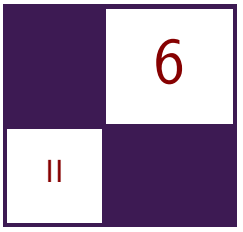

Wire Antialiasing

Emil Persson

6.1 Introduction

There are many sources of aliasing in rendered images. The two most common culprits are geometric edges and shading. Historically these sources of aliasing have been resolved by Multi-Sample Antialiasing (MSAA) and mip-mapping with trilinear filtering respectively. With mip-mapping and trilinear filtering, which were supported on consumer-level hardware even way back in the 1990s, textures on surfaces were essentially free from aliasing. In the early 2000s, as consumer-level hardware gained MSAA support, the remaining problem of edge aliasing, often referred to as "jaggies," was more or less a solved problem. Games of this era could then be relatively aliasing free since the geometric detail was limited enough that MSAA effectively eliminated all geometric aliasing, and shading was typically simple and low frequency and did not introduce any additional aliasing on top of the surface texture. The main exception was alpha-tested objects, such as fences and foliage, which sorely stood out in an otherwise relatively aliasing-free environment.

As games have adopted increasingly more sophisticated lighting models and with geometric density constantly on the rise, aliasing has unfortunately made a strong comeback in modern games. Mip-mapping alone no longer fully solves the shader aliasing problem. Complex lighting introduces aliasing where the mip-mapped textures alone exhibit none. In particular the specular component tends to cause lots of aliasing. This field is poorly researched and only a few approaches exist to properly deal with the problem. The most notable work here is LEAN mapping [Olano and Baker 10]. On the geometry side we are getting increasingly denser geometry, and as geometry gets down to the sub-pixel level, MSAA is no longer sufficient.

Much research remains to be done to solve these problems once and for all. This chapter does not present a final solution to all these problems; however, it presents a technique for solving one specific, but common, subset of geometric aliasing in games, namely that of phone wires and similar long and thin objects.

While the technique has been dubbed Wire Antialiasing, it applies to any object that can be decomposed to a set of cylindrical shapes. This includes wires, pipes, poles, railings, light posts, antenna towers, many types of fences, and even grass if represented as actual geometry rather than as alpha textures. Tree trunks and branches may also be cylindrical enough to work. These are all common elements in games and frequent sources of aliasing.

6.2 Algorithm

6.2.1 Overview

The problem with very thin geometry, such as wires, is that it tends to degenerate into a set of flickering and disconnected pixels when it gets sub-pixel sized. Why is that? In technical terms, what happens is that the visibility function gets undersampled. In plain English, the wire simply misses the pixels. A pixel only gets shaded if the geometry covers the pixel center. If the wire is thinner than a pixel, chances are that at some points it will simply slide in between two pixel centers and end up not shading any of them. In this case there will be a gap. The thinner the wire gets, the worse the problem gets. However, if the geometry is wider than a pixel, we are guaranteed to have a continuous line with no gaps. The problem thus occurs when geometry goes sub-pixel sized. The idea of this technique, then, is to simply keep the wire one pixel wide or larger. To emulate a sub-pixel coverage, we simply fade away the wire by outputting the coverage to alpha and applying blending.

What about MSAA? Unfortunately, MSAA does not solve the problem. It alleviates it somewhat, but the improvement is rather marginal. With 4× multisampling, the visibility function is now sampled at twice the rate horizontally and vertically. In other words, the wire can now be about half a pixel wide (depending on sample locations) without risking missing all samples somewhere. This is not much of an improvement, and all we have accomplished is to push the problem into the distance. We can now have the wire twice as far away before the problem occurs, or have a half as thin wire, but that is all. Actually, when we enable MSAA, we normally want to eliminate "jaggies." If we have a half-pixel wide wire, MSAA may keep it continuous in this case; however, it will be jagged, because it only hits a single sample per pixel. There is not enough resolution to estimate the coverage, but it simply boils down to more or less a binary on/off per pixel.

While MSAA does not by itself solve the thin geometry problem, it is still valuable for this technique. While we are guaranteeing gap-free wires, we are not producing smooth antialiased results as such. In fact, the wires will be jagged by default. So we use MSAA for what it excels at, namely removing those jaggies. So despite increased visibility function sampling rate with MSAA enabled, we still limit wires to a one pixel width and keep the additional resolution for MSAA

to estimate coverage and eliminate the jaggies, just like MSAA normally does. Our technique is thus independent of the MSAA mode. It can also run without MSAA, but the wires will then look as jagged as (but not worse than) the rest of the scene.

6.2.2 Method

The wire is represented in the vertex buffer much like one might normally design a wire, with the exception that we need to be able to vary the wire's radius at runtime, so we do not store final vertex positions but instead store the center of the wire. The final vertex position will then be computed by displacing the vertex along the normal. For this we provide a wire radius, in world-space units. This would typically be a constant for wires and would be best passed in the constant buffer, but for some types of objects (such as antenna towers, light posts, and grass straws), it could make sense to store a radius value per vertex instead.

The first step is to estimate how small a radius we are allowed without violating our minimum one pixel width requirement given the current vertex position in the view frustum. This scales linearly with the w-value of the transformed vertex, depending on the field of view (FOV) angle and render-target resolution. With a projection matrix computed the usual way with a vertical FOV, a constant scale factor can be computed as in equation (6.1):

$$PixelScale = \frac{\tan(FOV/2)}{height}. \tag{6.1}$$

This value can be computed once and passed as a constant. The radius of a pixel-wide wire is then given by multiplying by the vertex's w-value. The w-value can be found by doing a dot product between the vertex position and the last row of the view-projection matrix (or column, depending on the matrix convention used). Once we know the radius of a pixel-wide wire, we can simply clamp our radius to this value to ensure our wire is always at least one pixel wide. The shader code for this is in Listing 6.1.

While adjusting the radius guarantees that we get gap-free wires, the result is inevitably also that the wires will appear wider and wider as they go farther into the distance if we do not also take the original unexpanded wire's pixel coverage into account. Compare Figure 6.1 and Figure 6.2 for an illustration of this effect. What we need to do is to compensate by computing the coverage of the real unexpanded wire and fading the contribution accordingly. This is what will give the wire its natural and alias-free appearance. This can be accomplished by outputting the coverage value to alpha and enabling alpha blending. As shown on the last line in Listing 6.1, the coverage fade factor is simply the original real radius of the wire divided by the expanded radius. In other words, if the radius was expanded to twice the size to cover one pixel, then the wire is half a pixel wide and the coverage consequently 0.5. As the wire gets farther into the distance, it

```
// Compute view-space w
float w = dot(ViewProj[3], float4(In.Position, 1.0f));

// Compute what radius a pixel-wide wire would have
float pixel_radius = w * PixelScale;

// Clamp radius to pixel size.
float new_radius = max(Radius, pixel_radius);

float3 position = In.Position + radius * In.Normal;

// Fade out with the reduction in radius versus original.
float fade = Radius / new_radius;
```

Listing 6.1. Computing radius and fade factor.

Figure 6.1. Original wire. Note how the wire is aliased and full of holes despite 4× MSAA being enabled.

will now appear fainter and fainter, maintaining the appearance of a thin wire, as illustrated in Figure 6.3. Comparing Figure 6.3 to Figure 6.1, it can be seen that the wires look identical, except for the fact that Figure 6.3 does not suffer from aliasing artifacts.

6.2.3 Lighting

The technique as described so far works fairly well; we have eliminated the geometric aliasing, but unfortunately thin wires also tend to suffer from shading aliasing as the lighting function gets sampled very sparsely and pseudo-randomly. When the wire reaches pixel size, any texture sampling would likely go down to the lowest mip-map and get uniformly colored by default; however, the normal

Figure 6.2. Expanded wire. Note that the wire is now free from gaps and aliasing, but appears unnaturally thick in the distance.

Figure 6.3. Final wire. Wire now appears natural and alias free.

can go from pointing straight up to straight down over the course of a single pixel, potentially resulting in severe aliasing from the lighting. The simplest approach to deal with this problem is to choose a soft lighting model that results in a minimum amount of aliasing, such as Half Lambert [Valve 08]. For many wires this could be good enough as it blends fairly well into an environment otherwise lit by more physically correct models. However, even with this simple model (and even more so for more complex ones), it is crucial to enable centroid sampling on the interpolators. The error introduced from sampling outside of the wire is simply far too great.

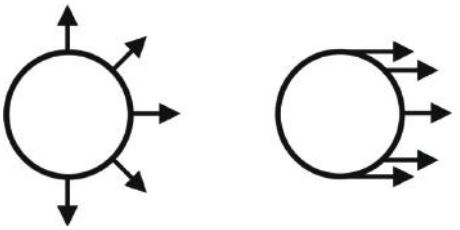

Figure 6.4. Left: Original normals. Right: Normals by the time the wire goes sub-pixel, assuming the viewer is to the right of the wire.

If sticking to Half Lambert is not enough, there are a few different approaches for dealing with this. We have tried a couple of methods that both work relatively well and can also be combined. Both involve fading over from our regular lighting model into something uniformly colored over some distance before reaching pixel size. The first method is simply computing our lighting as usual, then computing the Half Lambert as well, and then fading between these two. In our test scene a good fade range was from about 4 pixels wide and fully faded over to Half Lambert as it reached a single pixel. This is not based on any science but purely from testing various ranges and subjectively selecting what looked best for us. Depending on your lighting model and parameters, this could be tweaked differently. The advantage of this method is that it is straightforward and simple and tends to work fairly well.

The other method we tried involves flattening the normals over the distance. The thinner the wire gets, the more we are bending the normals toward the center normal from the viewer's point of view. As the wire goes down to a single pixel width, the normal will be entirely flat, and consequently result in a uniform lighting result. (See Figure 6.4.)

Since all this does is modify the input normal, this method is compatible with any lighting model, although results may vary.

6.2.4 Use with Deferred Shading

Deferred shading typically has a single global lighting model. This could make it tricky to fade between a soft light model, such as Half Lambert, and the engine's regular lighting model. The most straightforward approach is to simply render wires in a forward pass after the deferred lighting pass. The blended part of the wire (i.e., where it is smaller than a pixel wide) will have to be rendered as a translucent object anyway because of the blending. Typically wires take up a very small amount of screen space, so rendering the entire wire, including fully opaque parts, would normally have a minimal impact on performance. In the case where it is desirable to render opaque parts into the G-buffer, the normal bending approach above integrates very well as all it does is modify the input

normal. The bent normal can then be written to the G-buffer and shaded as part of the regular deferred lighting pass.

6.2.5 Use with FXAA

This technique works very well together with MSAA, where MSAA takes care of all the jaggies, just like it is designed to do. But not all games use MSAA anymore; a whole bunch of filtering-based approaches to antialiasing have been used recently [Jimenez et al. 11a] and several have gone into shipping games. Arguably the most popular ones are FXAA [Lottes 09] and MLAA [Reshetov 09, Jimenez et al. 11b]. We have tested this technique in conjunction with FXAA and found that FXAA became nearly ineffective on the pixel-wide wires left by this technique. Consequently, as a workaround, the wire needs to be expanded somewhat for FXAA to take effect. Fortunately, as little as about 1.3 pixels width is enough for FXAA to pick up the edges. Somewhat wider wires do result in a somewhat lower quality overall, but not too bad. Other similar techniques, such as MLAA, may be better suited for dealing with pixel-thin wires, but the author has not verified that this is the case.

6.3 Conclusion and Future Work

A technique has been presented that effectively deals with aliasing on a specific (but frequently occurring) subset of aliasing-prone geometry. This is a welcome tool to reduce aliasing in games, but it does not solve the entire problem space of aliasing in real-time rendering. Much research is still needed. An obvious next step would be to explore ways to extend this technique to other shapes than cylinders. We believe extending it to rectangular shapes such as bricks and planks should be relatively straightforward and could work fundamentally the same, with the exception that we need to take view orientation into account for estimating size in terms of pixels. From there it may be possible to solve the staircase. Stairs are a frequent source of aliasing artifacts in games, as the top and side view of the steps tend to get lit differently, resulting in ugly Moiré patterns when the steps get down to pixel size.

Solving all these special cases may be enough for games, but ideally it would be desirable to solve the general case, where a model with any amount of fine detail and thin sub-pixel sized triangles could be rendered in an alias-free manner, without resorting to super-sampling, and preferably without a preprocessing step.

Bibliography

[Jimenez et al. 11a] Jorge Jimenez et al. "Filtering Approaches for Real-Time Antialiasing." SIGGRAPH 2011 Course, Vancouver, Canada, August, 2011. (Available at http://iryoku.com/aacourse/.)

[Jimenez et al. 11b] Jorge Jimenez, Belen Masia, Jose I. Echevarria, Fernando Navarro, and Diego Gutierrez. "Practical Morphological Antialiasing." In *GPU Pro 2*, edited by Wolfgang Engel, pp. 95–120. Natick, MA: A K Peters, Ltd., 2011. (See also http://www.iryoku.com/mlaa/.)

[Lottes 09] Timothy Lottes. "FXAA." White Paper, Nvidia, http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf, 2009.

[Olano and Baker 10] Marc Olano and Dan Baker. "LEAN Mapping." http://www.csee.umbc.edu/~olano/papers/lean/, 2010.

[Reshetov 09] Alexander Reshetov. "Morphological Antialiasing." Preprint, http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf, 2009.

[Valve 08] Valve Developer Community. "Half Lambert." https://developer.valvesoftware.com/wiki/Half_Lambert, 2008.