The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading

Christopher A. Burns
Intel Labs

Warren A. Hunt
Intel Labs

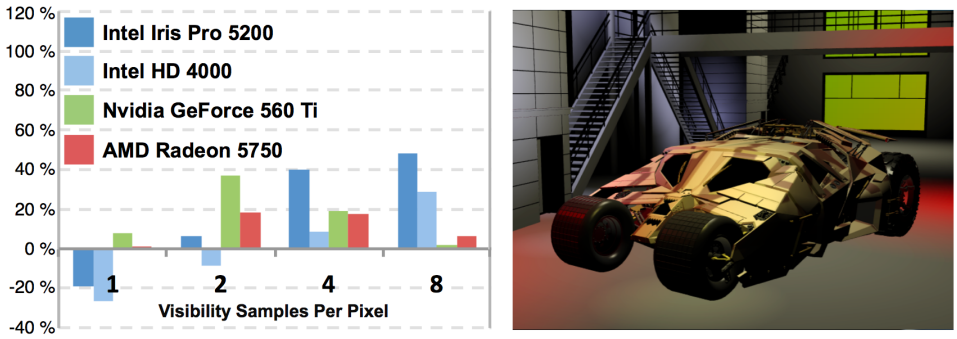

Figure 1. Compared to a conventional g-buffer pipeline, our technique demonstrates outperformance with large sample counts, especially on micro-architectures with deep cache hierarchies.

Abstract

Forward rendering pipelines shade fragments in triangle-submission order. Consequently, non-visible fragments are often wastefully shaded before being subsequently occluded—a phenomenon known as over-shading. A popular way to avoid over-shading is to only compute surface attributes for each fragment during a forward pass and store them in a buffer. Lighting is performed in a subsequent pass, consuming the attributes buffer. This strategy is commonly known as deferred shading.

We identify two notable deficits. First, the buffer for the geometric surface attributes—the *g-buffer*—is large, often 20+ bytes per visibility sample. The bandwidth required to read and write large g-buffers can be prohibitive on mobile or integrated GPUs. Second, the separation of shading work and visibility is incomplete. Surface attributes are eagerly computed in the forward pass on occluded fragments, wasting texture bandwidth and compute resources.

To address these problems, we propose to replace the g-buffer with a simple *visibility buffer* that only stores a triangle index and instance ID per sample, encoded in as few as four bytes. This significantly reduces storage and bandwidth requirements. Generating a visibility buffer is cheaper than generating a g-buffer and does not require texture reads or any

surface-material-specific computation. The deferred shading pass accesses triangle data with this index to compute barycentric coordinates and interpolated vertex data. By minimizing the memory footprint of the visibility solution, we reduce the working set of the deferred rendering pipeline. This results in improved performance on bandwidth-limited GPU platforms, especially for high-resolution workloads.

1. Introduction

Deferred shading techniques are attractive because they limit the lighting workload to visible fragments. On shade-as-you-go hardware, this is often achieved by resolving visibility in a pass that computes only surface attributes and stores them in a buffer—no lighting is performed. A second *deferred* pass is required to illuminate these surface fragments. The technique is popular because it neatly separates lighting from surface visibility and shading, and also because it avoids the wasted effort illuminating occluded fragments. Despite its effectiveness, this basic approach has several fundamental problems.

The size of the surface attribute buffer—the *g-buffer*—is typically 16 to 32 bytes per visibility sample in optimized high-quality real-time systems. The DRAM bandwidth consumed in writing this buffer, then reading it for each light pass is significant, even with only a single light pass. For example, a screen with a four-megapixel display, using four 24-byte samples per pixel at 60 Hz, would consume 46 GB/s of bandwidth, assuming only one lighting pass, just for the uncompressed g-buffer write and subsequent read. Thus, in practice, either anti-aliasing or pixel resolution (or both!) is often sacrificed to maintain high frame rates on economical hardware. This is perhaps the most serious issue with the technique, as low visibility sampling rates confound simple solutions to efficiently rendering partially transparent surfaces, edge anti-aliasing, and higher-dimensional rasterization.

The second issue is the imperfect separation of visibility and shading. With a g-buffer solution, the visibility pass must perform enough shading to generate the g-buffer. Thus, not all shading work is deferred, and some of that work, including texture sampling, is wastefully performed on hidden surfaces. An additional consequence is that geometry must be batched by surface material, closing off any opportunities for a more efficient scheduling of the visibility workload. Finally, by dividing the shading workload into two phases mediated by a very large DRAM buffer, all material surface attributes must conform to the same carefully constructed memory footprint. This constrains programmability and can burden the shader authoring process.

A third well-known problem concerns the correct rendering of partially transparent geometry. The g-buffer solution explicitly assumes a single visible surface per visibility sample. In practice, transparent geometry is rendered in a later forward pass, and composited onto the frame buffer.

To summarize, g-buffer deferred shading fails to cleanly separate visibility from per-pixel shading in a scalable way. The per-visibility-sample data written in the visibility pass does not abstract material properties. The data is too bulky to allow multisampling on a scale needed to implement advanced solutions to long-standing problems, such as aliasing. Partially transparent geometry must be rendered separately in subsequent forward rendering passes.

We address all of these problems except transparency. Our pipeline replaces the g-buffer with a simple *visibility buffer*. This buffer stores only a triangle index and instance id per sample. Only four bytes per sample are required, except for large or tessellated scenes, in which case eight bytes suffice. By capturing the full-screen primary visibility solution in as small a footprint as possible, we significantly reduce the bandwidth costs of deferred shading on multisampled or high resolution frame buffers. All shading occurs in a deferred pass in which the visible vertices are loaded, transformed, shaded, intersected, interpolated into screen-space, and illuminated. Our results show large speedups on GPU platforms featuring sizable caches.

1.1. Related Work

Most GPU hardware implements a sort-last, forward shading rendering pipeline, the logic of which is dictated by common graphics APIs such as OpenGL and DirectX. Fragments are shaded as they exit the rasterizer if they pass the early *Z*/Stencil test. Therefore, any scheme to defer shading until visibility is finally resolved must be implemented in user-space with multiple rendering passes.

One of the simplest solutions is known as z-prepass. All scene geometry is rasterized first with a null pixel shader to prime the z-buffer with the nearest surface depth. A second standard pass with actual shader is performed, with the z-test set to equal. A modern variant of this approach is Forward+ [Harada 2012] which uses the depth buffer from the z-only pass to perform light-culling on a tile granularity, resulting in an efficient forward shading pass. While effective for simpler scenes, it scales poorly with geometric complexity, as it requires resubmission of all potentially visible geometry. For many high-end production rendering systems the cost of resubmission is prohibitive [Andersson 2011]. Therefore, we seek an approach that performs visibility calculations only once.

Light pre-pass rendering, sometimes referred to as *deferred lighting*, modifies the z-prepass approach in a way that separates irradiance calculation from surface shading [Engel 2008]. The z-only pass is extended to emit shading normals and specular lobe values in addition to z. A full-screen-quad pass evaluates diffuse and specular irradiance. Finally, geometry is resubmitted to integrate irradiance with fully-evaluated surface attributes. This approach nicely separates lighting from surface shading, but still requires two geometry passes.

Avoiding the second visibility pass is achieved by further extending the first pass to write all surface attributes required for shading, rather than just those needed to compute irradiance. As we have mentioned, this buffer is large and limits resolution and/or sample rates on most hardware.

Liktor and Dachsbacher present a scheme for compressing this buffer while simultaneously supporting motion blur and depth of field [Liktor and Dachsbacher 2012]. Their *compressed geometry buffer* only stores surface attributes at shading samples, which are placed more sparsely than the visibility sampling rate. Visibility samples store references to these shading samples. Their approach reduces storage costs as compared to a fully super-sampled g-buffer in the case where the shading rate is less than the visibility rate and permits arbitrary many-to-one mappings between visibility samples and shading samples. While their approach is promising for applications that use stochastic sampling to resolve blurred geometry, its benefits to traditional rasterization pipelines are more modest. It also appears to require additional hardware (e.g., on-chip caches accessible from pixel shaders) to execute efficiently in a GPU rasterization pipeline.

Another way to limit the footprint of an intermediate render target is to partition the frame buffer into tiles small enough to fit into on-chip caches. This strategy was used by Intel's Larrabee [Seiler et al. 2008] graphics driver. Using a tiled approach in the application layer is problematic, however, because it amplifies the number of required draw calls to the graphics API by a factor nearly proportional to the number of tiles. Even in the absence of tiling, draw calls are a frequently cited bottleneck in modern real-time rendering systems.

Low-power GPUs such as the PowerVR product line from Imagination Technologies [Imagination Technologies Ltd. 2011] are known to pursue a strategy similar to ours in the driver and hardware layers. They implement a tiled renderer that resolves visibility for a tile before interpolating attributes or executing the pixel shader. References to rasterized triangles are stored in a tile-sized hardware *tag buffer*, similar to our full-screen visibility buffer. Our technique can be viewed as a full-screen application-layer implementation of this strategy, targeting DirectX 11-class GPUs. To our knowledge, this is the first published analysis of such an implementation.

2. The Visibility Buffer

Our solution, as compared to a traditional g-buffer solution, is illustrated in Figure 2. All geometry is rendered into a visibility buffer that, for each visibility sample, stores a four-byte integer that encodes a triangle id and an instance id. For scenes too large (or with tessellated geometry, discussed in Section 5.1) for this compact encoding, eight bytes can be used per sample. Once visibility is complete, we construct, for each surface material, a worklist of tiles (16×8 pixels) requiring shading. Lastly,

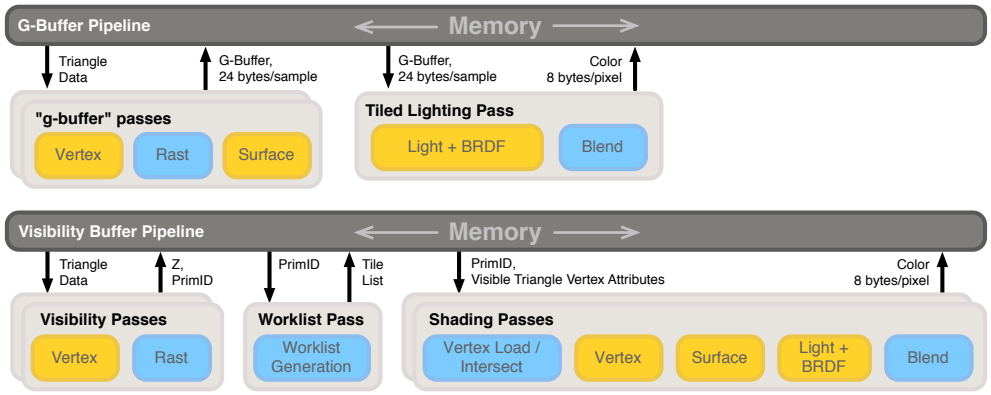

Figure 2. Illustration of the key differences between a conventional g-buffer pipeline (above) and our visibility buffer pipeline (below). Instead of writing surface attributes to memory on the forward visibility pass, we record only a triangle and instance id, which can be as small as four bytes. Our deferred pass is responsible for all shading, and loads only visible triangle vertex data as indicated by the visibility buffer. The compact representation of the visibility buffer enables high sampling rates on platforms with limited bandwidth to main memory and an effective cache hierarchy. Interior orange boxes represent user-programmable logical stages, while blue indicates non-programmable functions.

shading kernels are dispatched for each unique material in the scene, ranging over the list of tiles known to contain at least one sample of visible geometry with that material. This kernel performs all shading, from vertex interpolation to lighting, and outputs a single color per pixel. Pseudocode for this kernel is shown in Figure 3.

The primary motivation of this approach is to reduce the memory footprint— and hence the required bandwidth—of the buffer storing results from the visibility pass. For a 1080 p frame with 8x sampling, our buffer is 64 MB, compared with 398 MB for a 24-byte-per-sample g-buffer. This comes at the cost of loading triangle and vertex data for the visible triangles during shading. However, this additional data is still small compared to a multisampled g-buffer (vertex and index data for two million micro-polygons would fill about 88 MB if we assume roughly one unique vertex per triangle, and a 32-byte vertex). More importantly, *these memory accesses are highly coherent in screen-space, likely resulting in high cache hit rates even if the cache is significantly smaller than the total working set*.

We pay for the storage savings by recomputing several quantities. These overheads include barycentric coordinate generation, vertex attribute interpolation, texture coordinate differential calculation, and recomputation of the vertex shaders at pixel frequency. Except for very high-bandwidth discrete GPU platforms, bandwidth is the constraining resource for mainstream graphics workloads, especially on power-sensitive micro-architectures. Longer term trends indicate continued growth in the compute-bandwidth ratio. Therefore, we conjecture that this is a sensible tradeoff.

```
1  void ShadingPass
2  {
3     // Load the visibility sample, decode
4     int vSample    = loadVisSample(pixelID)
5     int instanceID = decodeInstanceID(vSample)
6     int triID      = decodeTriangleID(vSample)
7
8     // Bail out if this pixel requires another shader
9     int shaderID = instToShaderMap[instanceID]
10    if (shaderID != SHADER_ID): return
11
12    // Load the indices for the visible triangle
13    int base = instToIndexBaseMap[instanceID]
14    int3 indices = loadIndices(base, triID)
15
16    // Load the vertices for the visible triangle
17    int base instToVertexBaseMap[instanceID]
18    Vertex v0, v1, v2 = loadVertices(base, indices)
19
20    // Ray-tri intersection, vertex attrib interp.
21    float16 objToScreenSpace =
22       worldToScreen * objToWorld[instanceID]
23    float3 barys = computeInterpolants
24       (v0.pos, v1.pos, v2.pos, objToScreen, pixelID)
25    Vertex v = v0*barys.x + v1*barys.y + v2*barys.z
26
27    // Run user-written shading kernels
28    Vertex vShaded = ShadeVertex(v)   // vert shader
29    Surface s = ShadeSurface(vShaded) // surf shader
30
31    // Light loop: iterate over lights, apply shaders
32    float4 color = 0.0
33    for light from 0 to numLights:
34       Radiance r = ShadeLightSource(s, light)
35       color += ShadeBRDF(s, r)
36
37    writeColorToBuffer(color, pixelID)
38 }
```

Figure 3. Pseudocode for our deferred shading pass. For each unique material in the scene—defined as a vertex shader, surface shader, and BRDF shader assigned to at least one instance—an HLSL compute kernel is constructed of this form. Note, that almost all of the loads are of coherent data that is not unique to each pixel, but likely to be shared across many pixels, depending on triangle size. We also note that code from line 28 on is virtually identical to the lighting pass in our reference g-buffer system. We show one visibility sample per pixel, for simplicity. Section 3.3 explains how we extend this to support multisample anti-aliasing.

Finally we'll note that in none of our scenes were frame rates sensitive to the addition of math instructions in the shaders, indicating that we are not compute-bound.

One final overhead we incur is the need to generate tile worklists for each surface material. In a g-buffer system, the rasterization pass is coupled to pixel shader programs that evaluate surface attributes. This means that one draw call, at a minimum, is issued for each distinct surface. In contrast, the lighting stage code is generally invariant with respect to the surfaces and can be applied to the entire frame in a single

dispatch. Since our system moves *all* surface shading to the second phase, our situation is reversed. We must dispatch a shader for each unique material, and we'd like to do so only for relevant portions of the frame, thus the need for a brief stage to build a list of tiles for each shader. This stage is very quick and accounts for no more than 10 % of frame time in any of our scenes.

3. Implementation

We have implemented a reference g-buffer deferred pipeline alongside our visibility buffer pipeline, using DirectX 11 pixel and compute shaders as appropriate. Figure 2 illustrates how we map logical rendering tasks (orange and blue boxes) into passes for both pipelines

We do not implement light culling for our analysis. Given a depth buffer, tiled light culling is fairly well understood and does not have meaningful ramifications on the issues discussed here. Both renderers simply shade from per-tile light lists as if culling had been performed. We also do not address the issue of partially transparent surfaces, which are known to be problematic in deferred shading systems. We assert that our pipeline is compatible with conventional solutions to rendering partially transparent geometry in a g-buffer pipeline.

3.1. Reference G-Buffer Pipeline

Our reference g-buffer pipeline uses the 24-byte g-buffer format described in Listing 1. Normals are compressed using a sphere map encoding. We also store four bytes for triangle and instance id, exactly as in our visibility buffer pipeline in order to support the anti-aliasing scheme (see Section 3.3). Lighting is executed in a single compute shader kernel that illuminates 16×8 pixel tiles with per-tile light lists. Our choice of tile size is limited on the high end by the availability of group shared memory (which we use for anti-aliasing). Tile sizes smaller than 64 pixels underperformed, as smaller workgroups tend to undersubscribe modern GPUs.

3.2. The Visibility Buffer Pipeline

Our pipeline is implemented in three phases, as shown in Figure 2. The first phase resolves visibility, recording depth and primitive id into a multisampled buffer. No other data is written or computed.

The second phase builds tile worklists to drive phase three, and it is broken into two small parts. The first part reads the visibility buffer and builds a list of records, each consisting of a tile-shader pair (eight bytes). The second part sorts these records by shader and records only the tile ids in a packed list. An array of offsets indicates for each shader where its portion of the tile list begins. This phase is relatively inexpensive, as will be discussed in Section 4.

```
struct GBuffer {
   half4 positionXYZ_;       // Texture 0, 8 bytes, unused w
   half2 normalXY;           // Texture 1, 4 bytes
   unorm84 diffuseRGB_;      // Texture 2, 4 bytes, unused w
   unorm84 specularRGB_exp; // Texture 3, 4 bytes
   uint triangleID;          // Texture 4, 4 bytes
};
```

Listing 1. Our reference pipeline's 24-byte g-buffer format. The size of the g-buffer struct has a large impact on bandwidth consumption. We address this by replacing the g-buffer with a compact visibility buffer. Different systems have different needs as to the content of this struct, and consequently its size, but our choice is not atypical.

The third and final phase consists of a compute kernel dispatch for each surface shader in the scene. These kernels consume the visibility buffer and are responsible for vertex gather, vertex shade, attribute interpolation, surface shading, illumination, multisample blending, and the final write to the color buffer. Pseudocode for this kernel is shown in Figure 3. As described in Section 2, the vertex gather, intersection, interpolation, and shading is the computational price we pay for a compact visibility buffer. Once a thread computes an interpolated vertex for a given shading sample (line 25 in Figure 3), the kernel proceeds more or less identically to the lighting phase of the reference g-buffer pipeline. We also use the same $16\times$ 8 tile size.

3.3. Anti-Aliasing

Anti-aliasing is a challenging feature to efficiently support on GPU hardware in any deferred shading scheme. Hardware MSAA is only available in forward passes through the rasterizer, forcing deferred or compute-mode driven passes to roll their own. A line of research beginning with MLAA [Reshetov 2009] has yielded a series of techniques that are cost-effective but low quality. These are used largely to avoid the expense of multisampling, which we address directly. Since efficient high visibility sampling rates is a key benefit of our approach, an honest evaluation of our technique requires that we do something "smarter" than super-sampling. Shading every visibility sample would skew the performance results by over-representing shading costs as a fraction of total frame time.

Most multisample deferred anti-aliasing algorithms take the following form:

1. Generate a super-sampled buffer of useful surface attributes (Z, normal, etc.);
2. Do some work to determine what samples need to be shaded;
3. Schedule the shading work.

Our approach to anti-aliasing uses the visibility buffer generated in the first pass to determine which samples within a pixel require shading. We iterate over the samples

within a pixel and create one shading sample record (stored in group shared memory) for each unique visibility sample. These samples are weighted in proportion to the number of times they appear within the pixel. Each shading record is assigned a thread in the thread group, is shaded, and the color recorded in the shading record. Finally, the colors are blended, and a single color is written for each pixel. Packing the records and re-distributing them to the threads preserves SIMD efficiency.

Our approach is not highly exotic or novel, but instead represents a reasonable design choice that could be implemented identically in both pipelines. It results in one shading sample per pixel, per triangle. We experimented with a simpler approach that uses the stencil buffer to mark pixels as either requiring super-sampling or not [Lauritzen 2010], but we found the approach described above to moderately outperform on our scenes across most of the tested GPU platforms. We do not want to overstate this claim however, as these tradeoffs are highly sensitive to workload characteristics, as well as micro-architectural issues.

4. Results

We've tested our pipeline on three scenes of varying geometric and shading complexity, shown in Figure 4. These scenes' characteristics are presented in Table 1. All scenes were rendered at 1080p resolution on the four different GPU platforms described in Table 2. These platforms exhibit notably different memory subsystems. Discrete GPUs tend to feature high DRAM bandwidth and only modest quantities of general-purpose cache. In contrast, the integrated Intel GPU parts employ a CPU-like memory hierarchy. In particular the Intel Iris Pro 5200 "Haswell" integrated GPU features an additional level of off-die embedded DRAM that functions as a high-bandwidth 128 MB cache.

Our results are shown in Figure 5. As expected, our pipeline offers little to no net benefit with only two million visibility samples. The performance advantage accrues with higher sample rates, particularly on the bandwidth-constrained Intel platforms.

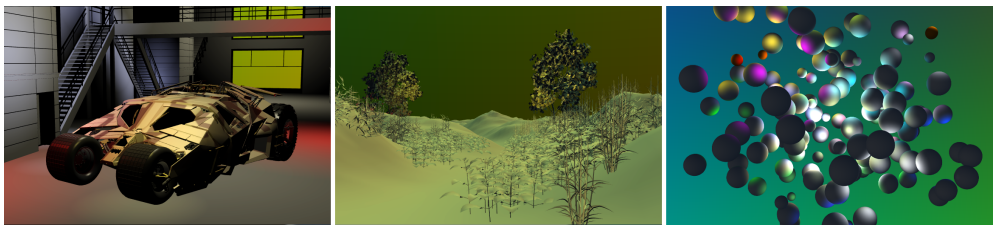

Figure 4. We evaluated our system on these three scenes with a variety of hardware, rendered at 1080p resolution. While the left two scenes are geometrically complex, the spheres scene is contrived to observe the performance of scenes with minimal unique geometry but non-trivial lighting.

Scene Statistics			
Quantity	Warehouse	Ecosystem	Spheres
Unique Vertices	1,422,012	1,293,435	439
Unique Triangles	559,604	431,145	760
Rasterized Triangles	559,604	1,744,380	92,720
Lights per Pixel	24	8	32
Shading Samples / Covered Pixel	1.2	1.5	1.5

Table 1. Data for the scenes used in our evaluation, as pictured in Figure 4. Unique triangles and rasterized triangles may differ due to the use of instancing.

GPU Memory Hierarchy Comparison				
GPU	1st Cache	2nd Cache	3rd Cache	DRAM B/W
AMD Radeon HD 5750	40 kB	256 kB	-	73.6 GB/s
Nvidia GeForce 560 Ti	384 kB	512 kB	-	128.0 GB/s
Intel HD 4000	256 kB	8 MB	-	25.6 GB/s
Intel Iris Pro 5200	512 kB	8 MB	128 MB	25.6 GB/s

Table 2. Memory hierarchy data for each of the GPU architectures tested in our evaluation. Our visibility buffer pipeline favors systems with larger caches. We do not show caches dedicated to texture sampling, z-testing, or other specialized caches not available for general load-store operations. We also adjust the reported first-level cache memory available to the Nvidia and Intel GPUs to account for the fact that a portion of physical cache is reserved for use as group shared memory in our kernels.

The warehouse scene features a wide range of triangle sizes and materials. Our visibility buffer pipeline scales with sample rate much better than the g-buffer pipeline, and particularly so with the Intel Iris Pro. The ecosystem scene has many small trian-

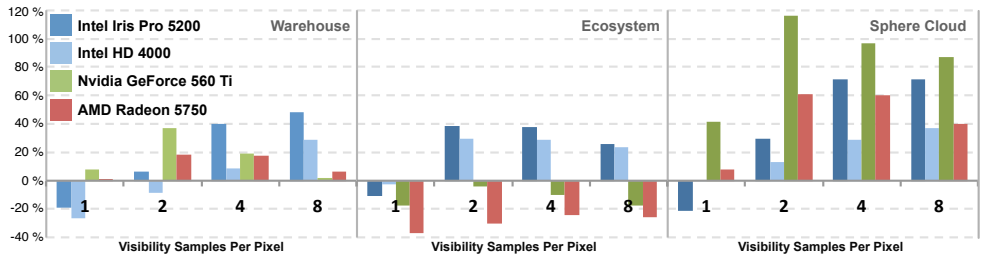

Figure 5. We plot the percentage speedup ($\frac{fps_0}{fps_1} - 1.0$) of our pipeline over the standard g-buffer pipeline as described in Section 3, for four different GPUs, three scenes, and four sampling rates at 1080p resolution. As expected, the gains are most pronounced on architectures with limited DRAM bandwidth and large cache hierarchies, and where the multisample rate is high enough that the system is bandwidth limited.

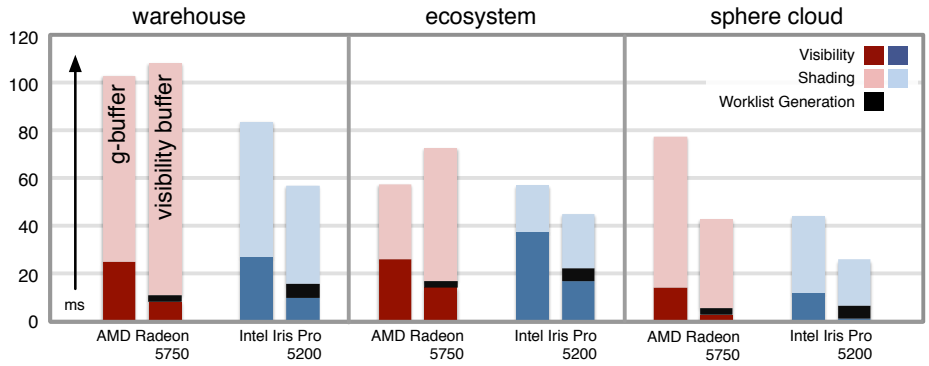

Figure 6. We break down the execution time of the key phases of our g-buffer control renderer, and our implementation of the visibility buffer pipeline. The scenes were rendered at 1080p resolution with eight samples per pixel. In each pair of columns, the left represents the g-buffer pipeline, the right is our pipeline. These two GPUs are comparable in their peak arithmetic throughput, making them a reasonable case study in how memory hierarchy affects rendering performance.

gles, challenging our approach, as it depends on the spatial locality of visible geometry during the shading phase. On this scene, the platforms with the smallest caches never benefit from our approach, while the integrated parts still perform well.

The sphere-cloud scene contains a single sphere mesh, instanced 200 times. The mesh data fits entirely within even the small caches on all platforms. We suggest that this is why our pipeline outperforms on all tested hardware on this scene. While unrealistic, it demonstrates the impact of geometric working-set size on our algorithm.

Figure 6 breaks down the cost of the major phases of each pipeline in the 8x multisampled case. Not pictured are the (minor) costs of buffer clearing, back buffer swap, and small gaps in utilization. We note that the visibility pass in our pipeline is far cheaper than in the g-buffer pipeline, similar to a z-prepass. In some cases, our shading pass outperforms the corresponding lighting pass of the g-buffer pipeline, despite the computational overheads. The Radeon 5750 is more efficient in building the worklists (the black segment) than the Intel part, but does not execute our pipeline's shading pass as well as the Intel parts. We believe this is because the additional memory bandwidth does not make up for the lack of an effective cache hierarchy. These two GPUs are comparable in their arithmetic peak throughput, making them a reasonable case study in how memory hierarchy affects rendering performance.

5. Discussion

We believe this work is important for three key reasons. First, rendering is about sampling. Higher sample rates improve the quality of just about every aspect of rendering. For primary visibility, higher sampling density addresses not only geometric aliasing,

but can be used to implement motion blur and/or depth of field [McGuire et al. 2010], order-independent transparency [Enderton et al. 2010], and potentially other effects as well. Our technique efficiently enables higher sample rates without resorting to forward shading or multiple visibility passes.

Second, in addition to enabling higher sample rates, our pipeline is, in one important sense, more "deferred" than the g-buffer pipeline. In a g-buffer pipeline, shading attributes are computed in the first pass. Even if this is inexpensive compared to lighting, it requires that geometry be sorted by surface material during the visibility pass and shaders/textures/constants swapped out between batches. It is unlikely that this ordering is optimal for visibility. Since our pipeline performs no surface-specific computations during the visibility pass, it provides opportunities for innovation in visibility algorithms. For example, it may allow for smarter approaches to culling.

Third, power efficiency is of paramount concern for the foreseeable future, and moving data to/from memory is among the most power-hungry operations. Algorithms that perform additional compute to reduce memory traffic will almost always come out ahead by this measure and will outperform on machines that strictly limit bandwidth to distant resources. While it may be the case that future architectures will have significantly more memory bandwidth than today's, compute resources will likely scale faster, which means the *relative* cost of memory bandwidth will continue to increase.

5.1. Tessellation

Tessellated triangles are generated and consumed in the front half of the pipeline. Since they are not stored, they are unavailable for loading during the deferred stages of our system. This requires the deferred stage for tessellated geometry to operate a bit differently than for non-tessellated geometry. Our implementation does not currently support tessellation, but we give consideration here as to how this would work.

Instead of recording triangle id and instance id in 32-bits, we would store patch UV coordinates (16-bit float each), patch id, and instance id, all in eight bytes. The deferred pass must then, for each shading sample, load the patch control points and recompute the domain shader for each sample before proceeding with surface shading. For some scenes, this may involve a fair amount of compute overhead, but it is important to recall that only *visible* geometry is touched in the second pass. Tessellation is not executed twice, only hull shading and domain shading, and only at visible shading samples. The visibility buffer doubles in size, but is still significantly smaller than a multisampled g-buffer.

6. Conclusion

We have implemented a novel real-time rendering pipeline that recomputes data in the deferred pass that would normally be stored during the visibility pass. By doing so,

we have significantly reduced the size of the effective working set during the deferred pass so as to better exploit cache hierarchies and minimize traffic to DRAM. We have also completely decoupled the visibility pass from the shading pass(es), without requiring a second trip through the pipeline front-end as with a simple z-prepass. Our results demonstrate the performance advantages of our pipeline on GPU architectures with limited DRAM bandwidth, but useful cache hierarchies. We hope these innovations will spur additional research on how best to design rendering systems targeting future SoC platforms, where power is always a concern.

Acknowledgements

The authors would like to thank Jim Hurley and the folks in the Visual Applications Research Lab in Intel Labs for their contributions, encouragement, and support. Additionally we are grateful to Aaron Lefohn, Chuck Lingle, and the rest of the Advanced Rendering Technologies group, as their input is always highly valued.

References

ANDERSSON, J. 2011. DirectX 11 rendering in Battlefield 3. In *Game Developers Conference 2011*. Available at http://www.slideshare.net/DICEStudio/directx-11-rendering-in-battlefield-3. 57

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, I3D '10, 157–164. 66

ENGEL, W. 2008. Designing a renderer for multiple lights: The light pre-pass renderer. In *Shader X7*, Charles River Media, Hingham, MA, W. Engel, Ed., 655–666. 57

HARADA, T. 2012. A 2.5d culling for forward+. In *SIGGRAPH Asia 2012 Technical Briefs*, ACM, New York, NY, USA, SA '12, 18:1–18:4. 57

IMAGINATION TECHNOLOGIES LTD. 2011. *POWERVR Series5 Graphics - SGX Architecture Guide for Developers*. Imagination Technologies Ltd., Kings Langley, UK. 58

LAURITZEN, A. 2010. Deferred rendering for current and future rendering pipelines. In *Beyond Programmable Shading, SIGGRAPH 2010 Course Notes*, ACM, New York, NY. 63

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, I3D '12, 143–150. 58

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-time stochastic rasterization on conventional GPU architectures. In *Proceedings of High Performance Graphics 2010*, Eurographics Association, Aire-la-Ville, Switzerland. 66

RESHETOV, A. 2009. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 109–116. 62

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3, 1–15. 58

Index of Supplemental Materials

We have provided HLSL shader code for the visibility pipeline algorithm. These shaders are generated at run-time from a set of user-provided code fragments (in this case, the warehouse scene). Our system is designed to allow the user-defined hlsl fragments (e.g., vertex shade, surface shade, light shade) to be written once and used in a variety of rendering pipeline implementations. The visibility buffer pipeline (as shown in Figure 2) is one such implementation. The reader may find them useful to clarify any details imperfectly explained in the text.

- **VisibilityPass.Vertex.hlsl**—Example vertex shader for the visibility pass;
- **VisibilityPass.Pixel.hlsl**—The pixel shader for the visibility pass, outputs only the primitive ID and (implicitly) depth;
- **WorklistPass.BuildList.hlsl**—Compute shader for first phase of worklist generation;
- **WorklistPass.SortList.hlsl**—Compute shader for second phase of worklist generation;
- **DeferredPass.hlsl**—Example compute shader implementing the deferred shading pass.

Author Contact Information

Christopher A. Burns
Intel Labs
burns.christopher.a@gmail.com

Warren A. Hunt
Intel Labs
warren.hunt@gmail.com

Christopher A. Burns, Warren A. Hunt, The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, 55–69, 2013
http://jcgt.org/published/0002/02/04/

Received:	2013-05-16		
Recommended:	2013-07-05	Corresponding Editor:	Morgan McGuire
Published:	2013-08-12	Editor-in-Chief:	Morgan McGuire