Anisotropic Texture Filtering using Line Integral Textures

Abstract

In real-time applications, textured rendering is one of the most important algorithms to enhance realism. In this work, we propose a line integral texture filtering algorithm that is both fast and accurate in improving filtered image quality. In particular, we present an efficient shader implementation of the algorithm that can be compared with 16x hardware anisotropic filtering. The authors show that the quality of the result is similar to 16x anisotropic filtering, but that the line integral method requires fewer samples to function, making it a more efficient rendering option.

I.3.7 [Texture Filtering]: Computer Graphics—Real-time Rendering

Keywords: texture, filter, anisotropic, real-time rendering

1 Introduction

Textures are a popular way to add detail to 3D scenes. However, because they are an approximation of actual geometry, they must be filtered to achieve satisfactory results. Traditional methods to filter textures include nearest neighbor, bilinear, bicubic, trilinear, and anisotropic filtering. These methods vary in reconstruction quality, particularly when the texture in question is being viewed at an extreme angle.

Nearest neighbor filtering suffers extreme aliasing, especially at glancing angles or when minified. Bilinear filtering improves the situation by taking four samples, and modern graphics hardware optimizes it to be nearly as fast. However, when minifying, aliasing reemerges. Trilinear filtering effectively allows for the bilinear filter’s footprint to cover any size square area in texture space, resolving aliasing during minification. However, mipmaps suffer from overblurring and/or undersampling when the texture is viewed at an angle, because the necessary sampling area is anisotropic.

Traditional hardware anisotropic filtering algorithms improve on these techniques by taking multiple samples inside this anisotropic sampling region. Although this approach is simple to implement, the result is merely an approximation. Also, for good results, oftentimes many samples must be taken. For some angles, texture quality suffers unavoidably.

The proposed line integral method addresses these issues. The line integral method computes a more accurate average sample inside the texture footprint, thus leading to more accurate visual results. The line integral method rivals the quality of even the best hardware anisotropic filtering but tends to require fewer texture samples than traditional anisotropic filtering, saving texture bandwidth and therefore rendering time.

The algorithm, implemented in a GLSL shader, already provides similar performance and quality to the algorithms of highly optimized current hardware. If the line integral method were also to be implemented in hardware, it would perform faster than classic anisotropic filtering, due to its greater efficiency.

2 Previous Work

Mipmapping: Mipmapping, also known in the context of two-dimensional texturing as “trilinear filtering”, has been used since 1983 [Williams 1983]. Mipmapping works by sampling from a prefiltered texture image. By choosing an appropriate mipmap level, there will be less aliasing of high frequencies in the texture.

Although mipmapping solves the aliasing problem inherent to texture minification, unfortunately, the prefiltered images are not directionally aware, and so when the minified texture is sampled anisotropically, the texture’s quality suffers either through overblurring or undersampling.

Summed Area Tables: Another approach introduces the summed area table to compute areas inside a sampling region [Crow 1984]. The algorithm works by adding and subtracting rectangular sums.
The summed area table can easily compute a rectangular region, but cannot compute rectangles that are not aligned with the axis of the table nor arbitrary quadrilaterals. In addition, precision issues prevent the table from being practical for 32 bit textures, especially for large image sizes.

Anisotropic Filtering: Traditional hardware anisotropic filtering improves upon standard texture filtering, usually being used in conjunction with trilinear or standard bilinear filtering. Anisotropic filtering takes multiple samples from within the sampling area, allowing the hardware to average multiple samples together for a more accurate result [NVIDIA 2000].

Unfortunately, the algorithm is an approximation to an actual average sample, and thus a large number of samples must be taken for good quality. The proposed algorithm typically requires fewer samples than hardware anisotropic filtering, and is capable of producing better results.

Other: Some other, such as Texture Potential Mipmap [Cant and Shrubsole 2000] and Fast Footprint Mipmap [Huttner and Strasser 1999], are hybrids of summed area tables and mipmaps (and variations on these techniques). By combining prefiltering and summed area tables, these authors were better able to anisotropically sample textures. However, both summed area tables and mipmaping remain inherently limited to axis-aligned sampling rectangles. The best known of these approaches is the RIP-map structure, which prefilters images in each direction. This allows for sample footprint to run in the “U” or “V” texture directions, but still remains ultimately orthogonally confined.

Others have attempted to refine the basic technique of classic anisotropic filtering by weighting and choosing their samples carefully. Such techniques include “Feline” [McCormack et al. 1999], and the Elliptical Weighted Average [Greene and Heckbert 1986]. These can provide very compelling results, because they weight samples nearest the pixel center more strongly, thus more accurately approximating a sinc function.

Still others seek to invent new hardware to solve the anisotropic filtering problem. “Texram” [Schilling et al. 1996], [Shin et al. 2006], a dedicated chip for anisotropic filtering, did something very similar to present-day hardware anisotropic filtering.

3 Line Integral Texture Filtering

For practical purposes, the world can be thought of as an analog function—defined everywhere; everywhere continuous. When a digital picture is taken, the real world is quantized into a two-dimensional collection of samples from this analog signal. Equivalently, when a texture is created from scratch, the imagined world is sampled in the same way—being reduced to a finite number of samples. Ideally, in the application, we would like to represent the texture at that location, the pixel’s color would only be accurate at that one location—other positions on the pixel would be inaccurate. Of course, a pixel can only display one color, so the ideal color is an average of all the possible samples under the pixel footprint (the quadrilateral a pixel’s outline subtends when projected into texture space). This is almost never the case for a single sample.

3.1 Motivation

The one sample the pixel nominally takes in the second sampling step is often a poor representation of the now continuously defined texture. Even if that sample were representative of the texture at that location, the pixel’s color would only be accurate at that one location—other positions on the pixel would be inaccurate. Of course, a pixel can only display one color, so the ideal color is an average of all the possible samples under the pixel footprint (the quadrilateral a pixel’s outline subtends when projected into texture space). This is almost never the case for a single sample.

3.2 Algorithm

A texture is sampled again at each pixel it subtends, before being reconstructed on the screen as a pixel (itself a 2D rect function).

Let texture be sampled on the screen with the pixel itself. The line integral method computes these operations into a single step, so that the texture reconstruction and the pixel reconstruction happen all at once, ensuring that the pixel’s final color is representative of the ideal colors inside a pixel footprint.

The line integral method defines a new kind of texture—the line integral texture—constructed from an original, ordinary source image. An example of a texture, its line integral texture, and a depiction of the following algorithm may be found in Figure 2.

Each texel in the line integral texture corresponds to the sum of all texels below it in the same column in the original source image. I.e., for a texture M by N pixels, and a sampling function f(m,n), then point P = m,n in the line integral texture has a value:

\[ S(x, y) = \int_0^x \int_0^y f(P.x, P.y) \, dy \]

Note that because standard byte textures may only hold 256 possible values per-channel, whereas a given sample could store the accumulated values of hundreds or a few thousand pixels, the line integral texture must be floating point. This gives the proposed method the same memory footprint as a summed area table. However, unlike classic summed area tables, in which precision becomes a problem due to integrating over a very large area, because we only integrate over a one dimensional region, the spread and error of values in the line integral texture is not extreme. In practice, single precision is far sufficient for the line integral texture. In addition, because the line integral texture only stores one-dimensional integrals, arbitrary sampling quadrilaterals may be computed.

The algorithm continues by computing each pixel’s footprint in texture space. An edge-walking algorithm then steps across the pixel footprint, finding the total texel sum along each column by sampling from the line integral texture. A sum from a column may be found as the difference of two samples from the line integral texture. Once the samples are accumulated, the algorithm divides the final sum by the number of texels underneath the pixel footprint. The resultant color retrieved with the line integral method is then an accurate average color of the original texture under the pixel footprint.

3.3 Implementation Notes

The line integral method is implemented in GLSL. By contrast, anisotropic filtering is implemented in hardware. This makes direct comparison of performance impossible.
Figure 2: Left: For a given point “A”, the line integral texture will store the sum of all the texels in the original image below “A” in the same column. Sums of any vertical line segment may be found by subtracting samples from one another (for example, the sum between points “B” and “C” is equal to the sample at “B” minus the sample at “C”). By using this technique, we can find the sum under any given sampling quadrilateral (grey). Center: A source image. Right: The line integral texture created from the source image, tonemapped to fit in the visual range.

To calculate the pixel footprint in texture space, we had to calculate the per-pixel derivative of the texture coordinate. However, the built-in GLSL functions “dFdx(...)” and “dFdy(...)” operate on four-fragment areas simultaneously. This causes four fragments to share the same derivative value. To correct this, we implemented a two-pass algorithm, wherein the texture coordinates are read out into a floating-point framebuffer in a prepass, being fed into the line integral algorithm. We find that this creates more accurate results.

We found that hardware bilinear interpolation of floating point textures is done with 8-bit precision, yet the algorithm depends on taking accurate interpolated samples from a floating point texture. Sampling from areas close together in texture space would return the same texture value, leading to a sum of zero. In practice, this manifested itself as 256 black bands between individual texels, instead of a smooth gradient. We corrected this issue by taking four nearest neighbor samples and executing the bilinear interpolation manually in the shader.

We present framerate benchmarks to demonstrate that even with all these handicaps, the line integral method still produces realtime framerates. However, we provide texture sample counts as a more realistic comparison of performance.

On our testing machines, there is no extra performance penalty for sampling from a 32 bit texture as opposed to an 8 bit texture. We tested this by sparsely sampling over a large texture, so as to avoid caching optimization. Thus, it makes sense to compare sample counts between 16x hardware filtering and the line integral method, because each sample takes the same amount of time to retrieve.

4 Results

4.1 Comparison to Ground Truth

In the controlled environment of the simple plane demo, we show how the algorithm’s accuracy compares with standard techniques and with a ground-truth rendering, established with a ray-tracer using 256 random samples per pixel (Mental Ray in Maya 2011).

The MSE and MS-SSIM readings demonstrate that the line integral algorithm provides better quality than standard mipmapping, showing that the algorithm functions as a solution to anisotropic texture reconstruction. Quantitatively, the algorithm’s quality exceeds that of hardware anisotropic filtering. We attribute most of the error in the line integral algorithm to our handling of polygon edges, which could be improved. However, even with these unnecessarily large errors, the line integral method regularly beats hardware filtering.

Unfortunately, no ground truth reference was available for the realtime engine, so no mean squared error results were computed—instead, we direct the reader to the accompanying video, which demonstrates the qualitatively the practical effectiveness of the line integral method.

4.2 Demonstration of Performance

To demonstrate the algorithm’s usefulness in real-time scenarios, we use two heuristics.

<table>
<thead>
<tr>
<th>Method</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>16x Anisotropic</td>
<td>27.133</td>
<td>6.593</td>
<td>50.090</td>
</tr>
<tr>
<td>Line Integral</td>
<td>20.664</td>
<td>7.384</td>
<td>35.661</td>
</tr>
</tbody>
</table>

Table 1: MSE metric for measuring the error of the proposed algorithm versus hardware 16x filtering.
Table 2: MS-SSIM metric for measuring the performance of the proposed algorithm versus hardware 16x filtering.

<table>
<thead>
<tr>
<th>Method</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>16x Anisotropic</td>
<td>0.380</td>
<td>0.375</td>
<td>0.515</td>
</tr>
<tr>
<td>Line Integral</td>
<td>0.987</td>
<td>0.982</td>
<td>0.676</td>
</tr>
</tbody>
</table>

Table 3: Average framerates of the proposed algorithm in software versus traditional rendering algorithms in hardware

<table>
<thead>
<tr>
<th>Method</th>
<th>Average frame rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mipmap</td>
<td>267.7 fps</td>
</tr>
<tr>
<td>16x-Anisotropic</td>
<td>265.1 fps</td>
</tr>
<tr>
<td>Line Integral</td>
<td>119.1 fps</td>
</tr>
</tbody>
</table>

We begin by directly comparing the framerates of the application using different texture filtering modes. The results are summarized in the following table:

The results in the table clearly demonstrate that the line integral method is already easily fast enough for realtime use in an actual game engine. However, recall that, due to implementation specifics (see 3.3, Implementation Notes), the proposed method is disadvantaged in this comparison because it is not implemented in hardware and because it must take four times as many samples due to inaccurate hardware bilinear interpolation.

To provide a more adequate comparison, in our second heuristic, we compare the number of texture fetches each algorithm requires. Texture sampling is the largest bottleneck in almost any shader; by comparing the number of samples each algorithm requires, we gain a better measure of how the algorithms’ performance would compare if they were to be implemented similarly. We visualize the results in Figure 5. Green tinted samples represent areas where our algorithm uses fewer texture samples than does 16x anisotropic filtering, while red tinted samples represent areas where our algorithm uses more. Grey tinted pixels contain magnified texels, and so are irrelevant to both techniques. As the figure shows, our algorithm tends to use fewer samples than 16x anisotropic filtering. This result suggests that, because the line integral method requires fewer samples, it would outperform classic anisotropic texture filtering if they were both implemented in hardware.

4.3 Further Analysis

These quantitative quad demo MSE and MS-SSIM results demonstrate that, in controlled environments, the algorithm produces very good results comparable to 16x anisotropic filtering. The algorithm is robust and fast enough to function in a realtime game engine. In the game engine, the line integral method visually matches the quality provided by 16x anisotropic filtering—but, tellingly, the line integral method uses fewer texture fetches if the scene is considered as a whole (again, see 5). Therefore, in theory, if the line integrals method and the traditional anisotropic texture filtering method were implemented in a similar framework, the line integrals method would perform faster than the anisotropic texture filtering method.

The line integral method provides the greatest gain in efficiency when the sampling quadrilaterals are anisotropic in the same direction as the line integral texture. Future work would include the use of two line integral textures running at right angles to each other; the algorithm would then select the best approach given the characteristics of the pixel footprint.

5 Conclusion

We have presented a novel algorithm for sampling and reconstructing textures that is both efficient and practical in real-time applications by the application of a new data-structure, the line integral texture. The line integral method frequently uses fewer samples per pixel than hardware anisotropic filtering, and can provide equivalent or superior image quality. The line integral method, implemented in a GLSL shader, already provides realtime performance despite a number of implementation difficulties. If implemented in hardware, its performance would exceed that of current, state-of-the-art anisotropic filtering techniques.

Acknowledgements

The images used in this work were either generated by the authors or taken from the public domain.

References


Greene, N., and Heckbert, P. S. 1986. Creating raster omni-max images from multiple perspective views using the elliptical


