Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

An Improved Shading Cache for Modern GPUs

Pitchaya Sitthi-amorn[1] Jason Lawrence[1] Lei Yang[2] Pedro V. Sander[2] Diego Nehab[3]

[1]University of Virginia [2]Hong Kong University of Science and Technology [3]Microsoft Research

Abstract

Several recently proposed techniques based on the principle of data reprojection allow reusing shading information generated in one frame to accelerate the calculation of the shading in the following frame. This strategy can significantly reduce the average rendering cost for many important real-time effects at an acceptable level of approximation error. This paper analyzes the overhead associated with incorporating temporal data reprojection on modern GPUs. Based on this analysis, we propose an alternative algorithm to those previously described in the literature and measure its efficiency for multiple scenes and hardware platforms.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms I.3.1 [Computer Graphics]: Graphics processors

1. Introduction

The natural spatio-temporal coherence of animated image sequences has long been exploited to accelerate ray-based rendering systems. Recently, several methods based on the principle of data reprojection have been shown to provide similarly profitable error/performance trade-offs in real-time systems. For example, the method proposed by Nehab et al. [NSL*07] allows reusing shading information from the previous frame (usually the final pixel color) in the calculation of the shading in the current frame at some level of approximation error. They demonstrated impressive speedups for a number of effects including precomputed radiance transfer, procedural texture generation, depth-of-field, and stereoscopic rendering. Data reprojection has also proven useful in multi-view rendering architectures [HAM06] and for amortizing the cost of supersampling shadow maps over multiple frames [SJW07, NSL*07].

This paper examines the overhead of using data reprojection to accelerate interactive rendering applications executing on modern GPUs. Specifically, we consider the demands that reprojection places on the level of support for dynamic flow control and multiple render targets for the ATI 2900 and NVIDIA G80 architectures. Based on this analysis, we propose an alternative algorithm that is more efficient than existing methods over a wide range of cache loads and for different hardware platforms.

2. Data Reprojection in Real-Time Pixel Shading

This paper focuses on single-view real-time rendering applications deployed on modern ATI and NVIDIA GPUs. Figure 1 illustrates the basic idea of how data generated inside a fragment shader may be reused across consecutive frames through reprojection. At each frame, some value generated at the fragment level is stored in a viewport-sized off-screen buffer. We will call this buffer the *shading cache*. This is equivalent to the *real-time reprojection cache* described by Nehab et al. [NSI06, NSL*07], the *history buffer* described by Scherzer et al. [SJW07] and the *exact view* described by Hasselgren et al. [HAM06]. In the following frame, each fragment may access the data associated with its generating scene point (assuming it was visible in the previous frame) by correcting for the relative scene motion between frames. This strategy thus allows reusing data at *fixed surface locations* as opposed to fixed locations in the framebuffer.

Accelerating a pixel shader with a shading cache first requires identifying something worth reusing. This decision ultimately depends on the effect in question and relies on the ingenuity of a user. Hasselgren et al. [HAM06] recognize that view-independent values may be reused at nearby camera positions, but at the expense of some reconstruction error. Nehab et al. [NSL*07] offer more general guidelines, concluding that caching slowly-varying and computationally intensive calculations inside the shader provides

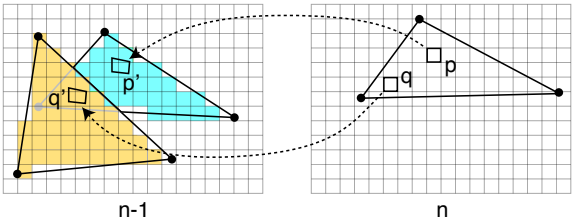


Figure 1: *Data reprojection at the fragment level.* ***Left:*** *At each frame, some value generated inside the fragment shader along with the scene depth are stored in the shading cache.* ***Right:*** *In the following frame, each fragment computes its position into this cache and uses the difference between the depth of its reprojected scene point and that stored in the cache to distinguish hits from misses. In this example, fragment p enjoys a cache hit whereas the scene location corresponding to fragment q was previously occluded and is not present in the cache.*

an ideal trade-off between speed improvement and shader fidelity. Scherzer et al. [SJW07] demonstrate that shadow map queries can be effectively reused across consecutive frame. Nehab et al. [NSL*07] discuss a similar approach for amortizing the cost of super-sampling a shadow map. In this paper, we put the question of what to cache aside and instead focus on the factors that determine the resulting performance gains, but pay particular attention to the consequences of caching and reusing *partial* shading data.

2.1. Fetching data from the cache

We will follow the techniques described by Nehab et al. [NSL*07] for computing the cache coordinates of a fragment, resolving hits and misses, and reconstructing the cache payload. Therefore, we will only briefly review these methods here (refer to Figure 1).

Computing cache coordinates: The homogeneous screen-space position of each vertex in the previous frame may be computed as

$$\mathbf{p}_{n-1} = \mathbf{R}_{n-1}\mathbf{M}_{n-1}\mathbf{P}_n,$$

where $\mathbf{P}_n$ is the object-space position of the vertex supplied by the application and $\mathbf{R}$ and $\mathbf{M}$ are camera perspective and modelview matrices, respectively. This calculation takes place at the vertex level and the homogeneous vector $\mathbf{p}_{n-1}$ is output as a per-vertex attribute and correctly interpolated across each face [HM91]. The final cache coordinates are obtained through a simple division within the fragment shader. The system proposed by Scherzer et al. [SJW07] instead performs this entire calculation inside the fragment shader. We recommend the computation previously described because it requires less pixel processing for presumably already pixel bound applications.

Resolving hits and misses: It's possible that a fragment's scene point was occluded in the previous frame and is thus not present in the shading cache. Differences in scene depth can be used to detect cache misses [NSL*07, SJW07]. This requires storing a single z-value along with the cache payload at every pixel. The depth of the reprojected scene point computed in the fragment shader ($\mathbf{p}_{n-1,z}/\mathbf{p}_{n-1,w}$ from above) is compared to the scene depth at that location in the previous frame. Whenever their difference is beyond some epsilon value [AS06], a cache miss is declared and the payload must be recomputed from scratch. Otherwise, a cache hit occurs and the payload may be reused in the calculation of the final pixel color. Note that this approach only allows reusing data for scene points nearest to the camera. For scenes with semi-transparent surfaces, for example, potential speed-ups are lost, although the accuracy of the shading is still ensured. Hasselgren et al. [HAM06] also point out this limitation.

Payload and scene depth reconstruction: Because a one-to-one correspondence does not exist between the fragments in the current and previous frames, it is necessary to reconstruct values at intermediate locations in the cache through the use of some type of reconstruction filter. Prior systems [NSL*07, SJW07] take advantage of efficient hardware support for bilinear texture filtering to reconstruct the cache payload and depth. Nehab et al. [NSL*07] note that a key benefit of this approach is to give more conservative hit/miss tests, favoring cache misses near depth boundaries. Hasselgren et al. [HAM06] project the endpoints of a filter kernel (e.g., tent or Gaussian) into the shading cache and integrate over this extent. Because most real-time applications exhibit very little scene motion between consecutive frames, we have found that more expensive reconstruction methods are not worth the additional effort compared to simple bilinear reconstruction.

Cache refresh: Although a value may remain available in the cache over many frames, it will eventually become stale and should be explicitly refreshed. This can happen due to changes in the shader inputs or from repeatedly resampling the cache, which tends to attenuate high-frequency spatial details in the signal. Nehab et al. [NSL*07] show that refreshing cached entries every Δn frames (on average) can be achieved by refreshing $1/\Delta n$ percent of every frame. They conclude that refreshing pixels along a random pattern uniformly distributed over the frame-buffer is preferable. The refresh pattern has important consequences in terms of rendering performance as we discuss in Section 4.

To support randomly distributed refresh patterns, a separate screen-sized texture $h(x,y)$ with pseudo-random numbers in the range $[0,\Delta n]$ and a global clock c are passed to the fragment shader. The global clock is a single unsigned integer which is incremented at each frame. A pixel is refreshed whenever the indicator variable b is zero

$$b = (h(x/k, y/k) + c) \mod \Delta n. \qquad (1)$$

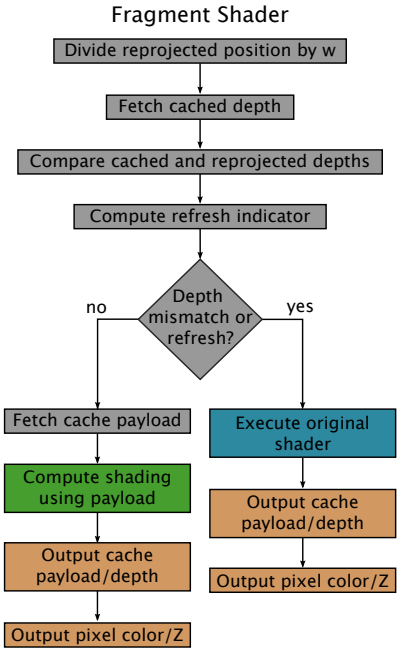


Figure 2: *Single pass algorithm for incorporating data reprojection into a fragment shader similar to that proposed by Nehab et al. [NSL*07].*

This causes refreshes to occur within $k \times k$ contiguous pixel regions as illustrated in Figure 5.

3. Three Algorithms

Figures 2, 3, and 4 illustrate three alternative algorithms of a shading cache. Each one allows caching a single intermediate floating point 3-tuple in the fragment shader. Therefore, the shading cache (which includes both the payload and scene depth) can be maintained in a single four-channel buffer. As noted by Scherzer et al. [SJW07], it is necessary to double-buffer the shading cache because current architectures do not support concurrent reads and writes.

3.1. One-Pass Algorithm

The most straightforward approach uses a simple branch to follow either the recomputation or reprojection code path according to the depth test and refresh indicator (Figure 2). Because GPUs process nearby pixels in parallel, the performance of this approach will depend on the branch efficiency of the underlying hardware and the distribution of hits and misses across the framebuffer. Whenever a cache miss occurs inside a block of pixels that are processed together that also contains at least one cache hit, the amount of speedup that is lost is proportional to the difference in processing times for each path and the relative number of hits and misses within that block (in the worst case, a single miss is surrounded by hits). Because it is clearly desirable to cache values which are expensive to recompute from scratch, this difference can be significant and the fact that disocclusion boundaries and refresh patterns are typically distributed across the entire framebuffer (Figure 5) significantly limits possible gains.

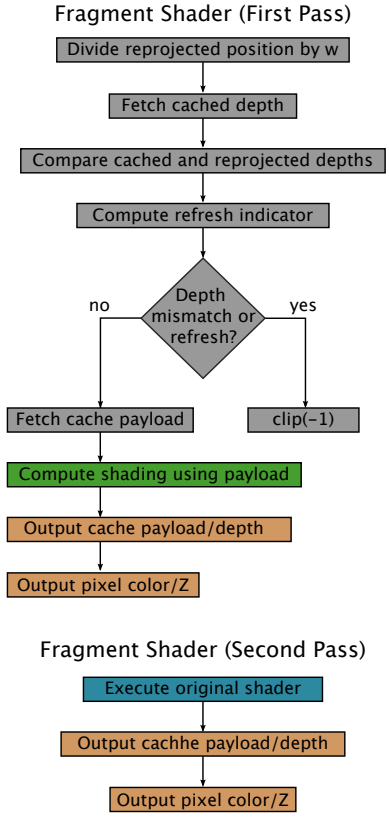


Figure 3: *Two-pass algorithm of a shading cache similar to that proposed by Nehab et al. [NSL*07].*

3.2. Two-Pass Algorithm

The fact that the previous algorithm relies heavily on efficient dynamic flow control (DFC) was identified and partially addressed by Nehab et al. [NSL*07] by using the Z-buffer in conjunction with two rendering passes as a mechanism for flow control (Figure 3). Unlike the one-pass algorithm, on a cache miss or forced refresh the shader simply primes the depth buffer to force execution in the next pass. In the second pass, the payload and final pixel color are both computed from scratch and output to their respective targets. Although this leads to greater data parallelism in the second pass, the two execution paths in the first pass may still have different processing times which can limit speedups for non-ideal DFC. This is most acute when the cost of completing the shading using a value restored from the cache is

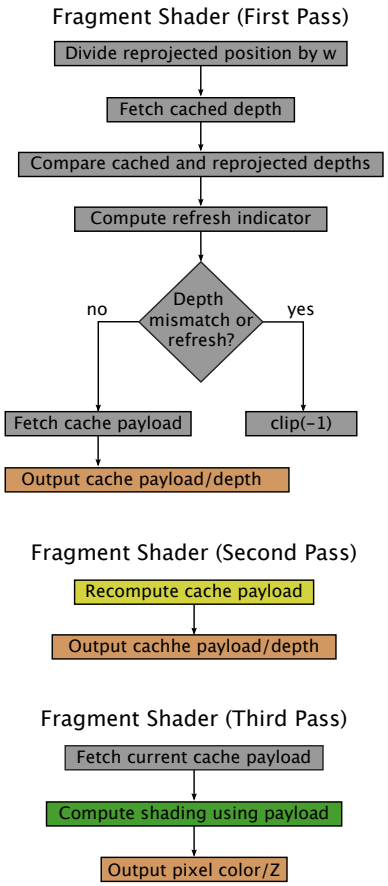


Figure 4: *Our alternative three-pass algorithm. This approach does not require the use of multiple render targets and is more efficient given the level of support for dynamic flow control in modern hardware.*

large. However, this will necessarily be less expensive than executing the original shader, so this approach tends to be more efficient than the one-pass approach in practice given the branch efficiency of modern GPUs. Finally, note that this approach also requires multiple render targets since the shading cache and framebuffer must be updated in each pass.

3.3. Three-Pass Algorithm

Figure 4 illustrates our proposed algorithm which requires three rendering passes. Although this approach is not ideal for every situation, it provides clear advantages over previous methods in many important cases. The first pass performs the same depth and refresh tests as before. However, in the case of a hit, it reconstructs the cache payload and simply writes this value to the shading cache for the current frame (along with scene depth). In the case of a miss or a forced refresh, it primes the depth buffer to force execution in the next pass. In a second pass, the cache payload is

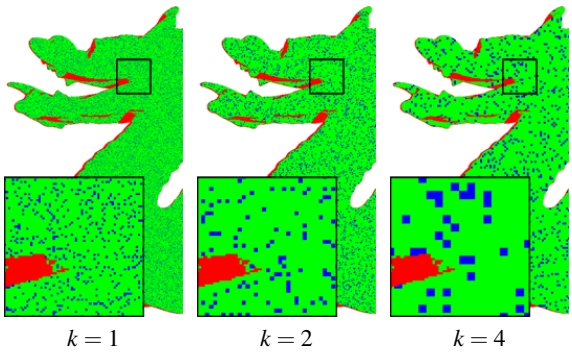


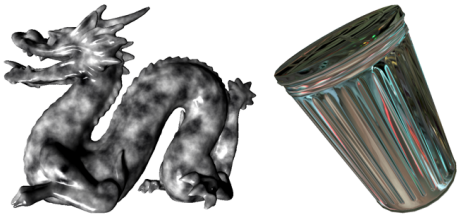
Figure 5: *Visualization of different refresh quad sizes. Green pixels correspond to cache hits, red pixels are misses, and blue pixels are forced refreshes. Note that misses occur at disocclusions and refreshes occur along a pre-determined random pattern uniformly distributed over the framebuffer within $k \times k$ blocks of pixels.*

computed from scratch for those pixels that were either not present in the cache or were refreshed. At this point the payload is available in the shading cache at *every* pixel. A third and final pass computes the pixel color using these payload values and updates only the framebuffer.

This approach has two key advantages over prior techniques. First, the processing times for each execution path in the branch are independent from the relative costs of evaluating the shader for a hit or miss; a single texture read is executed regardless of what is being cached. Second, only a single render target in each pass is required, which can significantly improve efficiency for many hardware platforms and scenes. This approach also has two important drawbacks. First, it requires an additional rendering pass. However, the relative cost of processing the geometry a third time compared to computing the shading is acceptable for pixel bound applications that would benefit from this type of data reuse in the first place. Second, potential compiler optimizations are lost since the calculation of the payload and final shading are decoupled even in the case of a cache miss. This can negatively affect performance when the payload and pixel color exhibit significant "computation overlap" as illustrated in Figure 7. However, we have found this situation is rare and that it tends to occur for values that are not good candidates for caching in any case. In our experiments, the fact that this three-pass approach has a better balanced branch and avoids the use of multiple render targets more than compensates for these drawbacks.

4. Results

We used a Dell XPS with an NVIDIA GeForce 8800 GTX and an ATI Radeon 2900 XT to generate the results reported in this paper. We used the scenes shown in Figure 6 to compare these three algorithms. The *Dragon* shader combines a procedural 3D noise function with a Blinn-Phong specular



Dragon Trashcan

Figure 6: *Test scenes used in our experiments. The Dragon model contains 75K faces and the Trashcan model contains 1K faces.*

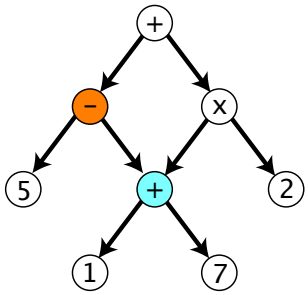


Figure 7: *Simple example of the computational overlap problem. This graph represents the expression* $(5-(1+7))+(2\cdot(1+7))$. *Caching the subexpression corresponding to the orange node would cause* $(1+7)$ *to be computed twice, once when recomputing the payload and again when computing the root node using this payload.*

layer and incorporates multiple texture accesses and expensive trigonometric functions. The *Trashcan* shader combines 25 samples of an environment map weighted by a Gaussian kernel. The sample directions are computed from a normal map plus a simple base geometry.

For each shader, we automatically generated versions that corresponded to caching each intermediate value for the three different algorithms previously described. This was done by first computing the abstract syntax tree (AST) [ALSU06] representation of each shader program and replacing every node in this tree by a subtree corresponding to a cache lookup. We then generated the source code corresponding to the modified AST and fed this to a standard compiler. We manually culled from this set of nodes those which were unsuitable for caching (e.g., values generated inside a `for` loop). The resulting set of shaders enabled us to evaluate these algorithms across a wide range of cache payloads.

We conducted a series of experiments aimed at analyzing the compute overhead of these algorithms when executed on modern NVIDIA and ATI hardware. Our goal was to examine the effect of the following parameters on rendering efficiency: the refresh quad size, the refresh period, and the relative cost of evaluating the cache payload P relative to the cost of evaluating the original shader T.

4.1. Relative Cost of Recomputing the Payload

The graphs in Figure 8 plot the average render time as a function of the intermediate value within the shader chosen for reprojection (i.e., a node in the AST) for all three algorithms. These numbers were generated using an animation sequence that shows the object rotating at a moderate rate in front of the camera. The nodes are sorted in ascending order based on the ratio of the cost of evaluating the payload P relative to the cost of evaluating the complete shader T as measured for NVIDIA hardware. These results assume a fixed refresh quad size of 4×4 and a refresh period of 32.

The trends in these graphs are consistent with our previous discussion: the difference in processing times between the two branch paths determines the relative efficiency of the 1-pass and 2-pass algorithms, while the overhead of our 3-pass algorithm is insensitive to this difference. Specifically, whenever a hit is much less expensive to process than a miss (i.e., P/T is large, toward the right of the graph) the 2-pass algorithm generally outperforms the 1-pass algorithm. The factor of improvement is a function of the branch efficiency of the underlying hardware and the pattern of hits and misses in the framebuffer. Conversely, as the cost of computing the final shading using the payload increases (i.e., $1-P/T$ is large, toward the left of each graph) the performance of the 2-pass algorithm decreases, along with the benefits of using data reprojection at all.

Another important benefit of our 3-pass algorithm is that it does not require multiple render targets. For these types of pixel bound scenes, we found that the overhead of using MRT exceeds the cost of an extra pass through the geometry for almost every node. Finally, note that the benefit of 3-pass vs. 2-pass decreases as P/T increases which we attribute to two factors. First, the processing time required for the miss branch in the 2-pass algorithm decreases, limiting the penalty of inefficient DFC. Second, the relative cost of the additional rendering pass increases as the scene becomes less pixel bound. In almost every case, however, the 3-pass algorithm is the most efficient (right column in Figure 8).

The outliers in these plots that violate these general trends correspond to particularly poor nodes to cache in which both hits and misses require a significant amount of processing. This is due to the *computational overlap problem* demonstrated in Figure 7.

4.2. Refresh Quad Size and Refresh Period

The graphs in Figure 9 plot rendering performance as a function of refresh period Δn for the *Dragon* shader modified to cache node #32. This node corresponds to the sum of two (of the five) octaves inside the Perlin noise calculation. The fact

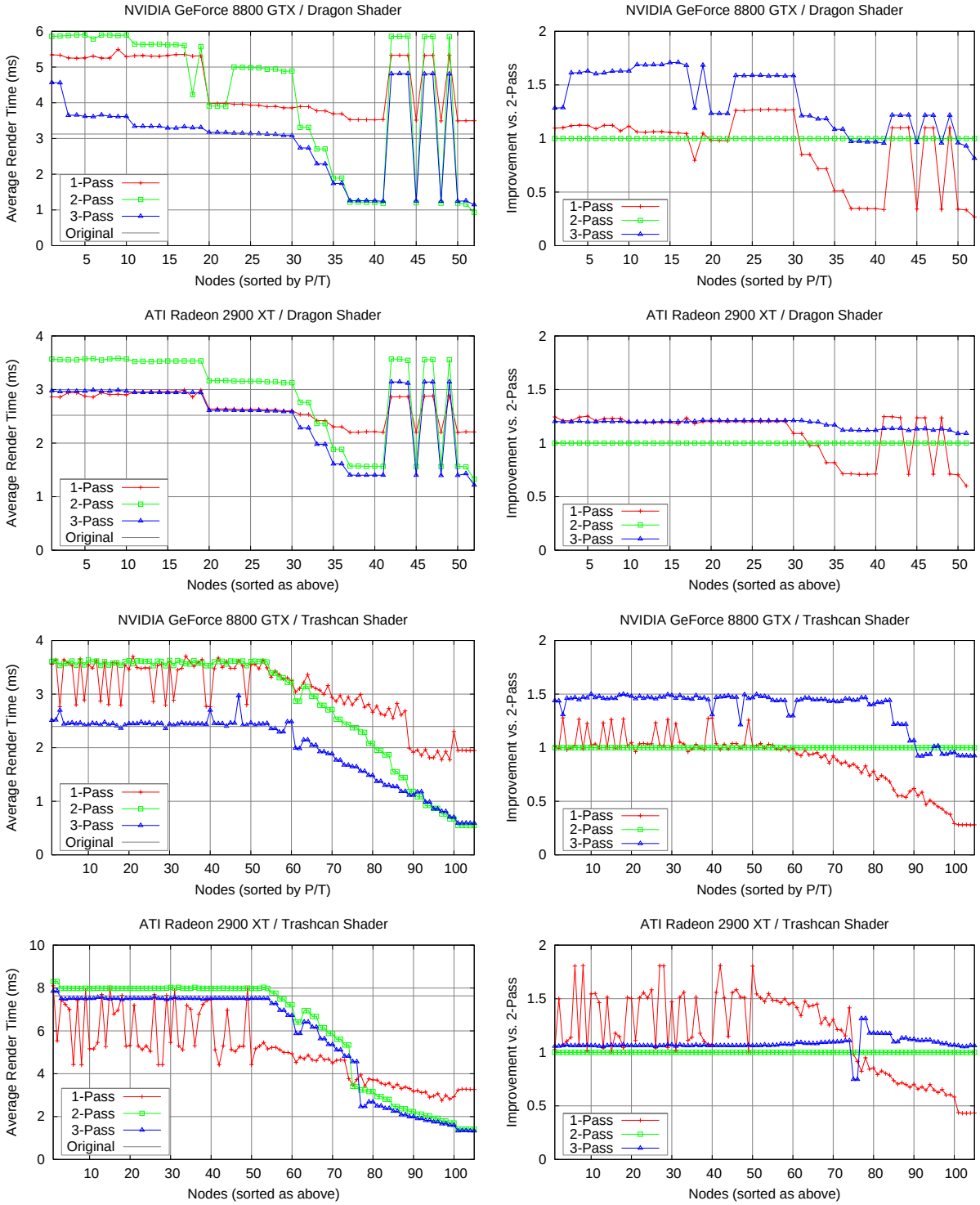


Figure 8: *Performance comparisons of the three different algorithms for multiple scenes and hardware platforms. In all cases a refresh quad size of* 4×4 *and a refresh period of* $\Delta n = 32$ *were used.* ***Left:*** *Average render times (lower values are better).* ***Right:*** *Rendering performance with respect to the 2-pass algorithm (higher values are better).*

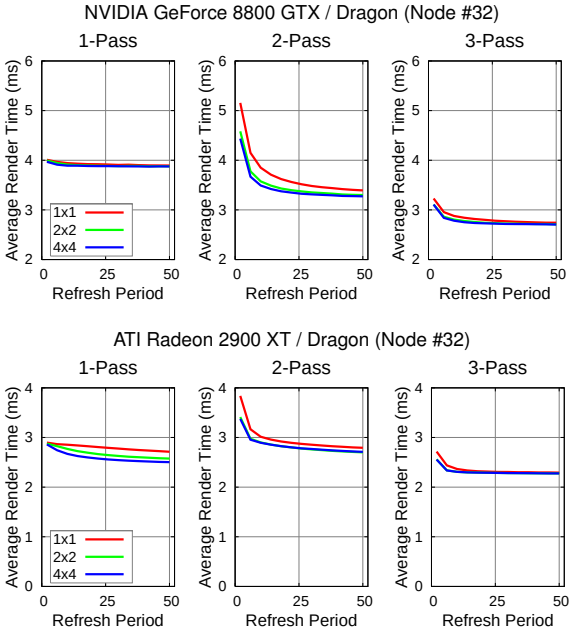


Figure 9: *Effect of refresh quad size and refresh period on performance (note different y-axis scales between top and bottom rows).*

that this value is independent of the light and view direction and exhibits low-frequency spatial frequencies makes it an ideal candidate for reprojection. We also measured the effect of the refresh quad size on rendering performance for $k = 1, 2, 4$ in Equation 1. We found that quad sizes beyond 4×4 behaved similarly to $k = 4$ and their results were omitted.

As expected, increasing the refresh period leads to a greater number of cache hits within each frame and causes a steady drop in rendering times. However, this rate of improvement decays exponentially which we attribute to the fact that the probability scene motion will prematurely force a cache miss increases alongside refresh period. For all three algorithms, we consistently observed a drop in rendering time when going from a refresh quad size of 1×1 to 2×2. We attribute this to the fact that modern hardware executes pixel shaders within 2×2 blocks together in order to approximate derivatives necessary for proper mip-map access so this becomes a lower bound on the branching granularity with any method; we were careful to align our refresh patterns with these underlying 2×2 pixel blocks through a simple trial-and-error process. We did not observe a significant improvement between $k = 2$ and $k = 4$ except in the case of the 1-pass algorithm on the ATI Radeon 2900. Our intuition is that this results from differences in the branching efficiency and thread scheduler between these architectures, although this information is not part of the public domain.

5. Conclusion

This paper analyzed the tradeoffs between different techniques for incorporating data reprojection into real-time applications. Specifically, we analyzed the overhead due to the level of support for DFC and MRT in modern GPUs. Based on this analysis, we proposed a new 3-pass algorithm which we verified is more efficient than existing methods over a wide range of cache loads and for multiple hardware platforms. Additionally, we analyzed the relationship between rendering efficiency and three fundamental parameters of a shading cache: the ratio of the costs of computing the payload and the entire shading, the refresh quad size, and the refresh period. We expect these results will provide further insights into how data reprojection can be best used to accelerate real-time systems.

References

[ALSU06] AHO A. V., LAM M. S., SETHI R., ULLMAN J. D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.

[AS06] AKELEY K., SU J.: Minimum triangle separation for correct z-buffer occlusion. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2006), pp. 27–30.

[HAM06] HASSELGREN J., AKENINE-MOLLER T.: An efficient multi-view rasterization architecture. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)* (2006).

[HM91] HECKBERT P., MORETON H.: Interpolation for polygon texture mapping and shading. In *State of the Art in Computer Grpahics: Visualization and Modeling*, Rogers D., Earnshaw R., (Eds.). Springer-Verlag, 1991, pp. 101–111.

[NSI06] NEHAB D., SANDER P. V., ISIDORO J. R.: *The Real-Time Reprojection Cache*. Tech. rep., Princeton University, 2006.

[NSL*07] NEHAB D., SANDER P. V., LAWRENCE J., TATARCHUK N., ISIDORO J. R.: Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware* (2007).

[SJW07] SCHERZER D., JESCHKE S., WIMMER M.: Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Proceedings of the Eurographics Symposium on Rendering (EGSR)* (2007).