering: Figure 8 (top) shows the CITADEL 2 scene rendered with peripheral CPS. The red box shows a region close to the center of the screen that is rendered at a pixel rate and the blue box shows a region at the periphery of the screen that is rendered with a CP size of 2×2 pixels. While shading in the peripheral region is blurrier, the sharp edges are preserved in both cases.

Adaptive Shading for Defocus Blur: Figure 8 (middle) shows the CITADEL 3 scene rendered with a post process defocus blur, where shading is evaluated at a lower rate in the blurred regions using CPS. The red box shows an out of focus region in the background where shading is evaluated with a CP size of 2×2. Although some differences can be seen in the zoomed in regions, these are barely observed in the original image, which is also reflected in the high SSIM score of 98.0%. We can also see that the post process defocus blur masks the effects of shading at a lower rate. The quality of the post process blur filter that is applied also affects the image quality with CPS. With a higher quality filter, we expect even fewer differences between CPS and PS.

Multi-rate Shading: Multi-rate shading presents one of the most compelling use cases for CPS with the potential to enable high quality lighting effects in future games. Figure 8 (bottom) shows the results for the POWER PLANT scene with multi-rate shading, which has the highest SSIM scores as well as a the largest reduction in shader computations with CPS. The combined result with ambient occlusion and diffuse textures is almost indistinguishable between the images with and without CPS. The differences can only be noticed when the ambient occlusion term is viewed separately as shown in the cropped images on the right. We also evaluate the UPSCALED approach discussed earlier, where ambient occlusion is rendered at a quarter resolution, upscaled and then combined with the diffuse texture. As compared to the UPSCALED result, image quality is remarkably better with CPS.

8.2. Performance

In order to evaluate potential performance gains with CPS, we measure two key parameters that are indicative of end

Scene	PS	CPS	Percentage of (CPS/PS)
CITADEL 1	523	260	49.7%
CITADEL 2	279	186	66.7%
CITADEL 3	444	359	80.9%
POWER PLANT	5115	1912	37.4%

Table 2: *Executed pixel shader instructions (in millions) with per-pixel shading (PS) and with CPS.*

to end performance. The first parameter is the number of pixel shader instructions that are executed, which estimates the computational overhead of pixel shading. The second parameter is the color bandwidth which typically constitutes a significant portion of the overall memory bandwidth. As discussed before, CPS can improve the effectiveness of color compression, significantly reducing color bandwidth.

Table 2 shows a comparison of the number of executed pixel shader instructions with per-pixel shading and CPS. These instructions include both forward rendering passes and post processing effects where CPS is not enabled. In spite of this, we find that in most cases, the instruction count with CPS is significantly lower than per-pixel shading, especially in the POWER PLANT scene where the shading cost with CPS is close to a third of per-pixel shading.

The CITADEL 3 scene has relatively smaller savings where the number of instructions with CPS is about 80 % of per-pixel shading. This scene has a significantly higher post processing cost, especially for computing the depth of field effect. Since CPS is not enabled for these expensive post processing passes, we see reduced savings.

Culling: In Figure 9, we show a simple example of using CPS for culling, similar to the work by Hasselgren and Akenine-Möller [HAM07]. This scene is based on the soft particles example from the DirectX 11 SDK, which renders 2D sprites and generates alpha and depth values in the pixel shader, by dynamically sampling a transparency texture to create the illusion of an animated smoke volume.

We have modified this application to introduce a coarse pixel shading phase with a CP size of 4×4 pixels that culls particle regions that are fully transparent or occluded by opaque geometry. We do this by computing an additional transparency texture, storing a conservative max-value mipmap of the alpha. Similarly we also compute a depth texture where each texel stores the maximum depth buffer value within a 4×4 pixel region. In the CP phase, we perform a lookup in the correct mip-level of the transparency texture to cull transparent regions followed by a lookup in the depth texture to cull occluded regions.

Including the added cost of creating the coarse depth buffer and running the cull shader, we save about 35% of the pixel shader (and CPS) instructions, and 45% of the texture lookups. For a fair comparison, we also enhance the original particle shader with an early discard against $\alpha = 0$. Furthermore, our numbers do not include saved interpolation setup due to culling, which may be beneficial on some architectures. It might also be possible to leverage early depth culling in hardware, by rendering particles with a certain minimum depth to ensure that the depth tests are conservative. However determining such a minimum depth might be impractical, for example, in cases where the depth is animated or generated procedurally in the pixel shader.

Figure 9: *A simple culling example of soft particles (left). The right image shows the culling potential by discarding fully transparent regions (yellow) in the coarse pixel shader, and performing a coarse depth test (red).*

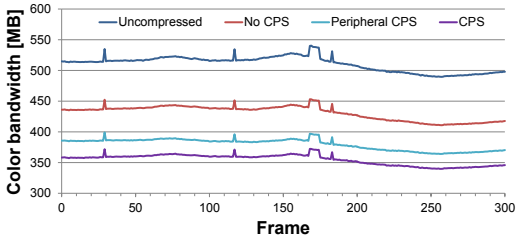

Figure 10: *Post color cache memory traffic for a 300 frame animation of* CITADEL1.

Memory Bandwidth: Figure 10 compares post color cache memory traffic with per-pixel shading and CPS for a 300 frame animation of the CITADEL 1 scene, using the modified Pool et al.'s scheme for color compression. Compression without any CPS (*No CPS*) achieves 16% traffic reduction on average, while with CPS, the average reduction is 31%. For peripheral CPS rendering, as expected, the reduction is slightly less (26%) than uniform downsampling, but is still significant. Note that with our color compression scheme, the compression ratio is limited to 50%. With an improved scheme, we can expect even higher compression ratios with CPS. For scenes with multi-rate shading, we do not expect significant reduction in color traffic with CPS since the resulting color would still have high frequency content produced by the pixel-rate or sample-rate stages.

In addition to saving bandwidth with better compression, CPS can also reduce the texture bandwidth due to the fewer texture sample instructions that are executed and also since the textures would be fetched from a higher mip-level. CPS can also be used to render to a render target, which is then accessed in a another pass through the texture sampler. In such cases, the texture bandwidth can be reduced as well. It would be interesting to evaluate the memory traffic with CPS for these cases, but we leave such analyses for future work.

9. Conclusion

We have introduced coarse pixel shading (CPS), a new algorithm that can significantly reduce the cost of shading with little to no perceivable impact on image quality. CPS fits naturally in the evolution of the real-time graphics pipeline as it introduces a new degree of flexibility through programmable shading rates, while still addressing a real need for energy efficiency for the fast growing market of hand-held devices. We have demonstrated how our algorithm can be used in several rendering scenarios and how it can be easily incorporated into an existing graphics pipeline, through small incremental changes to the graphics hardware and extensions to the user level interface. Based on this finding, we are optimistic that an evolutionary step towards flexible shading can be achieved in the near future, spurring the graphics community to develop several innovative techniques for efficient rendering.

Acknowledgements: We thank Prasoonkumar Surti, Benty Nir, Uzi Sarel and Tomer Bar On for their contributions. We also thank David Blythe, Charles Lingle, and Tom Piazza for supporting this research. The CITADEL test scenes are courtesy of Epic Games, Inc. Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg foundation.

References

[AHTAM14] ANDERSSON M., HASSELGREN J., TOTH R., AKENINE-MÖLLER T.: Adaptive Texture Space Shading for Stochastic Rasterization. *Computer Graphics Forum (Eurographics 2014), 33*, 2 (2014). 2

[Ake93] AKELEY K.: RealityEngine Graphics. In *Proceedings of SIGGRAPH 93* (1993), ACM, pp. 109–116. 2

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16. 2

[BFM10] BURNS C. A., FATAHALIAN K., MARK W. R.: A Lazy Object-Space Shading Architecture with Decoupled Sampling. In *High Performance Graphics* (2010), pp. 19–28. 2

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (1987), vol. 21, ACM, pp. 95–102. 2

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011) 30*, 7 (2011). 6, 8

[CTM13] CLARBERG P., TOTH R., MUNKBERG J.: A Sort-Based Deferred Shading Architecture for Decoupled Sampling. *ACM Transactions on Graphics, 32*, 4 (2013), 141:1–141:10. 2

[DHS*05] DURAND F., HOLZSCHUCH N., SOLER C., CHAN E., SILLION F. X.: A Frequency Analysis of Light Transport. *ACM Transactions on Graphics, 24*, 3 (2005), 1115–1126. 2

[FH11] FOLEY T., HANRAHAN P.: Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Transactions on Graphics, 30*, 4 (July 2011), 107:1–107:12. 5

[FK96] FRAENKEL A. S., KLEIN S. T.: Robust Universal Complete Codes for Transmission and Compression. *Discrete Applied Mathematics*, 64 (1996), 31–55. 7

[GBAM11] GRIBEL C. J., BARRINGER R., AKENINE-MÖLLER T.: High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility. *ACM Transactions on Graphics, 30*, 4 (2011), 54:1–54:12. 2

[GFD*12] GUENTER B., FINCH M., DRUCKER S., TAN D., SNYDER J.: Foveated 3D Graphics. *ACM Transactions on Graphics, 31*, 6 (2012), 164:1–164:10. 2, 3, 5

[HAM07] HASSELGREN J., AKENINE-MÖLLER T.: PCU: The Programmable Culling Unit. *ACM Transactions on Graphics, 26*, 3 (2007), 92.1–92.10. 2, 6, 9

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled Deferred Shading for Hardware Rasterization. In *Symposium on Interactive 3D Graphics and Games* (2012), pp. 143–150. 2, 6

[MCH*11] MUNKBERG J., CLARBERG P., HASSELGREN J., TOTH R., SUGIHARA M., AKENINE-MÖLLER T.: Hierarchical Stochastic Motion Blur Rasterization. In *High Performance Graphics* (2011), pp. 107–118. 2

[Mic12] MICROSOFT: Rasterization Rules, 2012. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/cc627092(v=vs.85).aspx. 4

[MN88] MITCHELL D. P., NETRAVALI A. N.: Reconstruction Filters in Computer Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (1988), vol. 22, ACM. 8

[PLS12] POOL J., LASTRA A., SINGH M.: Lossless Compression of Variable-Precision Floating-Point Buffers on GPUs. In *Symposium on Interactive 3D Graphics and Games* (2012), pp. 47–54. 7

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2000* (2001), ACM, pp. 159–170. 5

[Poo12] POOL J.: *Energy-Precision Tradeoffs in the Graphics Pipeline*. PhD thesis, 2012. 2

[RHAM07] RASMUSSON J., HASSELGREN J., AKENINE-MÖLLER T.: Exact and Error-Bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware* (2007), pp. 41–48. 7

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics, 30*, 3 (2011), 17:1–17:17. 2, 4, 6

[RSAM10] RASMUSSON J., STRÖM J., AKENINE-MÖLLER T.: Error-bounded Lossy Compression of Floating-point Color Buffers using Quadtree Decomposition. *The Visual Computer, 26*, 1 (2010), 17–30. 7

[SMD*06] STOLL G., MARK W. R., DJEU P., WANG R., ELHASSAN I.: *Razor: An Architecture for Dynamic Multiresolution Ray Tracing*. Tech. Rep. TR-06-21, Dept. of Computer Science, University of Texas at Austin, 2006. 2

[SWR*08] STRÖM J., WENNERSTEN P., RASMUSSON J., HASSELGREN J., MUNKBERG J., CLARBERG P., AKENINE-MÖLLER T.: Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware* (2008), pp. 75–84. 7

[VTS*12] VAIDYANATHAN K., TOTH R., SALVI M., BOULOS S., LEFOHN A.: Adaptive Image Space Shading for Motion and Defocus Blur. In *High Performance Graphics* (2012), pp. 13–21. 2, 6

[WBSS04] WANG Z., BOVIK A., SHEIKH H., SIMONCELLI E.: Image Quality Assessment: from Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing, 13*, 4 (april 2004), 600–612. 8