AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors

Petrik Clarberg[1] Robert Toth[1] Jon Hasselgren[1] Jim Nilsson[1] Tomas Akenine-Möller[1,2]

[1]Intel Corporation [2]Lund University

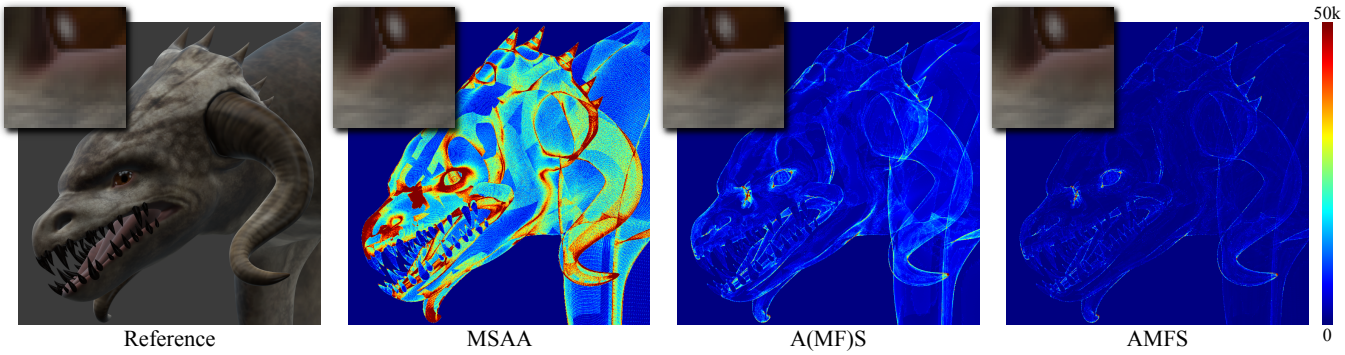


Figure 1: *Subdivision surfaces are widely used in offline rendering and an important tool for content creation. However, real-time rendering of tessellated geometry is still fairly expensive, as current pixel shading methods, e.g., multisampling antialiasing (MSAA), do not scale well with geometric complexity. With our method (AMFS), pixel shading is tied to the coarse input patches and reused between triangles, effectively decoupling the shading cost from the tessellation level, as shown in this example. AMFS can evaluate different parts of shaders at different frequencies, allowing very efficient shading. Our shading cost using a single shading frequency (middle right) is here 26% of that of MSAA (middle left), while the shading cost using multiple shading frequencies (far right) is only 7% of that of MSAA.*

Abstract

We propose a powerful hardware architecture for pixel shading, which enables flexible control of shading rates and automatic shading reuse between triangles in tessellated primitives. The main goal is efficient pixel shading for moderately to finely tessellated geometry, which is not handled well by current GPUs. Our method effectively decouples the cost of pixel shading from the geometric complexity. It thereby enables a wider use of tessellation and fine geometry, even at very limited power budgets. The core idea is to shade over small local grids in parametric patch space, and reuse shading for nearby samples. We also support the decomposition of shaders into multiple parts, which are shaded at different frequencies. Shading rates can be locally and adaptively controlled, in order to direct the computations to visually important areas and to provide performance scaling with a graceful degradation of quality. Another important benefit of shading in patch space is that it allows efficient rendering of distribution effects, which further closes the gap between real-time and offline rendering.

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

Keywords: graphics hardware, pixel shading, tessellation, GPU

Links: DL PDF

1 Introduction

In today's graphics processors and real-time applications, a large portion of the computational resources and power budget is spent on executing pixel shading on the programmable cores of the GPU. For over twenty years, the prevailing method has been *multisampling antialiasing* (MSAA) [Akeley 1993], where shading is invoked once per triangle and pixel. The cost of pixel shading is therefore tightly coupled to both the geometric complexity and the screen resolution, and it has been necessary to keep both low.

This is competing with the developers' goals of providing a richer visual environment. Tessellation is a tantalizing means to reach that goal, as highly detailed geometry can be generated without having to store and transfer huge polygonal meshes. However, tessellation drastically increases the cost of pixel shading. There is also a trend towards very high resolution displays in consumer devices, motivated by the reduction of distracting aliasing. This further increases the amount of shading work. For these reasons, rethinking the way pixel shading works is necessary, in order to reach a higher level of visual realism even on low power devices.

We propose a new hardware architecture that addresses these concerns by making the pixel shading work largely independent of the level of tessellation and screen resolution. Figure 1 shows an example of the substantial savings possible. Following previous work [Burns et al. 2010; Ragan-Kelley et al. 2011], we *decouple* pixel shading from screen space; similar to Burns et al.'s method, shading is lazily evaluated in a parametric space defined over each high-level input primitive, e.g., coarse patch. This means shading is efficiently reused between all the triangles in a patch. The shading rate over a patch is in our system, however, *not* a priori determined, but instead locally and automatically chosen based on the final tessellated geometry. This avoids potential problems with under/over-shading due to local curvature or displacement, and limits shading reuse between triangles with widely differing orientations.

The second and perhaps most important feature of our architecture, is that it allows lazy shading and reuse simultaneously at *multi-*

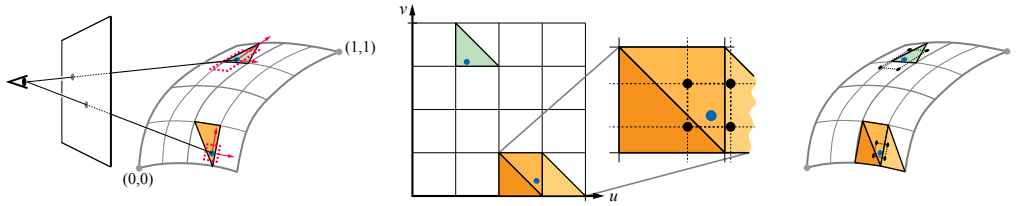


Figure 2: *Left: shading requests are sent for the points (blue) on two triangles in a patch, which are being rasterized. Note that the size of the screen-space footprints on the surface are different for these two points due to the orientation and curvature. Middle: one of the shading points (blue) is visualized in the patch's parametric (u, v)-space, and a shading quad of appropriate size is placed in its vicinity. Right: the four points forming shading quads are shown on the triangles of the patch. Note that the shading points of a quad may map to other triangles.*

ple different frequencies. The pixel shading operations can, for example, be split into a low-frequency component that computes approximate global illumination at a much lower rate than direct lighting. None of the existing pixel shading methods support multi-frequency shading. The overarching goal is to provide flexible control over the amount of pixel shading work, independently of the geometry and display resolution. This allows an application to stay within a given frame time or power budget, while maximizing image quality. In summary, the main features of our architecture are:

- ▷ Pixel shading is lazily evaluated in patch-parametric space and reused between triangles in a patch.
- ▷ The shading resolution is automatically adapted based on the local geometry and user-defined/computed shading rates.
- ▷ Shaders can be partitioned into multiple different components, which are shaded at different frequencies.
- ▷ The method integrates well into existing pipelines, and natively supports motion/defocus blur.

The rest of this paper is organized as follows. First, we will briefly discuss some related work. This is followed by a system overview and detailed description in Section 3 and 4, respectively. Our simulator, results, and conclusion are presented thereafter.

2 Related Work

With *multisampling antialiasing* (MSAA) [Akeley 1993] in current GPUs, shading is evaluated once per pixel rather than at the individual visibility samples of each rasterized triangle. The result is written to all covered samples within the current pixel (i.e., fragment) and triangle. In practice, shading is executed at a larger granularity, e.g., quads of 2×2 pixels. The total number of shader invocations thus depends on how many unique triangles overlap each such screen-space region with at least one visible sample. The shading cost rises dramatically when triangles become small due to tessellation, or when they are smeared out by motion/defocus blur. This is a fundamental problem of current screen-space shading methods.

Ragan-Kelley et al. [2011] suggest *decoupled sampling* of shading and visibility, where shading is lazily evaluated over a per-triangle shading grid in image space, and cached for reuse. Their method elegantly reduces the amount of shading with stochastic rendering, but it does not address tessellation and shading reuse between primitives. Fatahalian et al. [2010] focus on the latter problem and reduce over-shading for small primitives by *merging* quad fragments from adjacent primitives prior to shading. Their method operates in screen space and thus does not allow varying shading rates, and it was also not designed for stochastic rendering.

In traditional *object-space* shading methods, exemplified by the Reyes architecture for offline rendering [Cook et al. 1987], shading is evaluated at the vertices of a finely tessellated micropolygon mesh and interpolated over connected micropolygons. Since the shading rate is tightly coupled to the geometric complexity, the method does not provide the flexibility we seek. Burns et al. [2010] add flexibility by shading on a uniform object-space grid, separate from the micropolygon grid. The shading grid is shared within a sub-patch in the split/dice tessellation stage. Its dimensions are computed to yield approximately one shading point per pixel by estimating the projected screen-space area of the sub-patch. The main difference to our work is that we allow varying shading rates across a patch, including multi-frequency evaluation.

The Razor ray tracing architecture [Stoll et al. 2006] similarly performs on-demand shading of regularly spaced grids in object space, which are chosen from a set of predefined power-of-two tessellation grids. However, only modest gains are reported, as each such grid that is needed by at least one ray is shaded. Gribel et al. [2011] propose an object-space shading cache designed for high-quality offline rendering of motion blur. Their solution stores a per-object mipmap hierarchy of shading values, which is lazily populated during rendering. This is similar to our work in that the shading rate is locally and automatically adapted to the geometry.

Several authors note the importance of locally reducing the shading rate, e.g., in the presence of blur [Liktor and Dachsbacher 2012; Vaidyanathan et al. 2012]. These methods are orthogonal and compatible with our approach, and are encouraged if stochastic rendering is used. A further means to reduce the amount of shading work is to defer shading until after visibility has been determined [Liktor and Dachsbacher 2012; Clarberg et al. 2013]. The focus of these papers is on efficient shading with stochastic rasterization [Akenine-Möller et al. 2007], while our primary goals are to reduce shading also in traditional pipelines and to support multi-frequency shading.

Pixel shading at multiple different rates has not been widely explored in previous work. Current GPUs and graphics APIs support both per-pixel (MSAA) and per-sample shading, but the two rates are mutually exclusive. Hasselgren et al. [2007] took a step towards multi-rate shading by executing culling computations over tiles, prior to the pixel shader. Another strategy is to move computations to the vertex shader, which is tied to the geometry. Some techniques for sparse evaluation and caching in offline rendering, e.g., irradiance caching [Ward et al. 1988], can also be seen as a type of dual-rate shading. Illumination interpolated from such sparse representations is often combined with per-sample shading of higher-frequency effects, such as direct illumination.

While our work is similar to many of the discussed systems [Stoll et al. 2006; Burns et al. 2010; Ragan-Kelley et al. 2011; Gribel et al. 2011], the application domains differ; we primarily aim to *evolve* the current real-time graphics pipeline, without breaking legacy applications. Our architecture is also the first to support pixel shading at multiple different rates, unrestricted by the tessellation or visibility sampling rates, with broad reuse of shading across primitives. In the following, we describe our approach in more detail.

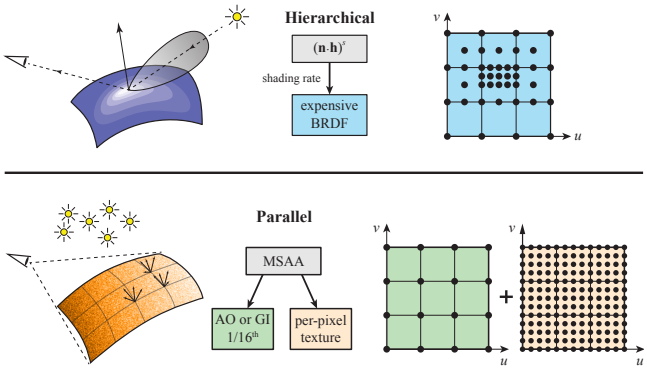


Figure 3: *Two types of multi-frequency shading. Top: our approach can be used hierarchically, e.g., to inexpensively compute a shading frequency that places more shading samples near the main lobe(s) of a reflection model. Bottom: parallel evaluation is also possible, where a texture and diffuse shading is evaluated per pixel, while ambient occlusion (AO) or global illumination (GI) can be computed at a much coarser granularity. The total amount of shading work may be substantially reduced in these examples.*

3 Overview

In our system, pixel shading is executed in parametric *patch space* rather than in screen space, as illustrated in Figure 2. While rasterizing each triangle in a tessellated patch, the rasterizer generates *shading requests*. These are answered by defining a small, local shading grid in patch (u, v)-space, over which shading is computed and cached for reuse. The scale and placement of the local shading grids are automatically determined based on the local geometry and/or programmatic control. Note that the local shading grids are not tied to the underlying tessellation, and may thus extend beyond the current triangle or over multiple triangles in a patch. In smooth regions, the grids tend to be similar and shading is automatically reused between triangles. Where there is faster change, the local grids are more likely to differ in scale, which intuitively limits shading reuse in difficult regions.

The main use case for our architecture is *adaptive multi-frequency shading* (AMFS). Figure 3 shows two examples of this. The top example shows how an inexpensive computation can be used to control the shading frequency of a more expensive lighting model, e.g., involving complex BRDFs or light scattering. In this case, more shading samples are placed around the specular peak(s). The bottom example shows how a shader can be run in parallel at multiple (in this case two) different frequencies. The texture and diffuse shading is executed at per-pixel rate, while slowly-changing functions, such as indirect illumination, can be computed at a much lower rate (e.g., one shading sample per 4×4 grid points). Another possibility is to vary the shading rate spatially to allow high-definition shading centered around the viewing point [Guenter et al. 2012] (although they also reduce the visibility sampling rate).

When using our algorithm at a single fixed shading frequency (i.e., approximately once per pixel), we denote it A(MF)S to emphasize that its multi-frequency (MF) capabilities are unused. Nevertheless, shading is in both cases effectively reused between the triangles in a patch, while the shading rate is adapted to the local geometry of the displaced/curved surface. This is illustrated in Figure 4. Note that the insertion/reuse of shading points happen automatically, without relying on a fixed shading grid as in previous work.

Figure 5 shows an architectural overview of our pipeline. The top row of units represent a traditional graphics processor supporting

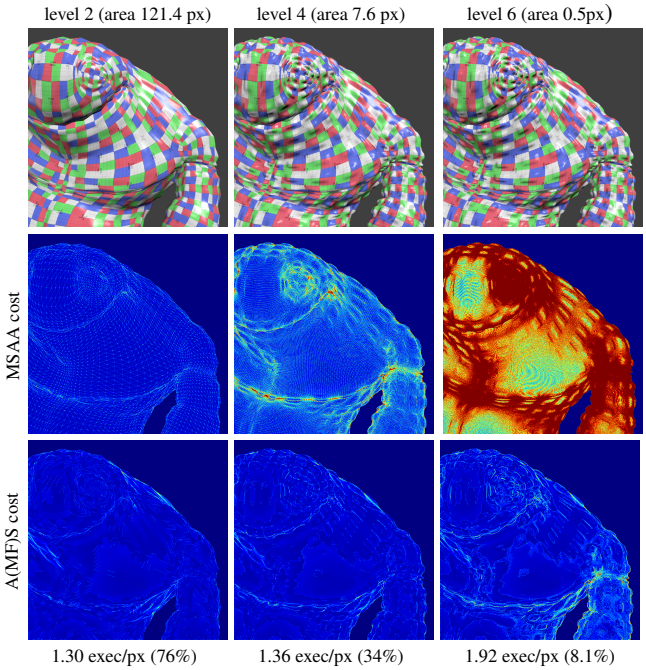


Figure 4: *The* BIGGUY *scene at three levels of subdivision using 8 samples/pixel. Tessellation combined with displacement mapping allows for generation of fine surface detail. However, the smaller primitives are problematic in current GPUs using MSAA. Our algorithm locally adapts the patch-space shading rate to provide good image quality without significant over-shading. The middle column represents our main design point, i.e., moderate tessellations with 5–20 pixels sized triangles, with $\sim 3\times$ reduction in shading work (values in parenthesis are costs relative MSAA).*

current APIs, with the exception that pixel shading is now handled by a *shading engine* operating in patch space. At a high level, the shading engine is responsible for lazily evaluating and caching pixel shading, which is computed over the small local shading grids on the patch. Shading can thus be shared between, potentially, all the triangles in a patch. This is in contrast to current pipelines, which process triangles one-by-one and do not reuse pixel shading.

4 Adaptive Multi-Frequency Shading

We will now describe the details of the operations performed in the shading engine. For the purpose of this exposition, assume a single patch is in flight. This does not preclude an actual implementation from being deeply pipelined to handle multiple patches in parallel.

4.1 Shading Requests

Current GPUs conceptually execute a pixel shader for each triangle and covered sample, or group of samples within a pixel (i.e., fragment) if multisampling antialiasing (MSAA) is enabled. In our system, shading is instead computed by issuing *shading requests* to our shading engine, which ultimately returns the color of the requested sample or fragment. A shading request consists of the parametric position $\mathbf{u} = (u, v)$ on the patch at which to shade, along with its screen-space derivatives $\mathbf{u}_x = \partial\mathbf{u}/\partial x$ and $\mathbf{u}_y = \partial\mathbf{u}/\partial y$.

The parametric coordinates $\mathbf{u} \in [0, 1]^2$ (and $u+v \leq 1$ in case of triangular patches) represent a contiguous parameterization of the patch, which is key to enabling shading reuse. The derivatives define the extent and anisotropy of a screen-space pixel in patch space

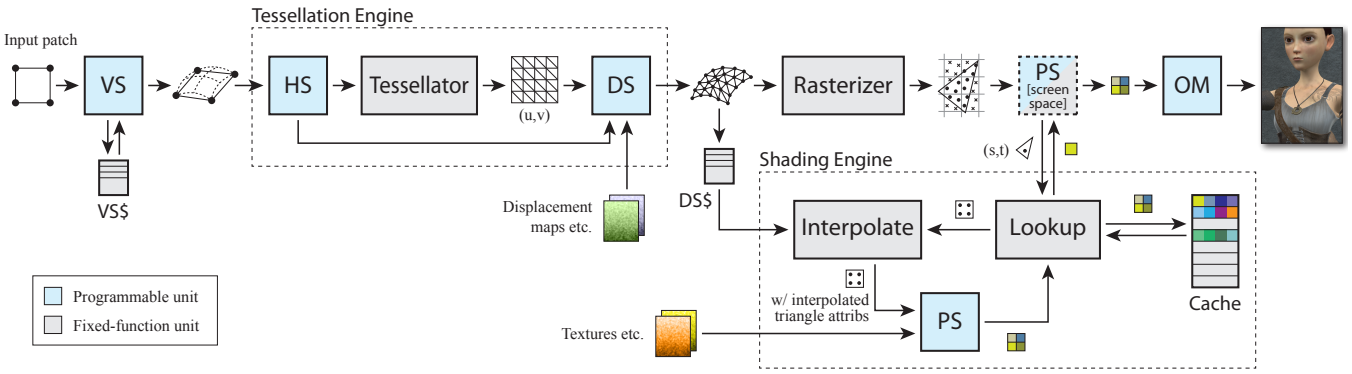


Figure 5: *Overview of a modern graphics pipeline, augmented with our adaptive multi-frequency (AMFS) shading system. The tessellation engine (in Direct3D 11 including hull (HS) and domain (DS) shader stages), takes a patch after vertex shading (VS) as input to generate a set of tessellated triangles. The rasterizer operates on one triangle at a time, and generates barycentric coordinates for each covered sample that passes a depth/stencil test (omitted). These are shaded and written to the render target(s) by the output merger (OM). We retain the possibility of executing a screen-space pixel shader (PS). However, the bulk of the shading work is done in a new shading engine. Internally, it first computes an appropriate small, local shading grid in patch space, and directly returns the shaded result if it is cached. Otherwise, an interpolation unit performs a patch-to-triangle lookup and fetches the relevant domain-shaded vertices through a cache (DS$) to setup a shading quad. This is shaded by the (patch-space) pixel shader, and the result is cached and returned.*

(c.f., Figure 6). Informally, the two 2D axes, $\mathbf{u}_x$ and $\mathbf{u}_y$, describe the change in $\mathbf{u}$ when stepping one pixel in x and y, respectively. The rasterizer analytically computes these values by transforming the hit point on a rasterized triangle and its derivatives, from triangle barycentric space to patch space. This is an affine 2×3 transform, which is constant per triangle (see Appendix A). Thus, the operations associated with issuing a shading request can often be performed in fixed-function hardware. However, we choose to retain the notion of an (optional) pixel shader operating in screen space, partly to support legacy applications, but also since certain operations benefit from knowing the exact screen-space position. Examples include frame buffer compositing, i.e., programmable blending, and read/write access to per-pixel data structures.

Our system makes it possible to implement *multi-frequency shading* by issuing several different shading requests, either in parallel or hierarchically, to compute partial results at different frequencies. For this purpose, we include a *shader kernel* identifier, k, in the request. At each request, the issuing shader (screen-space or patch-space) may also apply an arbitrary scaling and/or translation of the shading point and its derivatives to locally *adapt* the shading density. In summary, the shading engine is formally responsible for evaluating a function, f, expressed as shown below:

$$color = f(k, \mathbf{u}, \partial\mathbf{u}/\partial x, \partial\mathbf{u}/\partial y). \quad (1)$$

Also note that the screen-space shader (if used) runs at a rate of either once per sample, or once per fragment if MSAA is enabled, and thus issues shading requests at that rate.

4.2 Local Shading Grid Computation

The first step performed for each shading request is to compute an appropriate *local shading grid* in patch space. This is done by the unit labeled LOOKUP in Figure 5. Note that the "area" of a pixel projected to patch space can be approximated as the area of the parallelogram spanned by $\mathbf{u}_x$ and $\mathbf{u}_y$:

$$A_{\text{pixel}} \approx |\mathbf{u}_x \times \mathbf{u}_y| = |u_x v_y - v_x u_y|. \quad (2)$$

Based on this information, we divide the patch into a (local) axis-aligned shading grid. In the canonical case, to reach a shading rate of approximately once per pixel, a target resolution of $r_u \times r_v$ grid

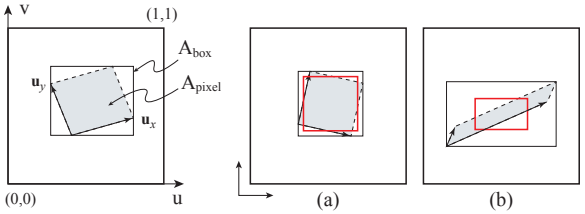
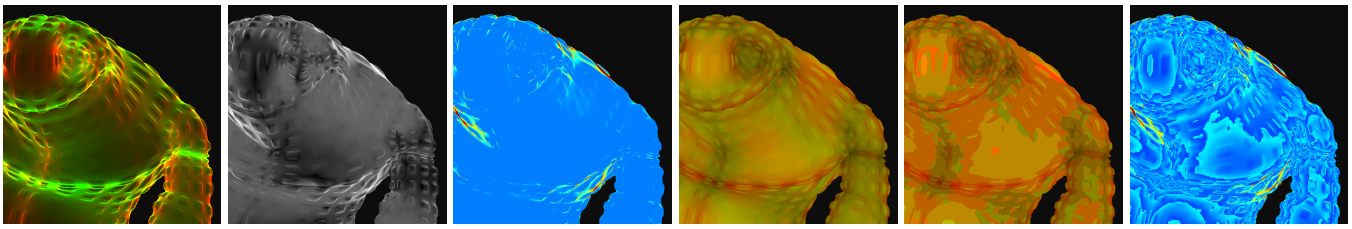


Figure 6: *We compute an axis-aligned target shading resolution based on the bounds of the partial derivatives, $\mathbf{u}_x$ and $\mathbf{u}_y$, scaled to pixel area (red boxes). The ratio $\alpha = A_{\text{box}}/A_{\text{pixel}}$ measures the distortion between screen and patch space, which we take into account to locally increase the shading rate. The most difficult case is anisotropically stretched and rotated patches.*

points is chosen so that the area of a grid cell is equal to A_{pixel}. We base this computation on the bounding box of $\mathbf{u}_x$ and $\mathbf{v}_x$ (with area A_{box}) in patch space. Note that the distortion due to the anisotropy and orientation of a patch may cause the grid points to lie significantly outside the pixel. To reduce this effect, the grid resolution is locally increased based on the ratio $\alpha = A_{\text{box}}/A_{\text{pixel}}$. These concepts are illustrated in Figure 6, with details in Appendix B.

Figure 7 (a) shows the magnitudes of the screen-space derivatives, $|\mathbf{u}_x|$ and $|\mathbf{u}_y|$, for the scene in Figure 4 (middle). In (b), the value of $\alpha \in [1, 8]$ is shown. Note that α is largest for patches rotated around 45 degrees and stretched due to perspective, i.e., near silhouettes, as expected. The target shading rate using our method is shown in (c), which is one shading point per pixel, except in a few difficult regions. The corresponding target shading resolution (r_u, r_v) (here in the range $[0, 256]$) is shown in (d).

In our system, pixel shading is executed and cached at the granularity of a *shading quad*, i.e., 2×2 grid points, in order to support shader derivatives through finite differences. This is standard practice, and multiple such shading quads may be buffered and shaded together. If subsequent shading requests map to the same shading quad *and* the same grid resolution, the previously computed results will be reused, as described below. It is thus important that the number of unique grid resolutions is limited, as otherwise no reuse would occur. We have chosen to quantize the grid resolution (r_u, r_v) to power-of-twos independently along each dimension, to



(a) derivatives $|\mathbf{u}_x|, |\mathbf{u}_y|$ **(b)** distortion α **(c)** target rate **(d)** resolution r_u, r_v **(e)** quantized resolution **(f)** final shading rate

Figure 7: *Our shading grid computation is based on **(a)** the screen-space derivatives, and **(b)** a measure of the distortion. This results in **(c)** a target shading rate, and associated **(d)** patch-space shading resolution. The latter is in **(e)** quantized to power-of-twos (independently along u and v), which gives the final shading rate shown in **(f)**.*

provide a discrete set of grid resolutions, but still to some extent respect the aspect ratio of the target resolution. The final quantized shading grid resolution is denoted $\mathbf{n} = (n_u, n_v)$ below.

Figure 7 (e) shows the values of $\mathbf{n}$, which correspond to the final shading rate in (f). The rate varies around once per pixel due to the local geometry. Since the shading grid computation is performed locally for each shading request, a patch may be shaded at different rates in different regions, as shown in the figure. At each transition in grid resolutions, some over-shading may occur. However, the effect is limited compared to using a single per-patch rate [Burns et al. 2010], which has to be conservatively chosen. This is a core strength of our method, as it allows the shading rate to be automatically *adapted* to fit the local displaced geometry.

So far we have looked at the canonical case of shading around once per pixel. To vary this rate, the user can *scale* the input derivatives that drive the computation. For example, using $\eta\,(\mathbf{u}_x, \mathbf{u}_y)$ as arguments in Equation 1, the system will shade approximately once per $\eta \times \eta$ pixels. Note that η does not have to be an integer, and that it may be varied spatially and independently along the two axes.

4.3 Cache Queries and Filtering

In our architecture, shading is lazily executed and cached. We have explored both nearest neighbor and bilinearly filtered lookups. In the former case, a single cache query is performed and the resulting color is returned, while in the latter, four cache queries are issued. Note that it would be possible to extend this mechanism to higher-order filtering schemes. However, we have found that simple nearest-neighbor lookups often give sufficiently good results at shading rates ≥ 1, while we use bilinear filtering for downsampled shading. A more thorough investigation is necessary though.

To perform a cache query, the shading point $\mathbf{u}$ is placed at the nearest grid point at the computed quantized grid resolution $\mathbf{n}$, or at the nearest four points if bilinear interpolation is used. For each such quantized shading point, the index of the shading quad q that it belongs to is first computed (through simple bit shifts), and then a shading cache lookup is done using the key:

$$\text{key} = h(k, q, \mathbf{n}), \tag{3}$$

where h is an appropriately chosen hash function. The shading cache is a memoization cache that operates similar to what Ragan-Kelley et al. [2011] proposed for decoupled sampling. The main difference lies in Equation 3, i.e., that we include the shader kernel ID k, and the quantized grid resolution $\mathbf{n}$, in the tag. Cache records are, in our case, evicted from the shading cache only when a patch is done or when the cache is full. Note that, similar to their work, shading is not order dependent since a shading quad at any quantized resolution always evaluates to the same result, independent of which shading request triggered its (re-)execution.

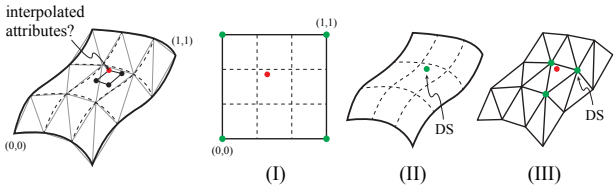


Figure 8: *The position and other attributes of each vertex in a tessellated patch is computed by executing a domain shader (DS). Prior to pixel shading, each shading quad has to be filled in with interpolated attributes at each of its shading points (one shown in red). There are several options: (I) interpolate from the patch's corners, (II) re-execute the DS at the shading point to evaluate its attributes based on the continuous surface, and the approach we take, (III) interpolate between the already domain-shaded vertices.*

4.4 Attribute Interpolation

Whenever a requested shading quad does *not* already exist in the cache, it will be shaded. In this case, the system first performs *attribute interpolation*, before the shading quad is put in the queue for pixel shading. The interpolants are attributes output by the domain shader, e.g., texture coordinates, normals, etc., which are fed to the pixel shader as inputs. The associated operations are performed in the INTERPOLATE unit in Figure 5.

In the traditional pipeline, attributes are interpolated in the plane of each triangle using barycentric coordinates. Things are more complicated in patch space, since a shading quad may overlap many different triangles. Figure 8 illustrates a few different strategies. Interpolating from the patch corners (I) is rarely useful, as it does not consider the shape of the patch. At the other end of the scale, one can evaluate the underlying continuous surface (II). This involves (re-)executing the domain shader (DS), or a subset of it, at each shading point, which is costly. Another problem is the discrepancy between the continuous surface and the rasterized triangulated surface. At larger than subpixel-sized triangles this can be significant, which causes problems with, e.g., shadow maps. Therefore, we opt for interpolating attributes over the final triangulated patch (III), which is further described below. This is both robust and efficient, as it avoids extra DS invocations. However, note that the user may still manually perform I or II in shader code.

The input to the interpolation unit is a quad with associated (u, v) coordinates. The unit also gets information from the tessellator about the currently used tessellation rates and scheme. To evaluate the interpolants at a point $\mathbf{u}$, we start by locating the triangle in which the point falls. Then, barycentric interpolation between its three vertices is performed. The task at hand is thus to perform a mapping P: $(u, v) \rightarrow (j, s, t)$, where j is the triangle-in-patch index and $(1\text{–}s\text{–}t, s, t)$ are the barycentric coordinates in that triangle.

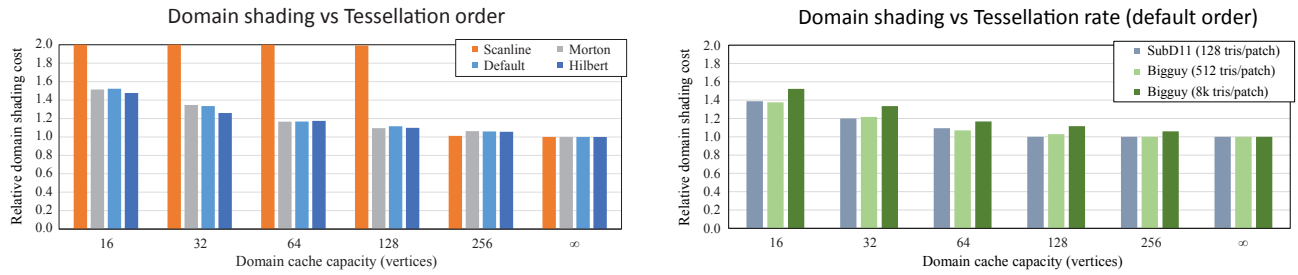


Figure 9: *Our algorithm accesses triangles in the vicinity of the currently rasterized triangle during patch-space pixel shading, and thus benefits from a tessellation scheme with good spatial locality. This is not unique to AMFS, as a good triangle ordering also improves memory coherency for texture/buffer accesses. Left: four different patterns were tested for the scene in Figure 4 (right), with the default (as generated by* `OpenSubdiv`*) giving results close to the Morton and Hilbert space-filling curves. Note that scanline order causes cache thrashing when an entire row does not fit, as expected. Right: three different tessellation levels were tested. Even at fine tessellations, e.g.,* $2 \times 64^2 = 8k$ *triangles per patch, a moderately sized (64 vertices) domain shading cache works well (7.0–16.7% DS re-shading).*

The attributes are then interpolated as follows:

$$\mathbf{a}(u,v) \underset{[P]}{=} (1-s-t)\mathbf{a}_0^j + s\,\mathbf{a}_1^j + t\,\mathbf{a}_2^j, \tag{4}$$

where $\mathbf{a}_i^j$ are the attributes of triangle j at the vertices $i \in \{0,1,2\}$. These are fetched from the *domain shading cache* (DS$), shown in Figure 5, which operates in the same way as a traditional vertex cache. We have simulated a DS$ that holds the N most recently used domain vertices. The capacity N necessary for good reuse depends on many factors, including the ordering of triangles within the patch, its access patterns, and so on. Figure 9 presents an initial analysis, where we find that a modest value of $N = 64$ vertices is often sufficient, even at high tessellation rates. This is on par with the capacity of vertex caches commonly used.

Triangle Lookup In its most general form, the lookup function P can be implemented by traversing a 2D accelleration structure in (u,v)-space, such as a grid or quad tree, which is built once per rendered patch. While such a strategy always works, it is unnecessarily costly if the tessellator and interpolation unit are properly co-designed. We have looked at a few different cases.

With *uniform* tessellation, the triangle index j can – with knowledge of the tessellator's triangle output order and split diagonal – trivially be found by quantizing $\mathbf{u}$ to the tessellation grid, and inverting the space-filling curve along which triangles are output. This can in most cases be done using simple bit operations. We have implemented this for `OpenSubdiv`'s default output ordering with uniform tessellation. In Direct3D 11, both uniform and non-uniform tessellation are relatively easily supported, although the latter is slightly more involved. With the non-uniform pattern, each patch edge has its own tessellation factor, resulting in an interior uniform region and a border with stitched triangles. The interior is trivial, and for the border, we can locate the relevant section and do a few specialized 2D point-in-triangle tests.

Once the triangle j has been found, the point's parametric coordinates (u,v) are transformed to triangle barycentrics using the inverse of the affine transform in Equation 7 in Appendix A, at a cost of 4 multiply-and-add (MADD) operations.

4.5 Pixel Shader Execution

In our prototype implementation, multi-frequency shading is exposed through the function:

`shade2D(k,m,u,dudx,dudy)`

where k is the shader kernel, and m specifies the filtering mode. This is analogous to the `tex2Dgrad` instruction for texture sampling in HLSL. The above function may be executed from within any screen- or patch-space shader kernel. Once a shader kernel finishes execution, its results are stored in the shading cache and returned to the caller.

The value of $\mathbf{u}$ and its derivatives are supplied to the issuing shader as system-generated values, which may be modified to adapt the shading rate (c.f., Section 4.2) before calling out. The kernel itself uses an input/output declaration very similar to the pixel shader, where the input may be any subset of the domain shader attributes. These are automatically interpolated to the location $\mathbf{u}$, as described in Section 4.4. The input to the pixel shader unit (the bottom unit labeled PS in Figure 5) is thus a shading quad with pre-interpolated vertex attributes. Similar to previous work [Burns et al. 2010; Ragan-Kelley et al. 2011], we use finite differencing over the locally regular shading grids to approximate shader derivatives, e.g., for texture filtering. These will in our case be expressed as patch-space gradients, `dd`$\mathbf{u}$, analogous to `ddx`/`ddy` in screen-space shading methods. Within a shader kernel, any texture filtering mode may be used, e.g., anisotropic filtering.

To allow different shader kernels k to run simultaneously, we assume *bindless* shader resources. Instead of relying on a fixed set of resource bind slots for constant buffers, textures, samplers, and so on, the shader kernel is self-contained and accesses its resources through handles that refer to resource descriptors allocated in graphics memory. This is a step away from current graphics APIs, which run the same pixel shader for all geometry in a draw call. However, a flexible bindless execution model is a logical next step, as it has clear benefits also from a usability point of view.

4.6 Task Scheduling

A modern graphics processor has multiple physical shader cores, each running a large number of logical threads (*contexts*), in order to hide latencies due to memory stalls etc. The dedicated register file is a finite resource that effectively limits the number of simultaneous threads. For good utilization, it must thus be ensured that each execution core receives enough work to keep it busy. Consequently, the hardware must be able to handle a large number of simultaneous shading quads and use a good *load balancing* strategy for work distribution. Also, mechanisms must be in place to handle out-of-order retirement of shading quads, while ensuring a consistent ordering of the frame buffer updates from frame to frame.

All these things also hold true for our system. However, whereas the rasterizer is normally responsible for generating a steady stream of shading quads, the majority of the shading work in our system is generated at misses in the shading cache. Hence the total amount

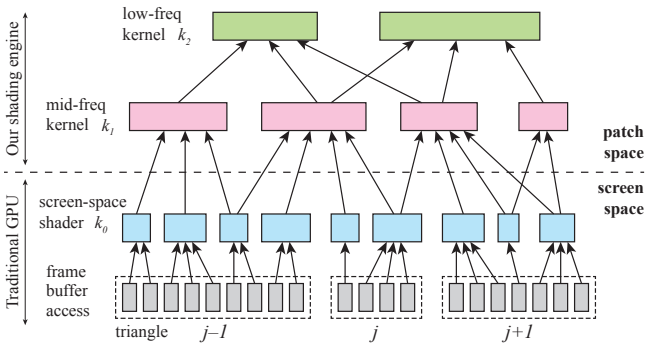


Figure 10: *Example of a task graph for multi-frequency shading, using both a screen-space pixel shader k_0, and two different patch-space shader kernels, k_1 and k_2, respectively, where k_2 runs at a lower frequency than k_1. The traditional real-time graphics pipeline implements the lower half of the graph, i.e., screen-space shading, while we add patch-space shading (upper half).*

of work is expected to be smaller than before, although the shading quads are generated in a more unpredictable fashion. Our architecture also supports hierarchical multi-frequency shading, which introduces dependencies between the shading quads.

Task Scheduling To handle this more difficult scheduling problem, we suggest a *distributed task-based scheduling* system. Each execution of a shader kernel for a particular shading quad is a *task*. Whenever a kernel issues a shading request that cannot be immediately answered, a new task is generated. In this case, the original kernel has a dependency on the newly added task to finish before it can proceed. Figure 10 shows an example of a possible task graph, with arrows depicting dependencies between tasks. To demonstrate that distributed scheduling is a viable strategy, we present results of scheduling simulations in Section 6.3.

The simulated architecture is specified by a number of execution cores, a fixed number of execution contexts, and a memory hierarchy statistically modeled by cache hit-ratios and latencies. Simulation of execution is performed cycle-by-cycle, taking into account memory stalls due to texture fetches. Memory accesses are assumed to occur with even probability throughout the execution of a task. Our architecture also adds the possibility to stall on other shading tasks. The main limiting factor can then be the number of stalled tasks waiting for other shading work to finish, since these consume valuable register space. It is important that tasks with many dependents finish early, and are given a high priority by the scheduling algorithm. When multiple tasks are ready for execution, i.e., they have no outstanding dependencies and have acquired contexts, they are thus first prioritized on the number of dependents, and thereafter according to an *oldest job first* policy.

While stalled, tasks still occupy contexts, which are returned only on task completion. To guarantee forward progress, we assume a preemption free architecture with a deadlock avoidance mechanism. By swapping out (spilling) execution contexts to RAM if available contexts are exhausted, new tasks can be launched. Another way is to use some variant of Banker's algorithm to partition available contexts to different shader kernel recursion depths (enforcing a hard upper limit). In the simplest case, one wants to ensure at least one free context per level of recursion. While such cases can indeed arise, none of our simulations required deadlock avoidance. Note that the user is in either case responsible for avoiding cyclic recursion. In the worst case, existing timeout mechanisms may have to interrupt and kill a faulty application.

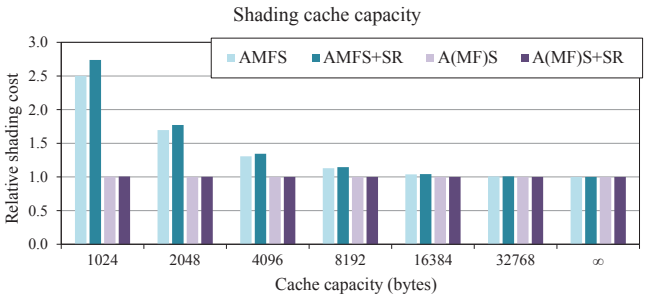


Figure 11: *Executing shading at a single frequency, A(MF)S is very insensitive to cache capacity for static images. At multiple frequencies (AMFS), a larger cache is required for full reuse. Stochastic rendering (SR) has a different access pattern and is slightly more cache-demanding. We find that a 16 kB cache is usually sufficient for both non-stochastic and blurry rendering. For the simpler A(MF)S case, even a meager 512 bytes is sufficient. The* DRAGON *scene was used for these measurements, with 20×20 pixels defocus blur added to simulate the impact of SR.*

5 Implementation

For the evaluation of our architecture, we have implemented a software simulator of the graphics pipeline in Figure 5. The simulator exposes a stateless rendering API, and runs C++ shaders using SIMD instructions through an abstraction similar to Microsoft's HLSL shading language. Internally, we build on `OpenSubdiv` for Catmull-Clark subdivision [Catmull and Clark 1978; Nießner et al. 2012], and `OpenImageIO` for texture sampling and image I/O. These are commonly used libraries for these tasks. Our simulator can also load data exported from the standard Direct3D 11 pipeline, including hardware tessellation.

The simulator is internally multi-threaded, but fine-grained task scheduling of the shading work is evaluated in a separate, cycle-based simulator. The rasterizer and shader execution is currently implemented with 8-wide SIMD instructions using the Intel AVX instruction set, and we thus shade up to two quads together. The system includes both a traditional hierarchical 2D rasterizer to simulate current graphics processors, and a stochastic 5D rasterizer [Akenine-Möller et al. 2007] for motion/defocus blur.

6 Results

In this section, we present results for several different important use cases. Our method is compared against *quad-fragment merging* (QFM) [Fatahalian et al. 2010], *decoupled sampling* (JRK) [Ragan-Kelley et al. 2011], and *lazy object-space shading* (LOS) [Burns et al. 2010]. Although we support a wider range of use cases, e.g., multi-frequency shading and adaptively chosen shading rates, these methods share the same goal of reducing the amount of pixel shading work in real-time graphics pipelines.

We use a shading cache with capacity for 512 entries, motivated by Figure 11. Using 16-bit half-float RGBA output, which is common today, our cache is backed by 16 kB of fast on-chip memory. The same size was used for JRK's shading memoization cache. We note that tessellation generally improves locality, as shading requests occur in the vicinity of each small triangle in shading space as they are rasterized. While multi-frequency shading increases the capacity requirements, the added shading components are usually evaluated at lower frequencies and thus do not occupy as many shading quads. For the QFM algorithm, we follow the authors' recommendation of using a 32-entry merge buffer, but to save memory we

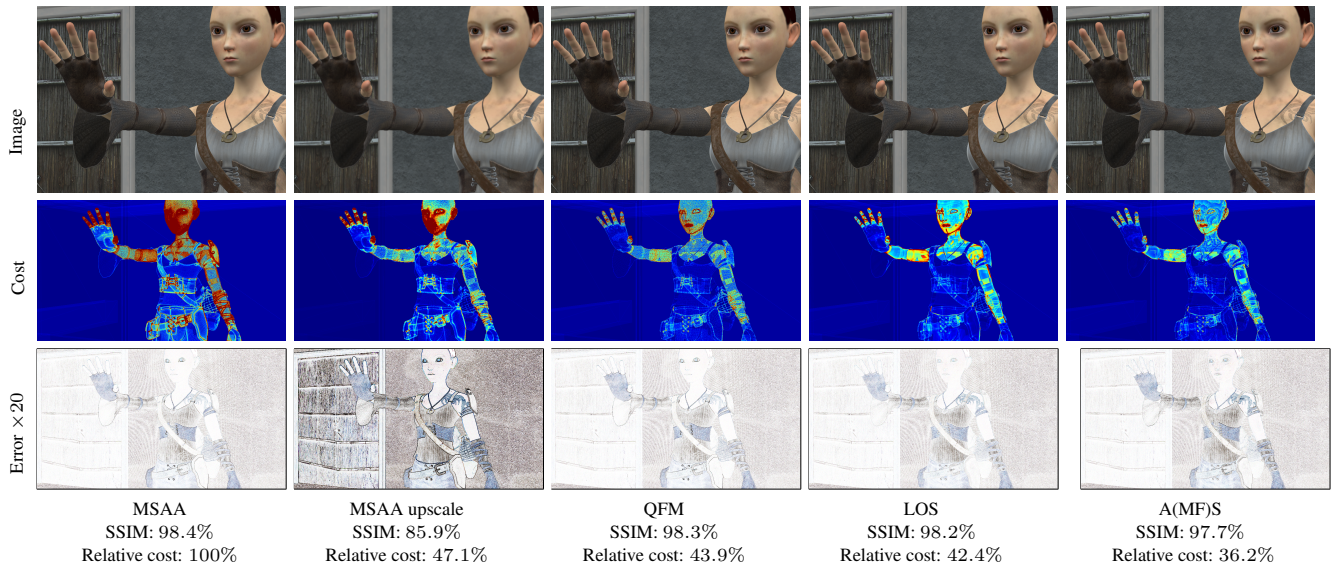


Figure 12: *Cost and quality for the* GIRL *scene. This scene contains many high-frequency materials, and the patch-space shading methods LOS and A(MF)S exhibit some slight, but visible, blurring. For completeness, an upscaled half-resolution MSAA rendering is also shown, exhibiting a significant loss of quality, while still being more expensive. This is a commonly used method for shading reduction today.*

store barycentric coordinates instead of pre-interpolated vertex attributes in each quad fragment. This results in a storage requirement of around 14 kB, comparable to the size of our shading cache. For LOS, we have chosen not to limit its memory usage.

To demonstrate realistic results using subdivision surfaces [Catmull and Clark 1978], we include two scenes (GIRL and DRAGON) based on assets from the *Sintel* short film. The geometry and shading networks were exported from the open-source 3D software *Blender*, and played back in our simulator. We also show one example (SUBD11) of an animated scene using Direct3D 11 tessellation of bicubic Bézier patches. This scene contains both non-tessellated geometry and quadrilateral patches. We also include two simple test scenes, BIGGUY and WOOD, in order to illustrate varying tessellation levels and distribution effects.

6.1 A(MF)S: Per-Pixel Shading

Shading Rates We start by looking at the common use case of shading approximately *once per pixel*, and compare our architecture against traditional MSAA [Akeley 1993] used in today's GPUs. At increasing tessellation levels, the benefits of MSAA are quickly lost as shading is not reused between primitives. In contrast, our technique scales very well and provides nearly constant shading rates irrespective of the geometric detail. Figure 4 shows BIGGUY with textured displacement mapping at various level of subdivision. It is clear that MSAA is not particularly efficient at finer tessellations. For example, at subdivision level 4, MSAA performs about 3× as many computations as our algorithm, A(MF)S. This factor increases to about 12× at level 6.

Figure 12 shows a more realistic case (GIRL), where the shading cost in terms of number of executed shader instructions is compared against our baseline of MSAA for each algorithm. As a simple approach to reducing shading, we also include an image rendered at half resolution and bilinearly upscaled. This is a common technique in today's game engines, despite the significant loss of image quality (see Figure 15 for another example). The QFM and LOS architectures neatly address the problem of reducing shading, and reach close to the same low cost as our algorithm *without* multi-frequency shading – A(MF)S. However, those methods only support a subset

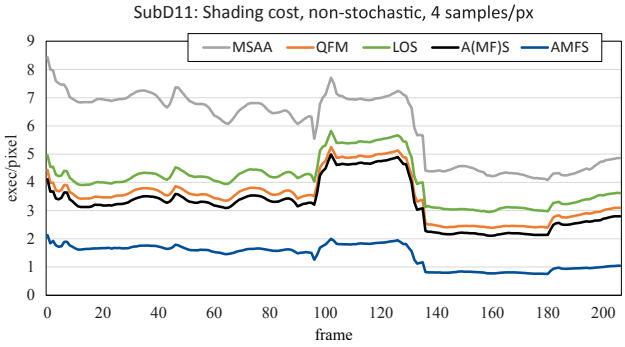


Figure 13: *The average #pixel shader executions per pixel (lower is better) throughout the* SUBD11 *animation, which was rendered using standard rasterization. Our algorithm A(MF)S, shading once per pixel, consistently outperforms the others. By further enabling multi-frequency shading, AMFS, the cost is drastically reduced.*

of the flexibility of our architecture, as we will see below. Note that we did not include JRK here, as its cost and quality are identical to MSAA for non-stochastic rendering, since each shading sample is mapped to a static pixel-sized and pixel-aligned shading grid.

Figure 13 shows that the shading costs using our algorithms are temporally stable, and consistently lower than the competing algorithms. The measurement was done by rendering the SUBD11 animation (of which one frame is shown in Figure 18) unmodified from the Direct3D SDK, using uniform tessellation to an average triangle area of 17.6 pixels over time.

Image Quality The different methods give rise to errors with different characteristics, although it should be noted that it is difficult to define a true reference. Even super-sampled shading is still only a sampled representation of the continuous surfaces, and relies on the usual approximations in texture filtering. To give an indication of the visual quality, we have computed the *structural similarity index* (SSIM) [Wang et al. 2004] of each rendered image, compared to a very densely sampled reference. See Figure 12, 14, and 18.

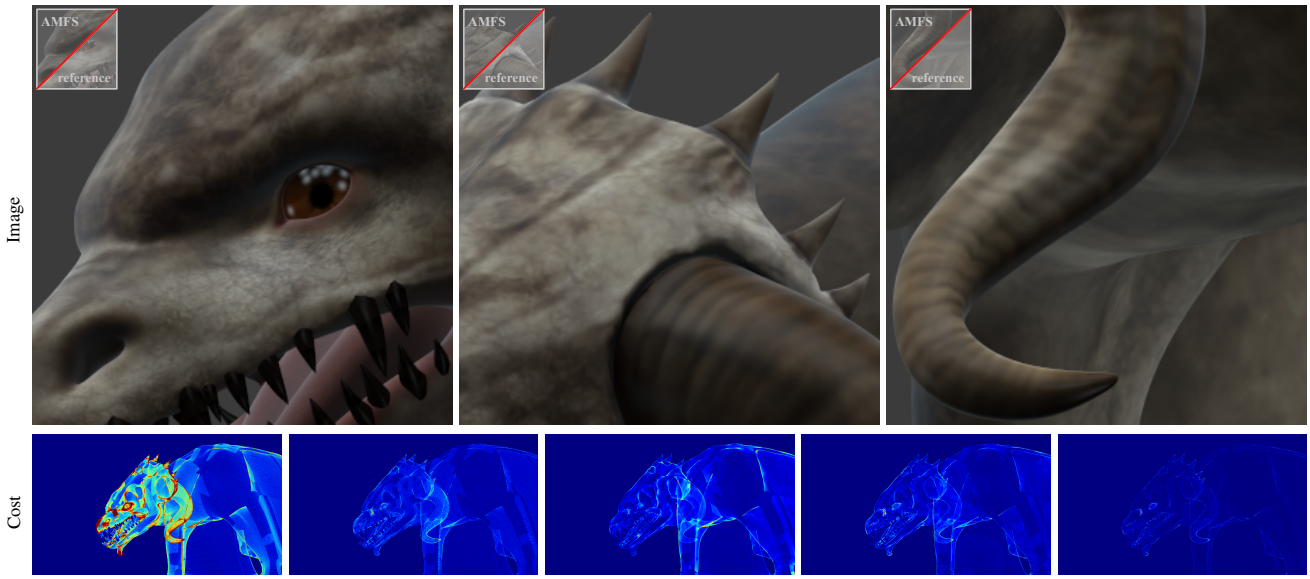


Figure 14: *Cost and quality evaluation for* DRAGON. *Costs are measured in number of scalar instructions and texture lookups, and errors against a high-quality super-sampled reference. MSAA suffers from inefficient shading due to many small triangles. QFM reduces most of the inefficiencies of MSAA, but is slightly hindered by the limited capacity of the merge buffer. LOS over-shades highly curved patches in order to maintain image quality at the lowest-frequency locations, while our algorithm, A(MF)S, locally selects a suitable shading frequency even in highly curved regions. Mainly due to its insensitivity to cache capacity, the overall shading cost of A(MF)S is lower than QFM. By leveraging our capability of splitting shader computations into several frequencies, the overall cost of AMFS is significantly lower than all the compared algorithms. Qualitywise, all algorithms perform very well, with SSIM scores above 99.8%.*

The screen-space shading methods (MSAA and QFM) may have an undue benefit here, as both the reference and those methods were computed using the 1×1-pixel box filter of many current GPUs. For patch-space shading (LOS and our methods), the shading samples are not pixel-aligned, and better resolve filters are motivated.

MSAA generally suffers from extrapolation artifacts when the shading point (i.e., pixel center) lies outside the primitive. This may show as stray pixels with a distracting color or intensity. The problem is partially alleviated by centroid sampling, although that may introduce other errors due to inaccurate derivatives. QFM heuristically merges shading requests from adjacent primitives, and may thus introduce unexpected visual differences, although in practice, its quality is often close to that of MSAA. LOS gives results similar to ours, but their selection of a single per-patch shading rate calls for a more conservative choice of grid resolution. The shading may thus be locally slightly sharper (c.f., Figure 16). It should be noted that we had difficulties controlling their shading rate, as estimating it solely based on the patch corners often fails to produce good results. We thus apply their heuristic over a 5×5 grid per patch.

6.2 Multi-Frequency Shading

None of the compared methods support *multi-frequency* shading, which is a strong argument for our algorithm. The overarching goal is to compute low-frequency components of the pixel shading at a rate much lower than once per pixel, in order to drastically reduce the total shading cost at only an insignificant loss in image quality.

Figure 14 shows an example of multi-frequency shading using high-quality assets. In this example, the shading networks were partitioned into two or three components, to be shaded at rates of approximately once per pixel, $0.5^2/\text{px}$, and $0.25^2/\text{px}$, respectively.

Figure 15: *Small crop of* SUBD11, *rendered at 1920×1080 pixels resolution. Left: MSAA at half resolution and bicubic upscaled for display, yielding an effective cost of 1.85 shader executions/pixel. Center: AMFS at full resolution, but shading once per 2×2 pixels on average. Right: AMFS, computing surface color once per pixel on average, but lighting only once per 4×4 pixels on average.*

This was done manually, but with help of an automated statistical analysis of the mean and standard deviation of each partial result in the shading networks created by artists. In this scene, the low frequency shading was bilinearly filtered to avoid banding and includes, e.g., diffuse lighting, dim specular lighting, and diffuse color for smooth materials. The *average* cost/pixel in terms of number of shader instructions is reported. In this case, the total cost is only around 7.0% of that of MSAA, without introducing any perceivable differences in image quality at normal viewing distances.

Figure 15 shows another example, using two different strategies: uniform downsampling to 0.5^2/px, or computing the lighting only at an agressively low rate of 0.25^2 per pixel, on average. The latter approach was used also for the rightmost example in Figure 18.

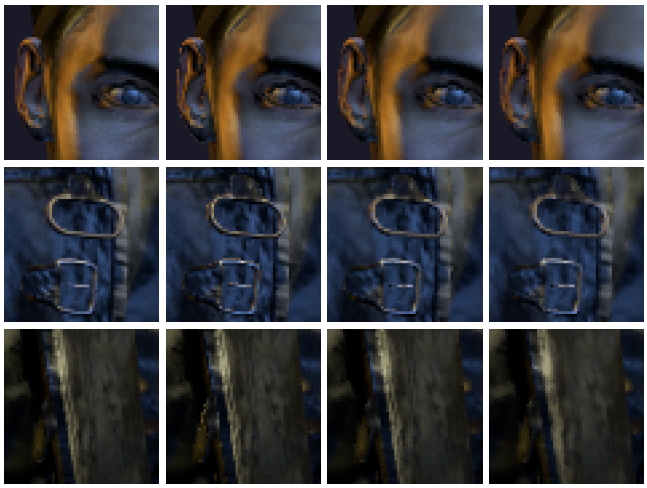


Figure 16: *Crops from* SUBD11. *MSAA and QFM shade in screen space and tend to produce extrapolation artifacts at object silhouettes (QFM is very close to MSAA, so omitted). LOS and A(MF)S operate in patch space and do not suffer from this problem. However, object-space methods can sometimes blur texture detail (c.f., bottom row) due to the misalignment of the shading and pixel grids.*

Although there are clear differences, the visual result is very useful considering its low cost. A game engine may, for example, use the shading rate as very simple and effective performance knob, in order to guarantee a constant frame rate. The accompanying video shows that the shading is also temporally stable.

6.3 Scheduling Simulations

To get an indication of the expected performance of a hardware implementation, we have performed scheduling simulations based on execution traces from our scenes. GPUs commonly have several levels in the texture cache hierarchy to reduce the average access time [Doggett 2012]. Coherent accesses associated with, e.g., trilinear mipmapping and anisotropic filtering, are particularly cache friendly. We simulate texture accesses using a statistical model with two levels of cache, backed by RAM, thus stalling the issuing thread for a varying amount of time. Texture fetches are here modeled with a hit-ratio/latency of $95\%/10$ cycles for L1, and $90\%/100$ cycles for L2, while RAM latency is 1,000 cycles.

Our simulated device has 48 cores, each executing one shading quad at a time in 4-wide SIMD. The more thread contexts that can execute per core, the less likely the cores are to stall due to memory latencies. Figure 17 shows simulated machine utilization and relative execution times for rendering the DRAGON scene with various numbers of thread contexts. Dependencies and memory stalls clearly impact the shorter shading tasks of the multi-frequency workload (AMFS) the most, requiring more contexts to reach a high utilization. Without dependencies, simply having twice the amount of contexts compared to cores is enough to reach 100% utilization for the other workloads. However, as expected, even at a moderate amount of thread contexts per core, AMFS clearly outperforms not only MSAA, but also A(MF)S, despite a relatively low utilization.

Energy efficiency is one of the most important design criteria for current and future graphics processors. As such, despite a lower shader core utilization, the much shorter total run times of our shading architecture should make it a substantially more energy-efficient alternative than previous methods. It should be noted that there are

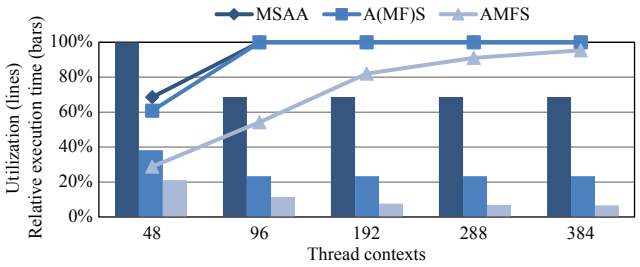


Figure 17: *Utilization (lines) and relative execution times (bars) of our algorithms compared to MSAA, as a function of number of thread contexts. Most importantly, our techniques significantly reduce the total execution times, even at lower utilizations.*

additional hardware costs associated with thread launch, scheduling, etc. However, there are also some positive effects not currently simulated, e.g., shading at lower frequencies means accessing textures at higher mipmap levels with better texture cache utilization and lower memory bandwidth as a result.

6.4 Distribution Effects

An important benefit of shading in a parametric space tied to the geometry, is that distribution effects such as motion and defocus blur can be much more efficiently rendered using stochastic rasterization [Akenine-Möller et al. 2007]. Several authors have noted the inefficiency of shading in screen space [McGuire et al. 2010; Munkberg et al. 2011] when blur is added. The problem is amplified when *both* tessellation and stochastic rendering is used. Decoupled sampling (JRK) partially addresses the problem by shading in parametric space over each triangle. Our architecture has the additional benefit, similar to LOS, that it supports the combination of tessellation and stochastic rendering efficiently. Note that neither MSAA nor QFM were designed for distribution effects. However, we do include them in the comparison for completeness, and note that QFM does provide a small improvement over MSAA.

Figure 18 shows a comparison using both defocus and motion blur, where our algorithm provides for the lowest shading rates, both for static and stochastic rendering. Figure 19 shows a case of motion blur only. In the top row, it is clear that LOS suffers from its single per-patch shading rate, as it is locally over-shading along the silhouettes of the displaced surface. The same effect is present with blur. Note that both JRK and our algorithm can additionally be used together with per-triangle adaptive anisotropic shading (AAS) [Vaidyanathan et al. 2012] to further reduce the cost in the presence of blur, but our algorithm retains its relative advantage. For the LOS method, AAS would have to be implemented per patch, instead of per triangle, which is less effective.

7 Limitations and Future Work

We have presented a flexible architecture for pixel shading that supports a wide range of uses. However, there are many areas worthy of further investigation. First, it would be important to study the interaction between the screen-space pixel grid and the (unaligned) patch-space shading grids. This is a sampling and reconstruction problem, for which the current simple MSAA box reconstruction filter is unsuitable. We expect that much of the over-blurring compared to screen-space shading methods can be removed (c.f., Figure 16 for examples).

Second, shading is currently not reused between adjacent patches or over triangle meshes. Several options exist, although they may add significant complexity. A parameterization that extends beyond sin-

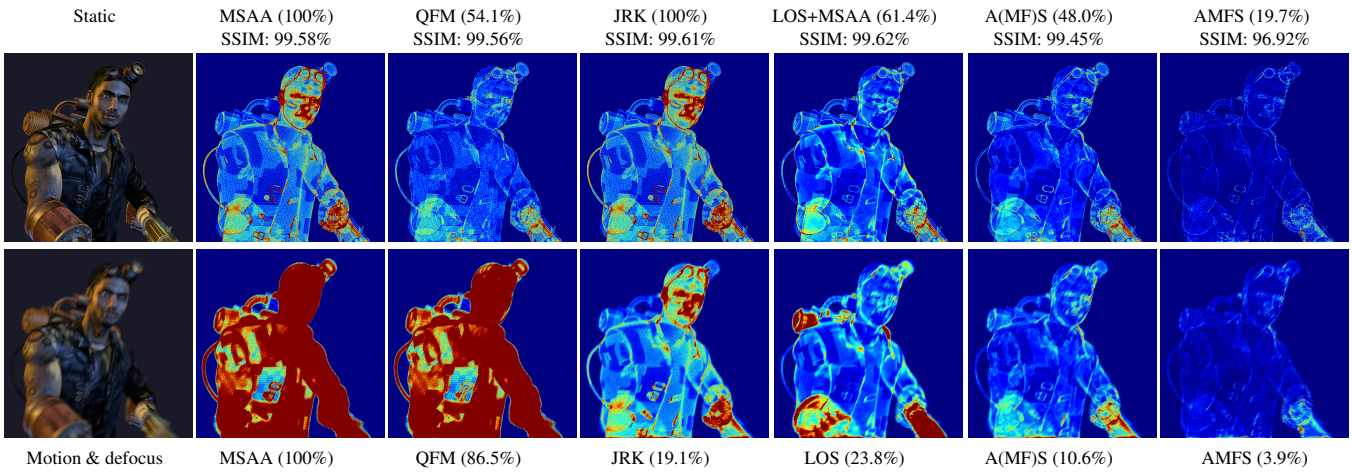


Figure 18: *The* SUBD11 *scene using Direct3D 11 tessellation to 128 triangles/patch, rendered at 16 spp without blur (top) and with motion/defocus blur (bottom). The heat maps show the average number of pixel shader executions in the range [0,16]. MSAA and QFM were not designed to handle blur and largely fails to reduce shading. Given this, it is intriguing to see that QFM still manages to reduce shading a bit. Decoupled sampling (JRK) operates in a stationary shading space, but it does not share shading between primitives. LOS, as well as our method, reuse shading within patches. Our A(MF)S algorithm, shading approximately once per pixel, achieves lower rates due to its adaptive shading grid computation, and scales very well with blur. We also include an example of multi-frequency shading (AMFS) on the right.*

gle patches or triangles is required, e.g., geometry images [Gu et al. 2002] or Ptex [Burley and Lacewell 2008]. If interpolation of per-vertex attributes is necessary, which it most often is, the hardware must be able to compute the mapping P (Section 4.4) based on the mesh connectivity and parametric location of patches/triangles. The geometry processing of the pipeline must also be more dynamic. There is currently no guarantee that an adjacent patch will reside in memory when shading is requested, so the geometry pipeline would have to be able to compute the tessellation on the fly.

As a next step, it would also be interesting to study methods for automatic decomposition of shaders into multi-frequency parts, in order to help developers make the best use of our system. The problem is related to shader simplification [Pellacini 2005; Sitthi-Amorn et al. 2011], which makes us optimistic that good solutions exist.

8 Conclusion

To reach a higher visual fidelity, while staying within the power envelope of modern graphics devices, it is critical to reduce the cost of pixel shading. We achieve this goal by shading in parametric patch space, thereby largely decoupling the cost of pixel shading from the geometric complexity. This allows developers to add fine geometric detail where needed, without severely increasing the number of pixel shader executions. Our architecture also brings the possibility to locally or globally reduce the shading rates for computations that can be performed at lower frequencies than once per pixel or sample. This flexibility allows a smooth degradation of image quality at increased performance, something that is very desirable in order to keep a constant frame rate in real-time applications.

In conclusion, by reducing the cost of pixel shading with advanced rendering techniques such as subdivision surfaces and/or stochastic rasterization, we hope to significantly narrow the quality gap between offline rendering and real-time graphics. We envision that future hardware support for these features, in combination with a powerful system for shading reuse such as ours, will also help bring down the cost of content development. This is a crucial limiting factor in game studios today. Admittedly, we have only explored a few possible use cases, but in the hands of smart developers, the flexibility of AMFS would be a powerful tool.

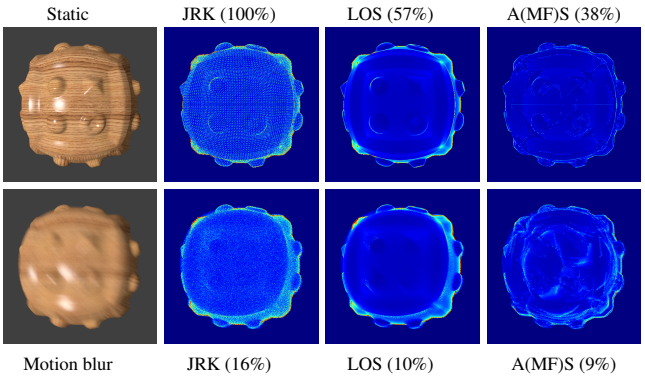


Figure 19: *A simple tessellated and displaced object (*WOOD*) with 12.0 pixels average triangle area, rendered with and without motion blur at 16 samples/pixel. The heat maps show the average number of pixel shader executions in the range [0,16].*

Acknowledgements

We thank Tom Piazza, David Blythe, and Charles Lingle for supporting this work, the anonymous reviewers for their valuable feedback, and the rest of Intel's Advanced Rendering Technology (ART) team. Jacob Munkberg also helped with video editing. The GIRL and DRAGON scenes are based on the *Sintel* short film by the Blender Foundation, for which we are very grateful. The SUBD and WOOD scenes use assets from the Microsoft DirectX SDK (June 2010), and BIGGUY was created by Bay Raitt. Tomas is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

References

AKELEY, K. 1993. RealityEngine Graphics. In *Proceedings of SIGGRAPH 93*, ACM, 109–116.

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, 7–16.

BURLEY, B., AND LACEWELL, D. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. In *Eurographics Symposium on Rendering*, 1155–1164.

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A Lazy Object-Space Shading Architecture with Decoupled Sampling. In *High Performance Graphics*, 19–28.

CATMULL, E., AND CLARK, J. 1978. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer-Aided Design, 10*, 6, 350–355.

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

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

DOGGETT, M. 2012. Texture Caches. *IEEE Micro, 32*, 3, 136–141.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing Shading on GPUs using Quad-Fragment Merging. *ACM Transactions on Graphics, 29*, 4, 67:1–67:8.

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

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry Images. In *Proceedings of SIGGRAPH 2002*, ACM, 355–361.

GUENTER, B., FINCH, M., DRUCKER, S., TAN, D., AND SNYDER, J. 2012. Foveated 3D Graphics. *ACM Transactions on Graphics, 31*, 6, 164:1–164:10.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2007. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics, 26*, 3, 92:1–92:10.

HECKBERT, P. S., AND MORETON, H. P. 1991. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, 101–111.

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled Deferred Shading for Hardware Rasterization. In *Symposium on Interactive 3D Graphics and Games*, 143–150.

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics*, 173–182.

MUNKBERG, J., CLARBERG, P., HASSELGREN, J., TOTH, R., SUGIHARA, M., AND AKENINE-MÖLLER, T. 2011. Hierarchical Stochastic Motion Blur Rasterization. In *High Performance Graphics*, 107–118.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Transactions on Graphics, 31*, 1, 6:1–6:11.

PELLACINI, F. 2005. User-Configurable Automatic Shader Simplification. *ACM Transactions on Graphics, 24*, 3, 445–452.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics, 30*, 3, 17:1–17:17.

SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. 2011. Genetic Programming for Shader Simplification. *ACM Transactions on Graphics, 30*, 6, 152:1–152:12.

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

VAIDYANATHAN, K., TOTH, R., SALVI, M., BOULOS, S., AND LEFOHN, A. 2012. Adaptive Image Space Shading for Motion and Defocus Blur. In *High Performance Graphics*, 13–21.

WANG, Z., BOVIK, A., SHEIKH, H., AND SIMONCELLI, E. 2004. Image Quality Assessment: from Error Visibility to Structural Similarity. *IEEE Transactions on Image Proc., 13*, 4, 600–612.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A Ray Tracing Solution for Diffuse Interreflection. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, ACM, 85–92.

A Parametric Patch Space

For each shading request, the patch-parametric coordinates $\mathbf{u} = (u, v)$ and their screen-space derivatives are computed analytically. Let $(1-s-t, s, t)$ be barycentric coordinates on a triangle. Perspective-correct interpolation of $\mathbf{s} = (s, t)$ at a screen-space position (x, y) is expressed as [Heckbert and Moreton 1991]:

$$\mathbf{s}(x,y) = \frac{\frac{\mathbf{s}}{w}(x,y)}{\frac{1}{w}(x,y)} = \frac{a_{\mathbf{s}}x + b_{\mathbf{s}}y + c_{\mathbf{s}}}{a_1x + b_1y + c_1}, \tag{5}$$

where the interpolation coefficients (a_i, b_i, c_i) are constant over the triangle and are computed in the setup (for non-stochastic rasterization). The partial derivatives of $\mathbf{s}$ with respect to screen-space position follows from differentiation of Equation 5:

$$\mathbf{s}_x = \frac{\partial \mathbf{s}}{\partial x}(x,y) = \ldots = (a_{\mathbf{s}} - a_1\mathbf{s}(x,y))w(x,y), \tag{6}$$

with a similar expression for $\mathbf{s}_y$, and using $w = 1/\frac{1}{w}$. Note that the rasterizer already computes the hit point $\mathbf{s}$ (and hence w), so the added cost is one MADD operation per partial derivative (four in total). Given the patch-parametric coordinates (i.e., the domain points) of the current triangle's vertices, $\mathbf{u}_0$, $\mathbf{u}_1$, and $\mathbf{u}_2$, the transform from triangle to patch space is an affine 2×3 matrix:

$$M = \begin{bmatrix} u_1-u_0 & u_2-u_0 & u_0 \\ v_1-v_0 & v_2-v_0 & v_0 \end{bmatrix}, \tag{7}$$

where the shading point is transformed as $\mathbf{u} = M \cdot (\mathbf{s}, 1)^T$, and the derivatives are transformed as vectors, e.g., $\mathbf{u}_x = M \cdot (\mathbf{s}_x, 0)^T$. In the general case, these three transforms carry a total cost of 12 MADDs. For certain tessellation schemes, e.g., uniform tessellation, faster special cases may be implemented. When stochastic rasterization is used, the triangle vertices are functions of the time/lens position. In this case, we choose to compute exact derivatives at each sample's location in 5D space.

B Shading Grid Resolution

First, the AABB of the parallelogram spanned by $\mathbf{u}_x$ and $\mathbf{u}_y$ in parametric space is computed, which has extents $\mathbf{b} = (b_u, b_v) = \max(\mathbf{0}, \mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_x+\mathbf{u}_y) - \min(\mathbf{0}, \mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_x+\mathbf{u}_y)$, and area $A_{\text{box}} = b_u b_v$. Based on this, an (unquantized) target grid resolution of $\mathbf{r} = (r_u, r_v)$ grid points is found as:

$$\mathbf{r} = \sqrt{c\alpha}\,(1/b_u, 1/b_v)\,, \qquad \text{where } \alpha = A_{\text{box}}/A_{\text{pixel}}. \tag{8}$$

Note that a correction factor c is included to locally increase the grid resolution for difficult cases. With $c = 1$, each of the $r_u \times r_v$ grid cells would have an area exactly equal to A_{pixel} (i.e., pixel rate shading). Heuristically, we define $c = \min(\max(\alpha/2, 1), N)$, where N is the multisampling rate. The rationale is that correction is only applied when the distortion is above some threshold, $\alpha > 2$, while being limited to not shade more often than the visibility rate. Finally, (r_u, r_v) is quantized by rounding each dimension up/down to its nearest power-of-twos, and selecting the combination $\mathbf{n} = (n_u, n_v)$ that yields an area as close to $1/r_u r_v$ as possible.