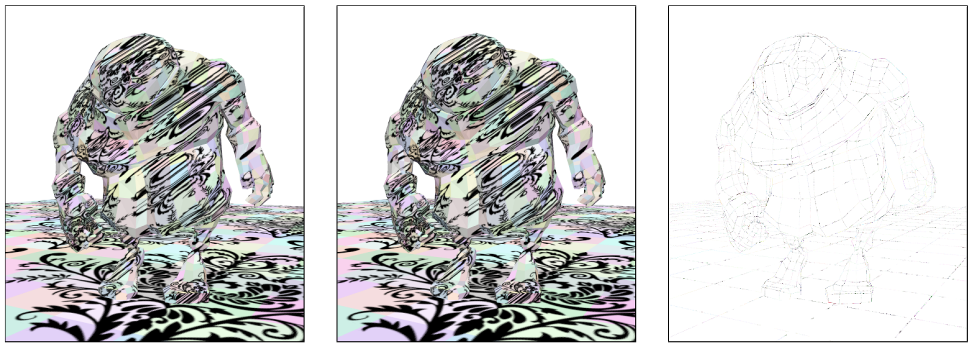
Avoiding Texture Seams by Discarding Filter Taps

Robert Toth
Intel Corporation



Prior art: 568µs Our method: 412µs Difference ×5

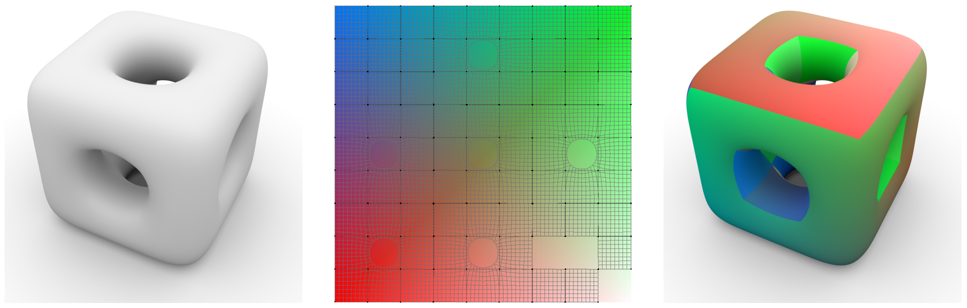
Figure 1. A scene using many individually textured patches. For illustration purposes, each texture has a different base color. The character uses 1450 textures, and the ground plane uses another 1024 textures. In this paper, we present an inexpensive method of filtering textures at seams with wide anisotropic filters, with quality similar to prior more expensive methods.

Abstract

Mapping textures to complex objects is a non-trivial task. It is often desirable or even necessary to map separate textures to different parts of an object, but it may be difficult to obtain high-quality texture filtering across the seams where textures meet. Existing real-time methods either require significant amounts of memory, prohibit use of wide texture filters, or have a high complexity. In this paper, we present a new method for sampling textures which is surprisingly simple, does not require padding, and results in high image quality. The method discards filter taps that extend beyond the texture boundary and relies on multisample antialiasing in order to produce good image quality. Our method is suitable for real-time implementations of rectangular-chart-based assets, such as per-patch texturing (e.g. Ptex).

1. Introduction

Assigning texture coordinates to objects can be a difficult task. It is often impossible to define a single continuous two-dimensional surface that can be wrapped around a



(a) (b) (c)

Figure 2. (a) The surface of this object cannot be mapped to a two-dimensional texture without introducing *seams*, where a neighborhood around a point on the surface is not continuous in texture space. (b) Result of automatic UV unwrapping, overlayed over a gradient texture. (c) Seams in the UV map are clearly visible as discontinuities in the surface color.

three-dimensional object. An example of this is shown in Figure 2. In these cases, multiple textures (possibly packed into an *atlas*) must be used to obtain full coverage of the object's surface. Even in cases where it is possible to use a single texture, it is often more practical to divide the surface into multiple regions with a separate texture for each region, as this vastly simplifies texel density management. In the extreme, each primitive may be assigned a unique texture – the Ptex texturing system [Burley and Lacewell 2008] does this and has been adopted with great success in the movie industry.

Texture sampling near *seams*—where textures meet—must be done with care to avoid rendering artifacts, as the seam should not be visible in the rendered result. Furthermore, bilinear and trilinear texture filtering are progressively being replaced by wider filters, with increasing adoption of anisotropic texture filtering. In these cases, the texture filter footprint may extend far outside of a texture, further complicating the handling of seams.

In this paper, we present a new method to sample textures, which supports wide texture filters while producing invisible seams between multiple textures without complex shader or sampler logic, and without using wide memory-consuming guard bands.

2. Existing Methods

Figure 3 illustrates two fragments in a pixel, each mapped to different textures. The texture filters for each of the two fragments extend outside of the boundaries of the texture mapped to the corresponding fragment. There are several ways to handle this situation—as described next—most of which aim to estimate the color of the filtered region outside of the texture bounds.

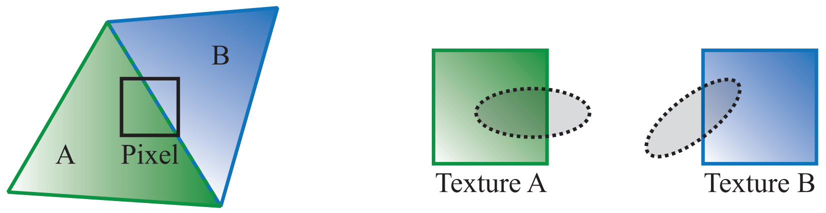


Figure 3. Left: Two adjacent fragments at a texture seam. Right: Wide filter kernels extend outside of each texture. In general, the texture space filter footprint will not have the same shape or size for both fragments.

The Direct3D and OpenGL graphics APIs use *texture address modes*, which specify how the full two-dimensional domain of texture coordinates should be interpreted. The texture address modes include clamping, mirroring, and repeating color data outside of the texture boundaries.

Multiple textures are often packed into a single larger image called an *atlas*. The textures within the atlas usually contain guard bands to allow for mipmapping, wide filters, and shading of points slightly outside of primitives.

Purnomo et al. [2004] map texels to texture edges such that bi- and trilinear filtering at texture coordinates within the surface never requires data from the neighboring textures. They also describe how to use a single-texel-wide border, available in the OpenGL compatibility context, to achieve invisible seams.

The Ptex texture mapping system [Burley and Lacewell 2008] was designed for offline rendering. During the sampling process, neighboring textures are located along with their orientation, so that filter taps outside of the texture bounds (taps with any texture coordinate component outside of the $[0,1]$ interval) sample the adjacent textures. We will refer to this process as *traversal*.

McDonald and Burley [2011] add a guard band as wide as the filter extents at each mip level, to avoid sampling texels outside of the guard band. The contents of adjacent textures are copied into the guard band, thus emulating the result of Ptex in real-time.

McDonald [2013] eliminates the need of a guard band by providing connectivity information to the shader and sampling adjacent textures, similar to Ptex.

Each of the methods above suffer from one shortcoming or another. The Direct3D and OpenGL texture address modes each introduce incorrect data at seams (except for some special cases, such as repeating textures). Traversing multiple textures in the texture sampler requires providing connectivity information and the relative mapping of texture coordinates between textures to the shader, as well as access to all of the neighboring textures. Placing texels at edges ensures that bilinear sampling inside a texture works well, but does not handle wide filters where taps may end up outside of the texture bounds. Adding a guard band to every mip level increases the memory and bandwidth usage of textures significantly due to the extensive duplication of data. Adding guard bands to atlases, sufficiently wide to support anisotropic filters even at coarse mip levels, requires even more memory than individually padding each mip level [McDonald and Burley 2011], since they grow exponentially in size for each finer mip level. Centroid sampling is required to guarantee that the texture footprint center is contained within the texture, both when using guard band methods and when placing texels at edges using a narrow filter.

In practice, most applications use some form of atlas, but with a much thinner guard band than would theoretically be required, and simply accept that a filter kernel may sample invalid data outside of the moderately padded texture region in unlucky cases.

Among the alternatives listed above, our proposed method is most similar to the method proposed by McDonald [2013], which we will refer to as *Traverse* since it visits neighboring textures during sampling. We will use *Traverse* for comparison purposes in the remainder of the paper.

3. A New Method: *Discard*

Our method limits texture filtering at seams, so as to integrate the texture only over the fragment, as opposed to the entire pixel. Thus, we do not try to estimate texture data for taps falling outside of the texture, but rather exclude them from the filtering process. It is assumed that texture edges coincide with geometric edges. We will refer to our method as *Discard*, since we discard filter taps.

Figure 4 (left) illustrates the texture filter footprint of two fragments in a pixel, each mapped to different textures and each extrapolated into the adjacent texture. Previous work strives to compute the filtered combination of the two textures when sampling the texture for each fragment, either by traversal or by otherwise guessing at the neighboring content (e.g., padding or mirroring the texture).

The sampling process of *Traverse* will now be described more formally. Let C_A and C_B be the fragment colors, and let T_A and T_B be the texture functions. Futhermore, let f_A and f_B be the filter functions within the respective texture of each fragment, and $\hat{f}_B$ and $\hat{f}_A$ be the extrapolated filter functions in the neighboring textures. The values

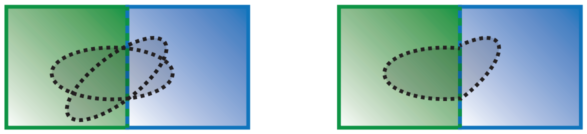


Figure 4. Filtering of the fragments from Figure 3. Left: The texture integral over the entire pixel footprint is estimated for each fragment in previous methods. Right: The texture integral over just the fragment's footprint is estimated using our proposed method, called *Discard*.

C_A and C_B are then:

$$\begin{aligned} C_A &= \int T_A f_A + \int T_B \hat{f}_B, \quad \int f_A + \int \hat{f}_B = 1, \\ C_B &= \int T_B f_B + \int T_A \hat{f}_A, \quad \int f_B + \int \hat{f}_A = 1. \end{aligned} \tag{1}$$

During multi-sampling resolve, the final pixel color P is a weighted sum of the fragment colors, using a coverage-based function w:

$$P = w_A C_A + w_B C_B, \quad w_A + w_B = 1. \tag{2}$$

With low curvature and perspective distortion, it holds that

$$\hat{f}_i \approx f_i. \tag{3}$$

Therefore, both fragments obtain approximately the same filtered color, which also becomes the final pixel color:

$$\begin{aligned} P &= w_A \left(\int T_A f_A + \int T_B \hat{f}_B \right) + w_B \left(\int T_B f_B + \int T_A \hat{f}_A \right) \\ &\approx w_A \left(\int T_A f_A + \int T_B f_B \right) + w_B \left(\int T_B f_B + \int T_A f_A \right) \\ &= \int T_A f_A + \int T_B f_B. \end{aligned} \tag{4}$$

Our main contribution is to introduce a different approximation for *Discard*. We assume that the multisampling filter and the texture filter have a similar distribution:

$$w_A \approx \int f_A, \quad w_B \approx \int f_B. \tag{5}$$

Note that the purpose of both f and w is to low-pass filter the texture and image function, respectively, before sampling at one sample/pixel, and both can thus be considered practical approximations of the theoretically ideal sinc filter. Since they both approximate the same filter, they also approximate each other. With this assumption, it turns out that we can simply ignore the part of the filter that extends outside of the fragments' corresponding textures and avoid approximating $\hat{f}_i$ altogether:

$$C'_A = \frac{\int T_A f_A}{\int f_A}, \quad C'_B = \frac{\int T_B f_B}{\int f_B}, \tag{6}$$

$$\begin{aligned} P' &= w_A C'_A + w_B C'_B \\ &\approx \left(\int f_A \right) \frac{\int T_A f_A}{\int f_A} + \left(\int f_B \right) \frac{\int T_B f_B}{\int f_B} \\ &= \int T_A f_A + \int T_B f_B. \end{aligned} \tag{7}$$

Note that both approximations (Equations (3) and (5)), when reasonably accurate, lead to the same result (Equations (4) and (7)). The difference lies in the required circumstances: previous work requires a good approximation of the filter shape in adjacent primitives, as well as knowledge of the neighboring texture function, while our proposed model instead requires the multi-sampling filter and the texture filter to be similar.

To summarize, instead of estimating a filtered combination of multiple textures, the proposed method lets each fragment limit the filter to only consider the area that falls inside the local texture bounds. This is illustrated in Figure 4 (right) and amounts to a texture address mode which discards any filter taps that end up with a texture coordinate component outside of the $[0,1]$-range. The multi-sample resolve then blends the fragments together according to their relative coverage.

The numerator in Equation (6), $\int Tf$, is obtained by setting the texture address mode to a constant border color of zero and performing sampling as usual. The denominator, $\int f$, is obtained by sampling a texture channel which is 1.0 within the texture, again with a constant border color of zero. The dimensions of this texture channel must match those of the texture being normalized, in order to ensure identical weights f are produced by the sampler hardware.

The overhead of adding the constant texture channel may or may not be significant, depending on the texture formats used and their content. For RGB textures encoded with BC1 (DXT1), there is already an implicit alpha channel with a constant one, so in this case there is no added cost. If, on the other hand, all channels in the texture format contain information, another single-channel texture needs to be added, bound to the pipeline, and sampled. If multiple textures are used, a single normalization factor may be shared for all texture lookups as long as they use the same texture coordinates, sampler settings, resolution, and number of mip levels—-otherwise, multiple normalization textures and/or lookups must be employed. The entire implementation described above is a subset of McDonald's proposed method [2013], with all parts related to sampling the adjacent textures removed.

Since GPUs shade at a granularity of 2×2 pixel quads, so called *helper pixels* with zero coverage may end up discarding all filter taps. Not handling these cases results in a divide-by-zero and a NaN color, which is acceptable for most uses since helper pixels' values are seldom used. For cases where the screen-space derivatives of the texture are used—e.g., for indirect texture lookups—this has to be handled more carefully since finite difference calculations may include the helper pixels' values. A suitable solution is to clamp the user-provided footprint center to the texture extents and perform a bi- or trilinear lookup for these cases.

GPU implementation. Pseudocode of our implementation is shown in Listing 1, which also includes the implementation of McDonald [2013]. Note that *Discard* only

```
1  float3 SampleTexture(float2 uv, int face_id) {
2      float4 c = texture.Sample(sampler_border, float3(uv, face_id));
3  #if defined(MCDONALD_2013)
4      for (int i = 0; i < 4; ++i) {
5          int id = NeighborID[face_id][i];
6          if (id < 0)
7              continue;
8          int mapping = NeighborMapping[face_id][i];
9          float2x3 transform = UVTransforms[mapping];
10         float2 neighbor_uv = mul(transform, float3(uv,1));
11         c += texture.Sample(sampler_border, float3(neighbor_uv, id));
12     }
13 #elif defined(CLAMP_IF_ZERO_COVERAGE)
14     if (c.a == 0)
15         c = texture.Sample(sampler_clamp, float3(uv, face_id));
16 #endif
17     return c.rgb / c.a;
18 }
```

Listing 1. Pseudocode snippet of our implementation and that of McDonald [2013], for textures with the alpha channel filled with 1.0.

executes lines 2 and 17 (and optionally 14–15 to handle helper pixels as described above), while *Traverse* executes lines 2–12 and 17. An example can be found in Appendix A.

4. Results

In this section, we evaluate two aspects of our algorithm: *performance* and *quality*.

Performance. We render the scene in Figure 1 on two systems: one equipped with an NVIDIA GTX680 discrete graphics card with a TDP[1] of 195W, and one with an integrated Intel Iris Pro 5200 graphics processor at a TDP of 47W shared between the CPU and GPU. The scene uses a shader consisting of a texture lookup followed by a very lightweight lighting calculation. The character consists of 1450 textures, and the ground plane consists of another 1024 textures.

The measured execution times at various MSAA sampling rates are presented in Table 1. Timings for sampling rates above one sample/pixel include resolving MSAA surfaces into a single-sampled display surface. As can be seen, the execution times of *Discard* are significantly shorter than those of *Traverse* at equal sampling rates. For this scene, *Discard* with 4× MSAA outperforms *Traverse* without MSAA on both architectures. Similarly, *Discard* with 8× MSAA outperforms *Traverse* with 2× MSAA.

[1]TDP, or "Thermal Design Power", indicates the power consumption of a device.

Device	Method	MSAA sampling rate			
		1×	2×	4×	8×
NVIDIA GTX680	*Traverse*	0.344	0.446	0.485	0.568
(195W TDP GPU)	*Discard*	0.167	0.265	0.301	0.412
Intel Iris Pro 5200	*Traverse*	2.36	2.94	3.25	4.02
(47W TDP CPU+GPU)	*Discard*	0.815	1.34	1.48	2.43

Table 1. Execution times in milliseconds of the scene shown in Figure 1, measured on two architectures. At equal sampling rates, *Discard* is significantly faster than *Traverse* on both architectures.

Quality. Due to the difficulty of choosing a fair ground truth, quality is hard to quantify in a meaningful way. We therefore present multiple data points to assess quality. All images used for quality assessment were generated on an NVIDIA GTX680 graphics card, using 16× anisotropic sampling.

The difference between *Traverse* and *Discard* at various MSAA sample rates is shown in Figure 5 (left). Note that this plot shows the difference between approximations, but does not indicate which of the two methods is responsible for "errors." The two methods grow more similar with an increased sampling rate as the quality

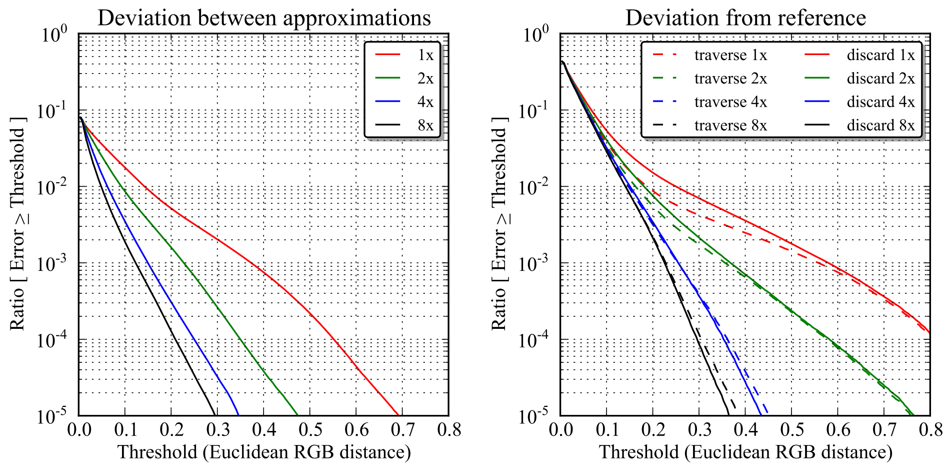


Figure 5. Quality assessment plots of the scene in Figure 1 with a 1000-frame camera orbit and with various MSAA sampling rates. The x-axis is a 2-norm distance in linear RGB space, and the y-axis is the fraction of all pixels that deviate by at least x (lower left is better). Left: Difference between *Traverse* and *Discard*. While the difference diminishes with higher sampling rates, the two methods do not converge to the same image. Right: Deviation from a high-quality reference. At low sampling rates, *Discard* results in quality inferior to that of *Traverse*. With high sampling rates, however, *Discard* results in quality slightly higher than that of *Traverse*, as the MSAA quality surpasses the extrapolation accuracy of *Traverse*.

of *Discard* increases. However, as expected, they do not converge towards the same image, due to the differences between the approximations, as described in Section 3.

To evaluate the error compared to some desirable high-quality reference, we render the scene at high resolution using *Traverse*, and down-sample it by a factor of 8×8 using a Mitchell-Netravali filter [1988] with the typical filter parameters B=1/3, C=1/3. This choice of reference is of significantly higher quality than what is obtained with typical real-time texture filters and multi-sample filters.

Figure 5 (right) shows the deviations of *Traverse* and *Discard* as compared to the high-quality reference. While most of the sources of error are shared between the two methods—namely how polygon internals and geometric silhouettes are filtered—they differ in the quality of texture seams. As expected, *Traverse* performs better at low sampling rates. As the sampling rate increases, the errors of both *Traverse* and *Discard* diminish due to the increased geometric filtering quality. The error of *Discard* diminishes at a faster rate than that of *Traverse*, since texture seams are also improved upon in addition to the silhouettes. Interestingly, at high sampling rates, the quality of *Discard* slightly surpasses that of *Traverse*. The two methods are roughly equivalent at the commonly targeted rate of four samples/pixel. Increasing the sampling rate by a factor of two improves the quality of this scene more than switching from *Discard* to *Traverse*, while costing less in terms of execution time.

5. Discussion

In practice, multi-sampled surfaces are most often resolved into single sampled display surfaces using a box filter. While this is far from the ideal sinc filter, it plays nicely with the color replication inherent in MSAA. Even if the pixel shader output is properly bandlimited, the replication of the shaded value to multiple samples re-introduces higher frequencies into the signal. Applying a good filter to the multi-sampled image would thus over-blur the image, while the box filter provides the desired result (for polygon interiors, that is). This, in combination with its computational simplicity, makes the box filter popular for real-time rendering.

Texture filtering in real-time graphics is usually performed using trilinear interpolation of a mipmap hierarchy at one or a few points, depending on anisotropy. In general, this results in a significantly blurrier signal than would be obtained with a good filter, and even blurrier compared to the box filter used for the MSAA resolve.

Despite the differences between the filters in practice, the proposed technique produces good quality at eight samples per pixel and moderate quality at four samples per pixel. There are two important shortcomings though: errors occur in places where a human observer can more easily detect them (rather than on high-curvature regions, for instance) and texture magnification is flawed due to limitations of current graphics APIs, as discussed next.

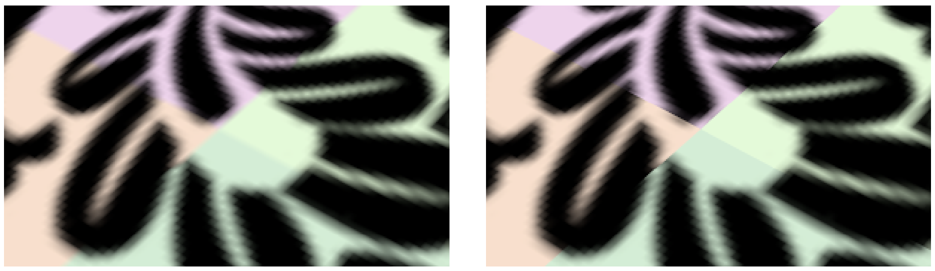


Figure 6. Left: Smooth interpolation at the junction of four textures during magnification. Right: Interpolation artifacts at the very edge of textures, if no data is available in the outermost half-texel-wide band.

Texture magnification. The Direct3D API does not expose any mechanism to seamlessly interpolate texture data all the way to the edges where textures meet. The outermost defined texels reside half-a-texel spacing inside the logical boundary of the texture, and so interpolation at the very edge can only be controlled with texture address modes. The compatibility layer of the OpenGL API includes a mechanism to provide texture borders, i.e., texels residing half-a-texel spacing outside of the logical boundary of the texture. This allows seamless interpolation between textures, but the feature cannot be used with compressed textures and is, therefore, of limited use. This is a limitation of current graphics APIs that affects the quality of the proposed method. The error is most prominent when there is insufficient resolution in the texture and magnification is required, as shown in Figure 6. *Traverse* is unaffected by this issue, since it obtains border information by loading texels from the neighboring textures. Purnomo et al. [2004] propose solving this issue by placing texels at the texture edges. Unfortunately, this has not yet been incorporated into GPUs and the common real-time graphics APIs.

Texture atlases. In real-time applications, multiple textures are often packed into disjoint regions of a single image called an *atlas*. This strategy has been of importance since it reduces the number of state changes necessary to bind different texture images to the hardware graphics pipeline. When textures are packed into an atlas, the sampler does not know where one texture ends and another begins. With large filter footprints, there is always a risk of unintentionally sampling unrelated textures within the atlas at coarse mip levels. To solve this issue, atlases can be replaced by multiple *bindless textures*, i.e., textures that can be accessed from the shader using handles instead of bind slots, which makes packing superflous.

Post-processed antialiasing. In recent times, there has been a trend of replacing multi-sample anti-aliasing by post-process antialiasing methods [Jimenez et al. 2011]. These methods detect high contrast regions and try to recreate a smooth gradient. As long as the post-process antialiasing method relies solely on color, it can, in theory, be applied to texture content as well. In practice, the nature of discontinuities across texture seams is different from the straight, high-contrast edges usually targeted by these filters, and the resulting quality is sub-optimal. See the supplemental material for an image rendered using *Discard* at a single sample per pixel, post-processed using FXAA.

6. Final Remarks

A number of factors have led to the proposal of our technique: multi-sampling anti-aliasing is widely adopted; anisotropic texture filtering is more or less expected in modern applications; more and more GPUs support some form of bindless textures. At the same time, asset production costs are higher than ever, and parametrization-free texturing systems like Ptex have successfully simplified the production process in the movie industry. All of the above are pieces of a puzzle that, once put together, hints at the proposed texturing method.

It seems odd that seamless interpolation is only attainable by forgoing much of the hardware acceleration for address calculation and mip level selection, and we would like to see the graphics APIs and hardware evolve to allow seamless interpolation where textures meet. Whether this goal is best reached by changing the location of texels, availability of texture borders for compressed textures, or something else entirely, is yet to be seen.

Acknowledgements

Thanks to Jacob Munkberg, Magnus Andersson, Jon Hasselgren, Jim Nilsson, and Tomas Akenine-Möller for assisting with figures and for valuable feedback, and to Chuck Lingle and Tom Piazza for supporting this research. Special thanks to Stephen Hill for reading countless revisions and for making the content comprehensible.

References

BURLEY, B., AND LACEWELL, D. 2008. Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008*, Eurographics Association, Aire-la-Ville, Switzerland, 1155–1164. 91, 92

JIMENEZ, J., GUTIERREZ, D., YANG, J., RESHETOV, A., DEMOREUILLE, P., BERGHOFF, T., PERTHUIS, C., YU, H., MCGUIRE, M., LOTTES, T., MALAN, H., PERSSON, E., ANDREEV, D., AND SOUSA, T. 2011. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*, ACM, New York. 99

MCDONALD, JR, J., AND BURLEY, B. 2011. Per-face texture mapping for real-time rendering. In *ACM SIGGRAPH 2011 Talks*, ACM, New York, SIGGRAPH '11, 10:1–10:1. http://doi.acm.org/10.1145/2037826.2037840. 92, 93

MCDONALD, JR, J., 2013. Eliminating texture waste: Borderless ptex. GDC 2013. https://developer.nvidia.com/gdc-2013. 92, 93, 95, 96

MITCHELL, D. P., AND NETRAVALI, A. N. 1988. Reconstruction filters in computer graphics. *SIGGRAPH Comput. Graph. 22* (June), 221–228. http://doi.acm.org/10.1145/378456.378514. 98

PURNOMO, B., COHEN, J. D., AND KUMAR, S. 2004. Seamless texture atlases. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, ACM, New York, SGP '04, 65–74. http://doi.acm.org/10.1145/1057432.1057441. 92, 99

A. Appendix: Operation Example

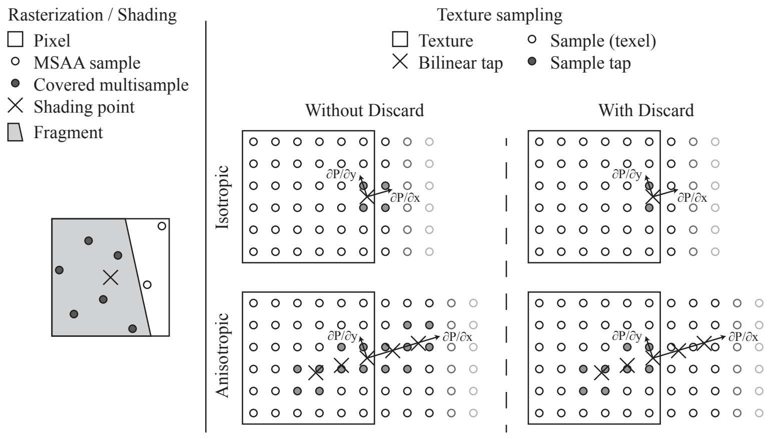


Figure 7. Left: a fragment is rasterized, resulting in a pixel shader invocation. Right: The texture sampler computes one (top) or a few (bottom) bilinear tap locations depending on anisotropy. In practice, these are actually trilinear taps, but are shown as bilinear for simplicity. The sample taps used to service the bi-/trilinear taps are shown in grey. Without *Discard*, sample taps outside of the texture are weighted into the returned color. With *Discard*, sample taps outside of the texture get zero weight, and do not affect the returned color.

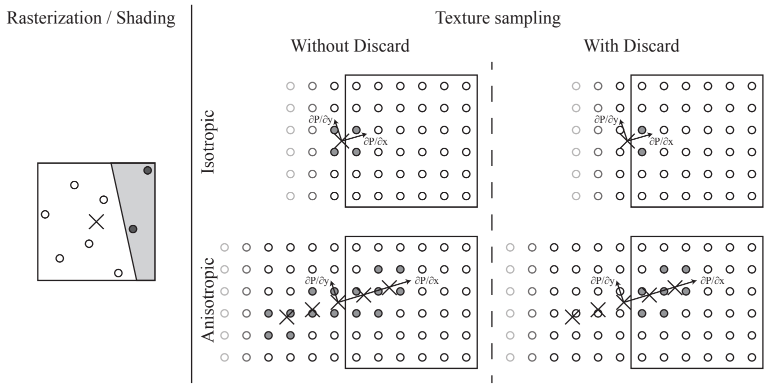


Figure 8. A fragment complementary to that in Figure 7. The texture coordinate lies outside of the texture bounds, but mipmapping ensures that there are sample taps within the texture.

Figure 7 shows an example of a texture lookup operation. The shader issues a texture lookup just inside the texture edge. For shallow angles, a single trilinear tap is determined by the texture sampler, while a set of trilinear taps are determined for steep angles. These trilinear taps are converted to a set of sample taps with corresponding weights. With *Discard*, any sample tap outside of the texture gets a zero weight. The remaining sample tap weights are renormalized to account for the discarded sample taps.

Figure 8 shows an example where the texture lookup location lies outside of the texture bounds. This is common when rendering with MSAA without centroid sampling. In most cases, at least one sample tap will fall within the texture bounds due to the normal way mip levels are selected. However, there are cases for which this is not true, as noted in Section 3.

Index of Supplemental Materials

The supplemental materials include images and videos of *Traverse* and *Discard*, as well as a reference image and an error video, all depicting the scene in Figure 1:

readme.txt File containing this description of the supplemental materials.

discard_*x.png Rendering using *Discard* at different MSAA sampling rates.

traverse_*x.png Rendering using *Traverse* at different MSAA sampling rates.

reference.png High resolution (3840×4320) rendering, downsampled with a Mitchell-Netravali filter to the same resolution as the other images (480×540).

fxaa_discard_1x.png FXAA 3.1 applied to a single-sampled rendering using *Discard*.

discard.mkv Animation showing temporal stability of *Discard* at different MSAA sampling rates.

traverse.mkv Animation showing temporal stability of *Traverse* at different MSAA sampling rates.

error.mkv Animation comparing error of *Discard* and *Traverse* with respect to the high-resolution downsampled reference.

Author Contact Information

Robert Toth
Intel Corporation
Scheelevägen 19
Lund, SE-22370
robert.m.toth@intel.com

Robert Toth, Avoiding Texture Seams by Discarding Filter Taps, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, 91–104, 2013
http://jcgt.org/published/0002/02/07/

Received:	2013-03-17		
Recommended:	2013-06-01	Corresponding Editor:	Stephen Hill
Published:	2013-12-06	Editor-in-Chief:	Morgan McGuire