Deinterleaved Texturing for Cache-Efficient Interleaved Sampling

Louis Bavoil
lbavoil@nvidia.com

March 2014

Document Change History

Version	Date	Responsible	Reason for Change
1	March 6, 2014	Louis Bavoil	Initial release

Overview

This DirectX 11 sample demonstrates how a large, sparse and jittered post-processing filter (here a SSAO pass with a 4x4 random texture) can be made more cache-efficient by using a Deinterleaved Texturing approach. First, the input full-resolution texture is restructured into an array of 16 quarter-resolution textures. Second, the filter is rendered using 16 separate quarter-resolution passes sourcing one quarter-resolution texture per pass.

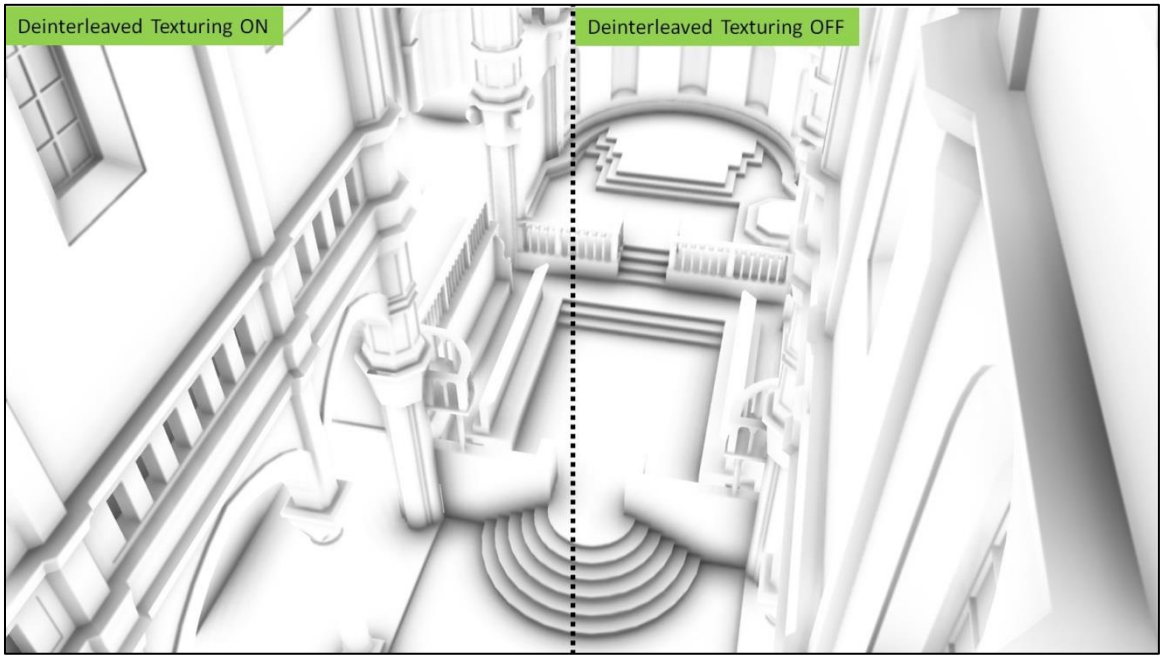

Figure 1. Using Deinterleaved Texturing is up to 2.3x faster than a single pass with per-pixel randomization, and looks very close.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Deinterleaved Texturing

Introduction

The post-processing filters that we are aiming at speeding up are large, sparse and jittered filters, for instance a Screen-Space Ambient Occlusion (SSAO) pixel shader using disk sampling and randomized texture coordinates per pixel [McGuire et al. 2012]. Other algorithms that could benefit from this approach are SSDO [Ritschel et al. 2009] and SSR [Kasyan et al. 2011]. All these algorithms are randomizing texture coordinates to trade bending for noise. We assume that the performance of the original implementation of the filter is primarily limited by texture latency. Our goal is to come up with a generic approach to speed up such large filters without sacrificing quality.

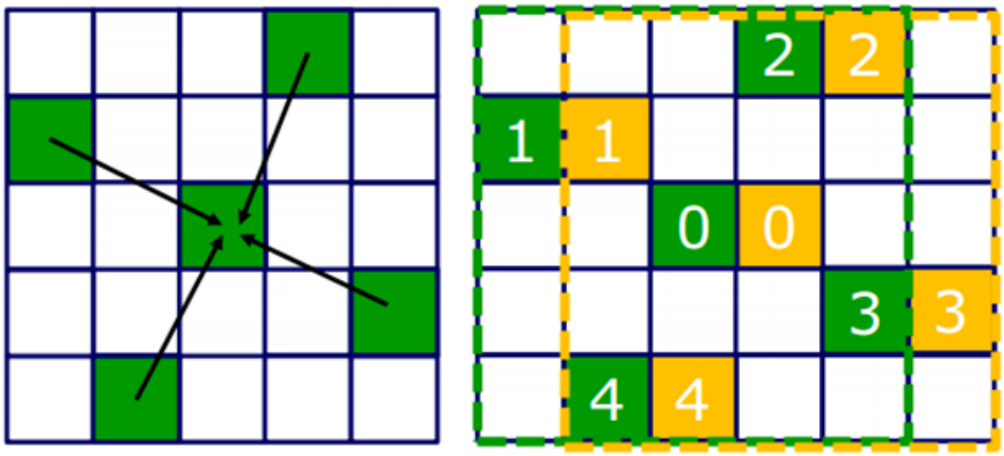

Figure 2. Fixed sampling pattern.

In the example sampling pattern shown on Figure 2, adjacent pixels are fetching adjacent samples. In this case, each pixel is gathering 4 texels around it at fixed offsets relative to the center pixel. If we look at a pair of adjacent pixels being executed in lock step (pixels marked as 0), the sample coordinates for each sample are adjacent to each other (samples marked as 1, 2, 3, 4). Therefore, this sampling pattern has good spatial texture locality, which is friendly to texture-cache hardware.

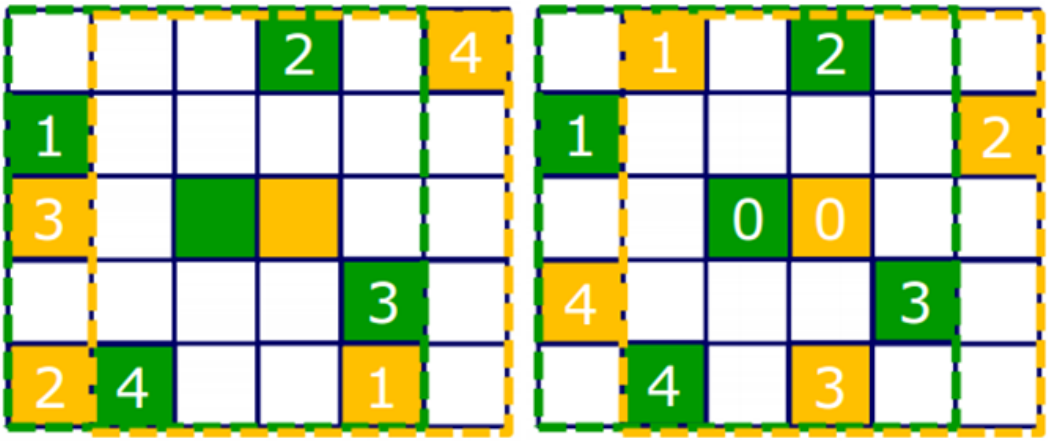

Figure 3. (a) Random sampling. (b) Sectored sampling.

Now if we consider the same 2 pixels with an arbitrary per-pixel randomization of the sampling pattern as shown on Figure 3a, the footprint of the first sample for the 2 adjacent pixels is spread out. That will cause inefficiencies in texture hardware. In this case, we have adjacent pixels fetching far-apart texels, yielding poor spatial locality.

A typical strategy is to do sectored, jittered sampling instead of fully randomized sampling. In this case, we have 4 samples and can subdivide the kernel area into 4 quarters. We can take sample 1 in the top-left quarter, sample 2 in the top-right quarter, and so on. In this way, the samples being executed in lockstep will be close together, in each sector. So they will have better spatial locality. However, for large kernels adjacent pixels will still be fetching far-apart texels within their sector and the texture-cache hit rates will suffer.

Previous Art

One simple strategy is using mixed-resolution inputs, binding both the high-resolution and a lower-resolution version of the input and:

- for the center tap of the kernel, use the full-resolution texture
- for the scattered far-away taps, use the low-resolution texture.

Another strategy is to use a mip-mapped input texture as in [McGuire et al. 2012] and to adapt the sampled LOD based on the step size at the current pixel. This way, adjacent pixels may fetch closer samples to each other in the same mip level.

These two strategies are still sub-optimal in terms of spatial locality because they still need to use a per-pixel jittering mechanism to avoid banding.

Our Approach

Our approach is to render one low-resolution image per unique sampling pattern, which is an old idea from [Keller and Heidrich 2001] also used in [Segovia et al. 2006] and [Bavoil and Jansen 2013].

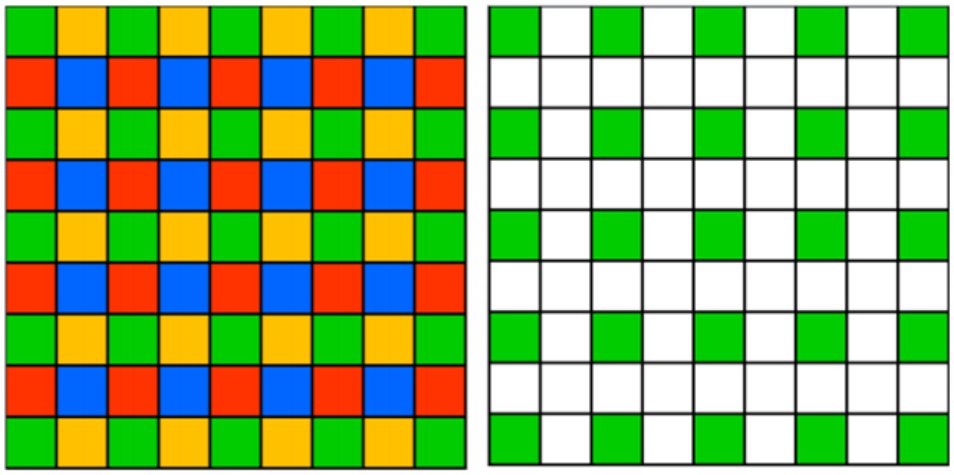

Figure 4. 2x2-interleaved sampling patterns.

We start from the 2x2 interleaved sampling patterns shown on Figure 4 (left). And we process all of the green pixels (which have the green sampling pattern) together, in one separate pass. Then we process all the red sampling patterns, in another separate pass, and so on.

We store all the results in intermediate textures. So at the end of this we have processed the full-resolution output. We process all of the pixels, and do that in separate passes per sampling pattern, so there is no per-pixel randomization needed in the pixel shader anymore. On top of that, we also use downsampled input textures so that adjacent pixels can fetch adjacent texels.

To sum up:
1. We render each sampling pattern **separately.**
2. For each of these passes, we use **downsampled input textures**.

We refer to this approach as Deinterleaved Texturing.

Algorithm

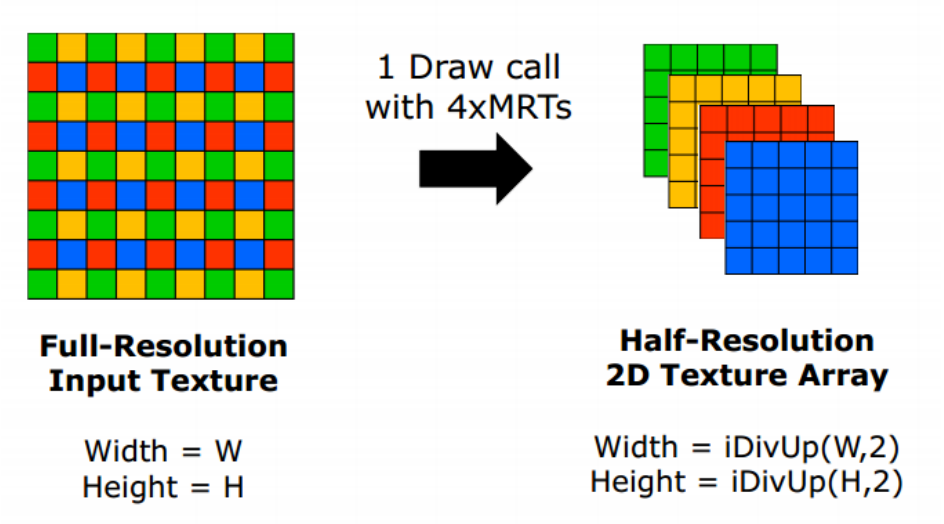

Figure 5. [Step 1] Deinterleaving the input texture.

We start from the full-resolution input texture (in this case 2x2 interleaved) and we **deinterleave** it as shown on Figure 5.

We generate one texture per sampling pattern. In this example with 2x2 sampling patterns, the textures are half-resolution. We render the deinterleaved data using Multiple Render Targets(MRTs) and store them in a half-resolution 2D Texture Array with 4 slices.

This deinterleaving step can be seen as transforming an Array Of Struct into a Struct of Arrays. We have the same input and output information, and just restructure the input.

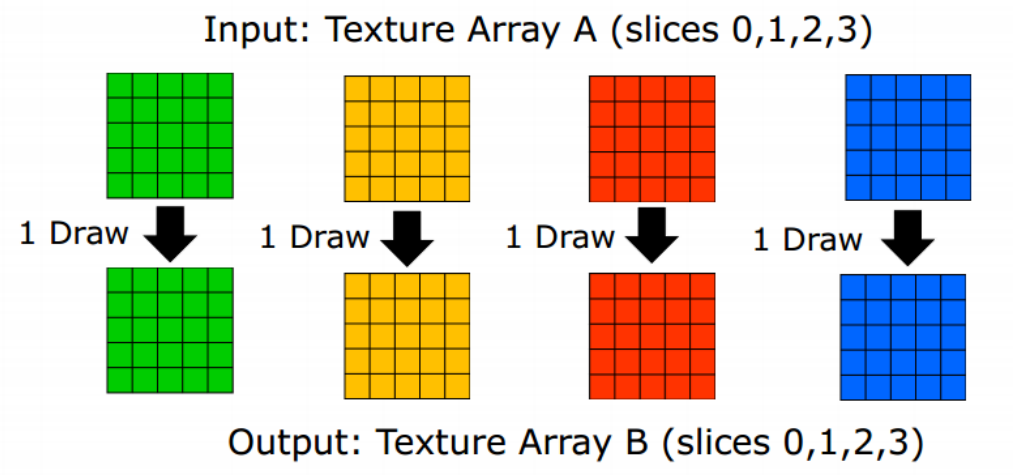

Figure 6. [Step 2] Sampling the deinterleaved data.

Next, we perform one draw call per deinterleaved texture and each of the draw calls has a single sampling pattern. Each of these draw calls is sampling a single slice from texture array A and outputting to a single slice in texture array B.

In the pixel shader for these draw calls, there is no per-pixel jittering being used and adjacent pixels are fetching close-by texels, which is friendly to texture caches.

Besides, because all of the texture slices are low-resolution (half-resolution), there is less memory bandwidth required for transferring the working set from video memory to the shader units, which can help performance dramatically.

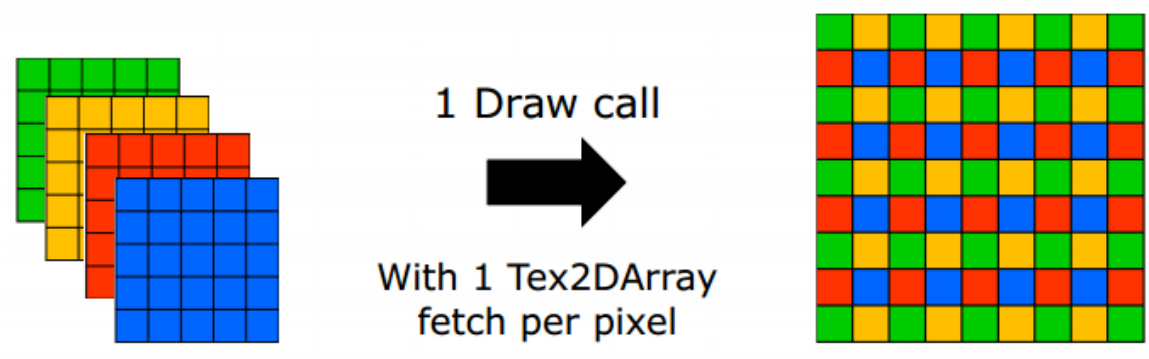

Figure 7. [Step 3] Re-interleaving the output.

Now that we have generated the 4 separated results for each of the sampling pattern, we interleave back the results into a full resolution texture. That is a full-resolution pixel shader pass, with one texture fetch from the texture array B.

This method can be extended to 4x4 interleaving as well, which is what this SDK sample shows. For the deinterleaving step, instead of doing 1 draw call with 4 MRTs, we use 2 draw calls with 8 MRTs. We process the 16 possible sampling patterns and the rest stays the same.

Acknowledgments

The Sibenik model was created by Marko Dabrovic. The AT-AT model used in this sample was created by Brad Blackburn and can be downloaded from http://www.scifi3d.com.

References

[Keller and Heidrich 2001] Alexander Keller and Wolfgang Heidrich. "Interleaved Sampling." Proceedings of the Eurographics Workshop on Rendering. 2001.

[Segovia et al. 2006] B. Segovia, J. C. Iehl, R. Mitanchey, B. Péroche. "Non-interleaved Deferred Shading of Interleaved Sample Patterns." Graphics Hardware 2006.

[Ritschel et al. 2009] Tobias Ritschel, Thorsten Grosch, Hans-Peter Seidel. "Approximating Dynamic Global Illumination in Image Space." I3D 2009.

[Kasyan et al. 2011] Nickolay Kasyan, Nicolas Schulz, Tiago Sousa. "Secrets of CryENGINE 3 Graphics Technology." Advances in Real-Time Rendering Course. SIGGRAPH 2011.

[McGuire et al. 2012] Morgan McGuire, Michael Mara, David Luebke. "Scalable Ambient Obscurance." HPG 2012

[Bavoil and Jansen 2013] Louis Bavoil and Jon Jansen. "Particle Shadows & Cache-Efficient Post-Processing.". GDC 2013.

.