Texel Shading

K. E. Hillesland and J. C. Yang

Advanced Micro Devices, Inc.

(a) Cabin scene. 2.3x speedup. (b) Porch scene. 3.9x speedup. (c) Harvester. 4.1x speedup.

Figure 1: Speedups for texel shading over fragment shading with 30 shadowed lights using shading rate reductions enabled by our method. Upper left is standard fragment shading, bottom right is with texel shading.

Abstract

We have developed a texture space shading system built on modern graphics hardware. It begins with a conventional rasterization stage, but records texel accesses as shading work rather than running a shade per pixel. Shading is performed by a separate compute stage, storing the results in a texture. As a baseline, the texels correspond to those required for mipmapped texturing. A final stage collects data from the texture. Storing results in a texture allows for reuse across frames. We also show how adapting shade rate to less than once per pixel further increases performance. We vary shading load to show when these techniques provide a performance win, with up to 4.1x speedup in our experiments at shading times less than 4 ms.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism —Color, shading, shadowing, and texture

1. Introduction

There has been a recent jump in pixel densities and target frame rates. Examples include high density mobile and tablet displays, emerging 4k and higher monitors, as well as head mounted displays running at 90Hz and above. With the shading largely being performed at a frequency of at least once per pixel and once per frame, this has a direct impact on shading cost.

Shading cost can be greatly reduced by decoupling the shading rate from pixel density, triangle density, and frame rate. Here we propose a method to accomplish all three on current hardware. Instead of shading fragments, we shade texels in a mipmap hierarchy. Since our shading is texel-centric, but different from previous methods that rasterize in texture space, we refer to our method as *texel shading*.

This object-space parameterization allows us to reuse shades from previous frames, while updating visibility at the full frame rate. We use the mipmap hierarchy to decouple and control shading rate relative to pixel rate by being able to select the mipmap level for shaded texels on a per-fragment basis. Finally, we are able to eliminate redundant texel shading requests, effectively removing redundant shades invoked by more than one primitive in a texel.

2. Previous Work

One way to reduce pixel shader load is to relax the assumption that a shade must be performed for each pixel, and instead shade for every integer multiple of the base rate. Games do this in software now [TTV13], although constrained to choose a fixed multiple up front. There are also hardware solutions that choose more dynamically [HGF14, VST*14], but are constrained within a primitive.

Another approach is to reuse shades from previous frames [NSL*07]. These methods work in screen space, presenting difficulties with respect to occlusion and resampling.

There are also methods to decouple shading rate from the ge-

DOI: 10.2312/egsh.20161018

ometry [BFM10, CTH*14], although they are focused on higher-order surfaces and constrain reuse to within a patch, and they are interested in hardware solutions that reuse shades that do not span multiple frames.

There has been quite a lot of interest in decoupled sampling [BFM10, RKLC*11] for the purpose of accelerating stochastic effects, such as depth of field and motion blur. The decoupling is intended to accelerate renderings of individual frames, not to amortize over multiple frames. However, there are two methods that are particularly relevant to ours [LD12, AHTAM14] and we discuss the relationship in further detail in Section 3.

We shade in texture space rather than in screen space. In the past, this has been done by rasterizing in texture space [CH02, AHTAM14], which presents some difficulties with respect to choosing the right resolution, and taking advantage of early depth tests. Adaptive texture space shading (ATSS) [AHTAM14] alleviates the resolution issue by making a resolution choice for each primitive.

3. Method

Our system has three stages. *Shade Queueing* is a geometry pass that identifies a tile of texels required for shading, consulting a specialized software cache before adding work to the queue. *Shade Evaluation* is a compute shader stage that fetches vertex attributes and writes shaded results to a *Result Texture*. *Shade Gathering* is a second and final geometry pass that collects the shaded results.

These stages are very similar to those of decoupled deferred shading [LD12]. Although they mention the possibility of working in a texture space shading domain, we found that our texture space approach led to different choices in design of some parts, particularly the caching system and its use across multiple frames.

One requirement of our method is that each mesh has an associated object-space parameterization. This parameterization must be unique such that there is no overlap in texture space. Second, this parameterization is assumed to be static. This is quite normative, but we state it explicitly, since this property is important for our method.

Triangle Index Texture. As a preprocess, we generate a texture map that stores a triangle index in each texel. This provides a mapping from a texture coordinate to a triangle, which is used in the shade evaluation stage (a compute shader stage). The map must contain an entry for each texel that is touched by a triangle in the sense of conservative rasterization [AMA05]. There is one complete mipmap chain for each unique texture space.

Shade Queuing. The first stage is a geometry pass to generate the shading work required to render the current frame, which may be skipped if we decide we can reuse shading from a previous frame.

The unit of work is a texel in the appropriate mipmap level. Selecting the appropriate mipmap level keeps the shade rate at approximately the same shading rate as fragment shading, although we can choose other rates to reduce shading load. We make this selection on a per-fragment basis, which is novel in the sense that we are the first to present a method that makes this selection in real-time on current hardware, whereas previous work has made this selection on a per-primitive basis [AHTAM14], at best, or through special hardware in the context of higher-order surface tessellation [CTH*14].

Once the mipmap level is computed, we generate four pieces of work, one for each tap of a bilinear filter. In the case of standard trilinear filtering, it would be eight pieces of work, and more for anisotropic filtering. However the texels identified will generally be shared with neighboring fragments. To reduce this redundancy, we use a caching scheme.

Algorithm 1 Generating shade work in 8x8 tiles.

```
1: ShadeWork = ∅
2: for each texel (i,j) at the chosen miplevel L do
3:     tile (i', j', L') = (i/8, j/8, L-3)
4:     if CacheAge(i',j',L') ≥ threshold then
5:         w = NewWorkItem(i',j',L');
6:         ShadeWork.Append(w);
7:     end if
8: end for
```

Caching. The caching system tracks when a particular texel was last shaded. If it was not shaded within the current frame, then it is added as a work item. We can actually change the shade rate by changing the threshold, so that if a texel was shaded within the last two frames, for example, we do not add it to the shade list, and instead use the older shade value.

The cache is simply a lower resolution mipmap chain, which we call the *cache texture*. Each texel in the cache texture corresponds to an 8x8 tile of texels at the target resolution for shading. Stored in each texel is a frame number. Before adding a texel of work, we compute which tile it is in and check to see if the frame number in the cache texture is recent enough that it does not need to be added as new work. There is one cache texture for each unique texture space; and because we need this information across frames, it must be stored in persistent memory. This differs from pervious work (Liktor et al [LD12] being the closest) which involved more global atomic contention, or a tradeoffs associated with tiling and limited local memory, and did not consider caching across frames.

Shade Evaluation. Shading is performed in a compute shader. A thread group shades an 8x8 tile of texels, which constitutes a work item, as well as the granularity at which the cache system operates.

Each thread in an 8x8 thread group shades a texel. Since this is compute rather than a rasterization pipeline, we need to interpolate any necessary vertex data in the compute shader itself. The vertices we need are those of the triangle covering this particular texel in texture space. The interpolation weights come from the barycentric value of the texel with respect to those vertices. We can then interpolate the per-vertex values we need, whether it be world space position, normal, or other.

This is where the triangle index texture comes into play. The output from that process was a mipmap pyramid storing an assigned triangle index for each texel. In the compute shader, we can use this to retrieve the three triangle vertices we need for interpolation.

Once we have the interpolated values from the vertices, the shad-

ing progresses in a manner very similar to a conventional fragment or pixel shader.

Not all shading needs to be performed at this stage. Some operations may be deferred to the next stage as part of a fragment shader.

Shade Gathering. The last stage occurs during a second geometry pass. Now we retrieve results from the shade evaluation stage from a texture. As long as we are consistent with the first geometry pass in our computation of the needed texels, we are guaranteed to have all the data we need stored in the shading result texture. For this reason, we compute the mipmap level in an identical way to the first geometry pass where the necessary texels were identified. The texels are fetched through bilinear texture filtering, giving us a filtered result of four shades. Additional shading operations may also be invoked here as part of the fragment shader.

4. Results

We evaluate our method by comparing against conventional fragment shading in terms of performance and error, first with and without MSAA, and then after reducing the shading load by reusing shading results from previous frames, and dynamically varying the spatial shading rate. We do this for the three setups shown in Figure 1.

For evaluation, we run both fragment shading and texel shading on a Radeon R9 290x at 1920x1080 resolution with 8x MSAA. Timing comparisons are for all three stages of texel shading against the single pass of a fragment shading (forward rendered).

Since D3D11 does not allow scattered writes to different levels of a mipmapped texture, we lay our mipmap levels out in a single 2D texture (4k x 6k), and use offset and scaling to write to the various levels as needed. In practice, choosing a 4k x 4k result texture resolution translates to 4k x 6k result and triangle index texture, and a (512 x 768) cache texture for each unique parameterization, in order to enable reuse across frames. The result texture has four 32 bit floating point channels, and the other two have one 32 bit channel.

For shader load, we scale the number of lights. Our testing was with all lighting computation in the shade evaluation phase, with just a final albedo texture modulation in the shade gathering stage.

We measured performance for three scenes, shown in Figure 1, each with a different camera path. The scenes include three texel-shaded models: a creature (34k triangles, skinned), a cabin (11k triangles), and a harvester (2.5k triangles). We also include ground geometry and a skybox, which are rendered conventionally.

The goal is to reduce shading rate. However, our method replaces a single geometry pass with two geometry passes and a compute shading stage, as well as additional cost throughout the system. The y-intercept of the graphs in Figure 2 give us a measure of this overhead. Subtracting the fragment shading cost from the texel shading cost gives us a total overhead measurement of 0.86 ms for the cabin scene, 0.55 ms in the porch scene, and 0.59 ms in the harvester scene.

Our shading reductions will be offset by two factors that increase shading rate. The first is the mapping from pixels to texels. We make a mipmap level selection for each pixel, and in our implementation, we round up to the higher mipmap level resolution, which is roughly expected to be greater than required for per-pixel shading. The second source of shading increase is in the caching system. For our chosen caching approach, it is overshading caused by the 8x8 tile size. In the end, we find that these two factors result in an increase of almost 2x over conventional fragment shading in some cases.

The creature is slightly faster in isolation with texel shading than with fragment shading at 30 lights (3.5 ms vs 3.9 ms). To further illustrate this effect, we amplified the triangle count of the creature using DX11 hardware tessellation. When we evaluate the highly tessellated version, the small triangle problem becomes much more severe for fragment shading, and the gap widens substantially (10 ms vs 54 ms).

Temporal Shading Reuse. In conventional rendering, every fragment is shaded every frame. In this section, we give results obtained by relaxing this restriction temporally. Rather than computing a full lighting solution for each texel in each frame, we seek to reuse results from previous frames. We choose a value N, reusing shades that are no more than N frames old. However, the model is still rasterized every frame according to camera changes and animation.

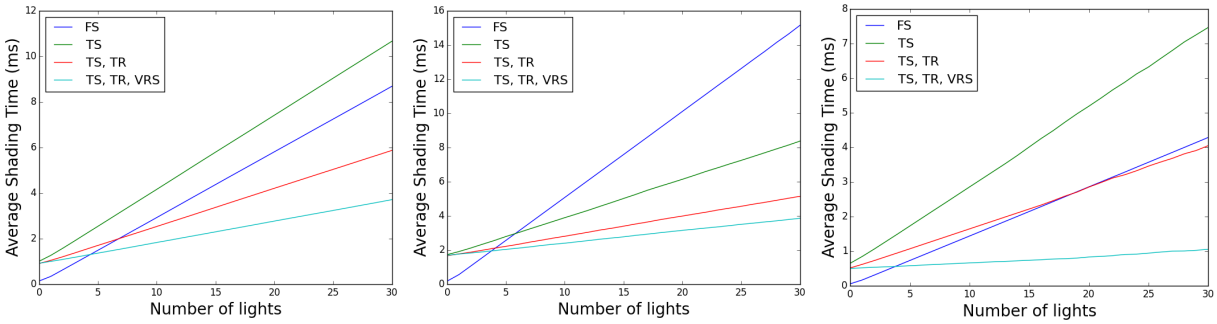
Error from this method is most visible in the porch scene on the surface of the character. It occurs when there is a rapid shift in lighting in an area where 8x8 tiles are not updated on the same frame. The error is negligible at $N = 1$ at 60 Hz, still fairly subtle at $N = 2$, but obvious at $N = 3$. Figure 3 shows an example of the artifact induced by this method when $N = 6$. How much shading reuse is possible without noticeable artifacts will depend on the scene, lighting, and how fast objects move or deform, but we found results to be quite good at 60 Hz when $N = 1$.

Figure 2 shows performance results when $N = 1$. Texel shading becomes faster than fragment shading for all scenes, but at different points. In general, the choice of N does not need to be the same for each object, or even each tile.

Variable Rate Shading. We are already decoupling shading rate from fragment rate. We can change the shading rate on a per-fragment basis by simply biasing the mipmap level choice. Our goal is to show that texel shading enables dynamic shading rate selection.

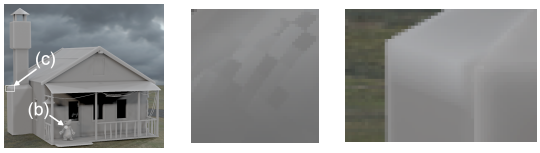
To demonstrate, we have implemented a simple heuristic and show measurable performance gains. We shift the mipmap level according to the change in the normal within a triangle, thus choosing lower shading rates in flatter areas.

This is a fairly brute force approach to dynamic shading rate control, but our results show substantial speedup without much loss in quality. We choose different bias scales on a per-object basis. For the creature and the harvester, the bias is 2, whereas the cabin has a bias scale of 1. The highest speedup is for the harvester, where we saw a total performance increase of 4.1x with at least 99% SSIM and 47 dB PSNR.



(a) Cabin Scene (b) Porch Scene (c) Harvester Scene

Figure 2: Shading time cost as a function of light count for fragment shading (FS), texel shading (TS), with temporal reuse (TR) and variable rate shading (VRS).



(a) (b) (c)

Figure 3: Here we see the kind of errors our method produces when pushed too far. (a) We show only the lighting term to make the error more visible. (b) Error on the creature's shoulder when shades are kept for up to 67 ms (4 frames at 60 Hz). (c) Error when we force bias of the mip level by two levels.

5. Conclusions and Future Work

We have presented a method that moves shading cost from a per-fragment to a per-texel basis on current graphics hardware. We show performance improvements in three ways. First, we show some improvement for the "small triangle problem". Second, we reuse shading results from previous frames. Third, we enable dynamic spatial shading rate choices, for further speedups. In total, we see shading speedups of up to 4.1x with very little error at under 4 ms. However, the shading cost must be high enough to justify the additional cost of the method, as well as a trade-off in terms of error introduced by reducing the shading rate, and additional memory cost.

For future work, we'd like to look at alleviating memory pressure through sparse texture representation. We'd like to take advantage of working in object space. Finally, we expect this work to be relevant to client-server rendering models.

Acknowledgements. Our thanks to Chris Brennan and Layla Mah for helpful discussions and advice.

References

[AHTAM14] ANDERSSON M., HASSELGREN J., TOTH R., AKENINE-MÖLLER T.: Adaptive texture space shading for stochastic rendering. *Computer Graphics Forum (Proceedings of Eurographics 2013) 33*, 2 (2014), 10. 2

[AMA05] AKENINE-MÖLLER T., AILA T.: Conservative and tiled rasterization using a modified triangle set-up. *Journal of Graphics, GPU, and Game Tools 10*, 3 (2005), 1–8. 2

[BFM10] BURNS C. A., FATAHALIAN K., MARK W. R.: A lazy object-space shading architecture with decoupled sampling. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 19–28. 2

[CH02] CARR N. A., HART J. C.: Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph. 21*, 2 (Apr. 2002), 106–131. 2

[CTH*14] CLARBERG P., TOTH R., HASSELGREN J., NILSSON J., AKENINE-MÖLLER T.: AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014) 33*, 4 (2014), 141:1–141:12. 2

[HGF14] HE Y., GU Y., FATAHALIAN K.: Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Trans. Graph. 33*, 4 (July 2014), 142:1–142:12. 1

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 143–150. 2

[NSL*07] NEHAB D., SANDER P. V., LAWRENCE J., TATARCHUK N., ISIDORO J. R.: Accelerating real-time shading with reverse reprojection caching. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), GH '07, Eurographics Association, pp. 25–35. 1

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for graphics pipelines. *ACM Trans. Graph. 30*, 3 (May 2011), 17:1–17:17. 2

[TTV13] TATARCHUK N., TCHOU C., VENZON J.: Destiny: From mythic science fiction to rendering in real-time. In *ACM SIGGRAPH 2013 Courses: Advances in Real-time Rendering in Games Part I* (New York, NY, USA, 2013), SIGGRAPH '13, ACM, pp. 12:1–12:1. 1

[VST*14] VAIDYANATHAN K., SALVI M., TOTH R., FOLEY T., AKENINE-MÖLLER T., NILSSON J., MUNKBERG J., HASSELGREN J., SUGIHARA M., CLARBERG P., JANCZAK T., LEFOHN A.: Coarse Pixel Shading. In *High Performance Graphics* (2014), pp. 9–18. 1