**Anti-aliased Alpha Test: The Esoteric Alpha To Coverage**

[[](https://bgolus.medium.com/?source=post_page-----8b177335ae4f--------------------------------)](https://bgolus.medium.com/?source=post_page-----8b177335ae4f--------------------------------)

[Ben Golus](https://bgolus.medium.com/?source=post_page-----8b177335ae4f--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fd4d33ac3d661&operation=register&redirect=https%3A%2F%2Fbgolus.medium.com%2Fanti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f&user=Ben+Golus&userId=d4d33ac3d661&source=post_page-d4d33ac3d661----8b177335ae4f---------------------post_header-----------)

15 min read

·

Aug 13, 2017

454

14

Aliasing is the bane of VR. We have several tools we use to try to remove, or at least reduce aliasing in real time graphics today. Tools like [super sampling](https://en.wikipedia.org/wiki/Supersampling), [temporal anti-aliasing (TAA)](https://en.wikipedia.org/wiki/Temporal_anti-aliasing), and [multi sample anti-aliasing (MSAA)](https://en.wikipedia.org/wiki/Multisample_anti-aliasing). They all have their pros and cons, but there’s one tool I don’t feel like gets the use and respect it deserves. That tool is a feature of MSAA, Alpha to Coverage.

This article will be assuming you’re using forward rendering with at least 4x MSAA. If you’ve chosen to use deferred rendering this article will be less useful for you. Some techniques are still applicable though. If you’ve chosen to use TAA and forward rendering this may solve one of the reasons you chose that option!

**The Problem**

**Self Inflicted Aliasing**

A very common case of aliasing is the use of alpha test, used for things like trees or bushes. Lets look at why it’s used so much even though it causes so much aliasing.



This is a basic plant mesh in Unity with a very basic lit alpha test shader. In Unity these are often called Cutout shaders. I’m using the bush03 mesh and texture from [Nature Starter Kit 2](https://www.assetstore.unity3d.com/en/#!/content/52977) on the asset store if you want to try this yourself. It doesn’t look too bad, but there’s a lot of aliasing there from the alpha testing. Note that the above image *is*using 4x MSAA. That doesn’t help here since this is shader aliasing which generally MSAA can’t help with. This is something people usually fix with some extra super sampling or they give up on MSAA and use TAA instead. Or they just live with it.

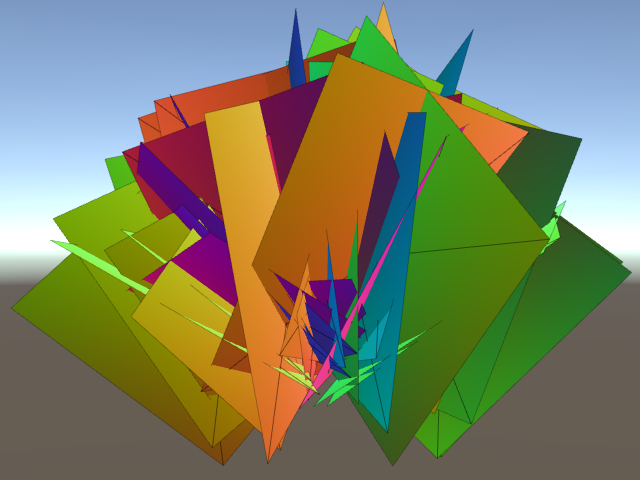
What about traditional alpha blending? Seems like that should be an obvious answer, so why don’t people use that?



That looks way better, at least in this static shot, maybe a little blurry, but the aliasing is gone! Success, right? Of course it isn’t that easy. In motion using alpha blending for this case looks *really*wrong.



It almost looks like the bush is inside out! It’s even more obviously wrong in VR, so this isn’t a usable option. So what’s happening here? Alpha blended shaders don’t get to use per pixel depth sorting like opaque and alpha test shaders do. The bush is made from a bunch of intersecting polygons. You could sort the polygons, but they’re still intersecting so that only helps so much. So, that’s why we’re stuck with using alpha test.



But there *are*ways to have the benefit of both alpha test and alpha blend; a way to get per pixel depth sorting and the smooth edge. There are some two pass techniques[¹](https://bgolus.medium.com/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f#620e) that can work well. There are also some order independent transparency approximations, though they are potentially expensive. There’s an even better option if you’re using MSAA. Enter Alpha to Coverage.

**The Solution**

**This One Simple Trick …**

Alpha to Coverage (sometimes referred to as A2C or ATOC) is a less commonly mentioned graphics technique. While used by several big games, it is seemingly unknown to the vast majority of smaller devs. It’s a powerful tool that is often overlooked due to the requirement of MSAA. Games often treat MSAA as an expensive option for high end PCs. Or they’ve used rendering techniques that don’t allow for easy use of MSAA to begin with, like deferred rendering. The result is games have overwhelmingly preferred post process anti-aliasing like [FXAA](https://en.wikipedia.org/wiki/Fast_approximate_anti-aliasing). This made Alpha to Coverage even more uncommon for a while. But the rise of VR has lead to the resurgence of forward rendering and of MSAA. This means A2C has become a useful tool again, especially for VR.

Anti-aliased alpha test is one of the most basic use cases for Alpha to Coverage. The basic idea for this has been around for as long as A2C has existed. It may even have been the original intended use for it. The first time I saw this idea was on Emil Persson’s (better known to many as Humus) site [here](http://www.humus.name/index.php?page=3D&ID=61). I believe the method in this article is superior to the one used in his original demo.

**Multi Sample Anti-Aliasing**

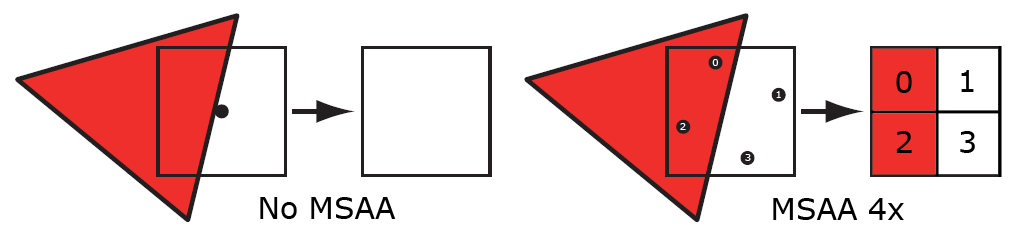
Before we get into that though, I’m going to quickly run down what Alpha to Coverage *is* and why MSAA is a requirement.

Alpha to Coverage maps the alpha output from a pixel shader to the coverage mask of MSAA. Obvious, right?!

Okay, that’s going to take a little more explanation. Be warned, I’m going to get into some of the gritty details here.

The big trick with MSAA and how it differs from super sampling is what Multi Sample Anti-Aliasing is doing multiple samples *of*. Super sampling at its most basic is rendering *everything*at a higher resolution. The final on screen image is a scaled down average of this higher resolution render. MSAA renders the color at the same resolution as the target resolution, but multiple *coverage*samples per pixel. In simpler terms it’s rendering the depth of the scene at a higher resolution than the target resolution. Each rendered triangle’s depth samples are its coverage. If you’re using 4x MSAA it’s rendering 4 depth / coverage samples within the bounds of each pixel. Each coverage sample can store a unique color. But it’s only getting a color by running the pixel shader once *per triangle*, per pixel. Most of time all 4 depth samples will be the same triangle. In this case all 4 coverage samples use the same single pixel shader output color. This behaves no different than when having no MSAA enabled at all.

Where MSAA’s strength comes into play is when a triangle doesn’t cover all the depth samples. For example, if a triangle only covers two depth samples the two matching coverage samples are set to the pixel shader ouput. The other coverage samples can store the color of other triangles’s pixel shaders that can be seen.



from <https://therealmjp.github.io/posts/msaa-overview/>

More in depth write ups can be found elsewhere, like on Matt Pettineo’s (MJP) site.

<https://therealmjp.github.io/posts/msaa-overview/>

I’m glossing over a bunch of details, but the main point is MSAA can store a list of multiple colors per pixel, one color per coverage sample. These get averaged together to make the final on screen pixel color. But it’s not always having to render a pixel shader more than once per pixel resulting in significant savings over super sampling of similar quality.

**Alpha to Coverage**

This is where Alpha to Coverage comes in. Alpha to Coverage let’s the pixel shader’s output alpha change the fraction of coverage samples rendered to. Hence the name. In the simple case of a triangle covering all 4 depth samples, all 4 coverage samples store the color output. With Alpha to Coverage enabled and an output color alpha of 0.5, only half of the possible coverage samples will store the color. The result will be as if it’s half[²](https://bgolus.medium.com/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f#5995) as opaque as it’s only contributing color to half the samples!

Okay, so with 4x MSAA and Alpha to Coverage you get 5 levels of opacity you can play with, 4/4 through 0/4 samples. What good is that? With traditional alpha blending you get at least 256 levels, so why would you ever want to use this? Because with Alpha to Coverage those coverage samples are opaque. That means they can be depth sorted! You can have a mesh of polygon soup and have intersecting triangles accurately sort at better than per pixel accuracy. This is no different than opaque objects or even alpha test with MSAA, but you also get soft edges.

Lets just turn Alpha to Coverage on and see how that looks. In Unity’s ShaderLab it’s enabled by just adding AlphaToMask On to an otherwise opaque shader that outputs alpha. So here we go …

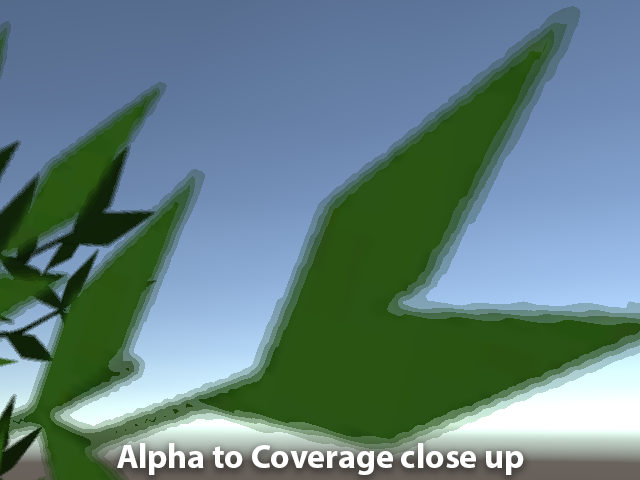


And we’re done, doesn’t that look so much better than before?! Lets compare it to the original alpha test version!





Wait, no, Alpha to Coverage looks terrible! It’s somehow simultaneously fuzzy and aliased, not to mention sort of noisy, and seems like the bush got more transparent through the middle? The story only gets worse when we get close up to those leaves.



Oy vey, that’s no good at all. The limited number of opacity steps makes for some ugly banding. Clearly there’s a reason why this isn’t used more often, so why would I even suggest doing this if it looks so bad?!

**Fixing the Solution**

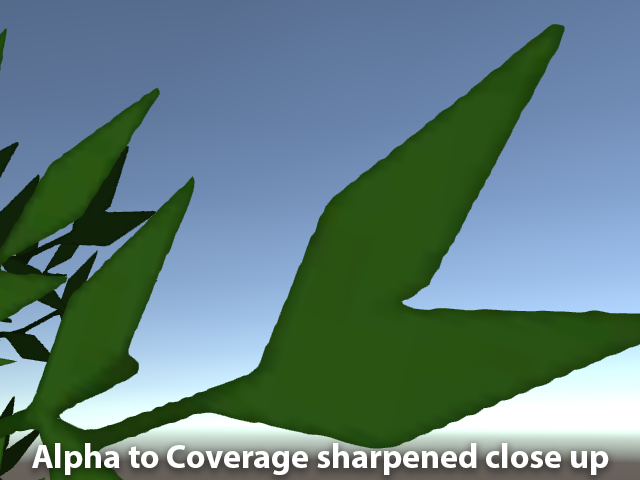
Well, because that’s what happens if you use Alpha to Coverage with the alpha of the texture straight. This is what replacing basic alpha blending using alpha to coverage looks like. Many implementations of Alpha to Coverage I’ve seen in games really do leave it here.

But what’s happening in that first alpha to coverage image to make it so bad? It’s not just the banding, it’s also the interaction between the coverage samples. In short it’s a limitation of using Alpha to Coverage. Essentially when using Alpha to Coverage only “one layer” of a specific coverage can be seen at one time. Two or more overlapping triangles with 50% Alpha to Coverage results in no change in opacity. Only the closest coverage points get rendered. This is unlike alpha blending where they will blend over each other increasing the perceived opacity. If you remember earlier I said the coverage samples are individually opaque. If the output alpha is 50% with 4x MSAA only two coverage samples are written to. But it’s going to be the *same* two coverage samples for everything that renders to that pixel with 50% A2C! The answer might seem like “well, render to different samples!”. But that’s harder than it sounds to do and be effective. It is an option though. Or you might be thinking “nope, still don’t get it at all”, and that’s okay. Don’t worry about it. It’s unnecessary to understand for this particular case as it’s relatively easy to solve.

So what is the solution here? Alpha to Coverage is great for anti-aliasing alpha *test*, not so much for replacing an alpha blend! We need to sharpen the alpha so there are fewer overlapping areas of partial opacity. This will simultaneously solve the banding, and interior transparency. Anyone who’s done signed distance field text, or shader based line rendering probably knows whats coming next. For everyone else I’m going to introduce you to one of my favorite functions in shaders:

fwidth()

With out further explanation we get this (I’ll talk about it later):





Now that’s better. Looks very similar to the original alpha tested version, but is nicely anti-aliased. And it’s not strangely transparent, noisy, or blurry anymore! Here’s the original alpha tested version one more time for reference.

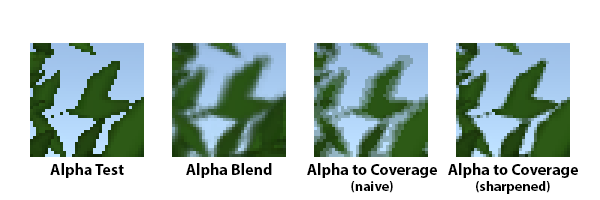




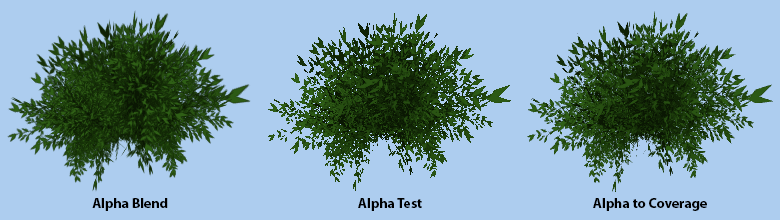
So, what did I do? What was that I mentioned earlier? All the magic is in this one line of shader code.fwidth()

col.a = (col.a - \_Cutoff) / max(fwidth(col.a), 0.0001) + 0.5;

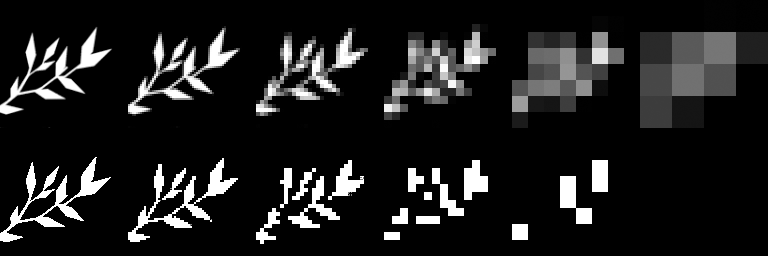
What that line does is sharpens the alpha of the texture to the width of a single pixel. The function returns the sum of the pixel’s partial derivatives for a value, usually described as the equivalent of . Those two functions, and , are partial derivative functions. They measure the change of a value between the two pixels either to the side or above[³](https://bgolus.medium.com/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f#68b9) the pixel that’s currently being processed. Derivatives are wonderful, magical things in shaders. For the moment it’s enough to know gives a good approximation of how much a value changes per pixel. If you divide a value by how much it’s changing it’ll rescale it to give you a nice crisp edge. The is there because if the texture’s alpha is a flat color at some point, like black or white, will be zero. Dividing by zero is bad! The 0.0001 is just an arbitrarily small number, small enough that you have to really get close to a texture before the banding will show up again. The is the same value you would be using for an alpha test shader and helps center the alpha around the edge you want.fwidth()abs(ddx(value)) + abs(ddy(value))ddx()ddy()fwidth()max()fwidth()\_Cutoff



We could end here, but another problem that will show up that most people will probably miss initially. It is an issue that plagues alpha test already, so people might just be living with it. As you get far away from an object using alpha test it’ll appear to shrink and eventually vanish. Alpha blending does not suffer from this.



The cause is the texture’s mip mapping making the alpha fade out from averaging. Since the alpha is fading out that hard “cutoff” edge is vanishing too.



Texture mip mapping, Alpha Blend (top) vs Alpha Test (bottom)

There are several workarounds to this problem. Lowering the cutoff value (a common solution for alpha testing in general). Super sampling in the shader (I do this for font rendering). Or using some kind of noise like stochastic sampling or [hashed alpha testing](https://casual-effects.com/research/Wyman2017Hashed/index.html). Probably the best solution is the one proposed by Ignacio Castaño on The Witness blog back in 2010. It requires recalculating each mip maps’ alpha values in a way that’s aware of it’s intended use for alpha testing or Alpha to Coverage. The Witness uses A2C for foliage as well as several other effects too.

**[Computing Alpha Mipmaps](http://the-witness.net/news/2010/09/computing-alpha-mipmaps/?source=post_page-----8b177335ae4f--------------------------------" \t "_blank)**

[A little problem that we had when we started to create trees and vegetation is that the standard mipmap generation…](http://the-witness.net/news/2010/09/computing-alpha-mipmaps/?source=post_page-----8b177335ae4f--------------------------------" \t "_blank)

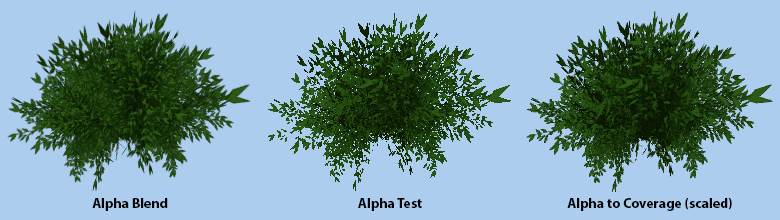
[the-witness.net](http://the-witness.net/news/2010/09/computing-alpha-mipmaps/?source=post_page-----8b177335ae4f--------------------------------" \t "_blank)

It has the great advantage of being “free” since all of the work is done before hand and there’s nothing the shader has to do. For Unity it could be implemented with an asset post processor or an editor script.

*Edit: Sébastien Lagarde brought to my attention the fact that Ignacio’s technique is now available as an official texture import option in Unity 2017.1! There’s now a Mip Maps Preserve Coverage check box.*

But there is another way that works almost as well, scaling the alpha by mip level. Effectively that’s what the above solution is doing, so we can just do an approximation the shader. It’s not “free”, but it’s not that expensive either.

col.a \*= 1 + CalcMipLevel(i.uv \* \_MainTex\_TexelSize.zw) \* \_MipScale;



We can approximate the loss of density from mip mapping almost perfectly with a \_MipScale of 0.25. The CalcMipLevel() function is the magic here. You can see the implementation of the function in the shader code below. Note if you use any kind of LOD bias on your textures you’ll have to account for this manually. If you want more information about the function you can read this Stack Overflow post:

<https://stackoverflow.com/questions/24388346/how-to-access-automatic-mipmap-level-in-glsl-fragment-shader-texture>

The same scaling trick works for alpha test as well. In fact if MSAA is disabled Alpha to Coverage falls back to alpha test. This means it isn’t only useful for VR or other games where you’re guaranteed to have MSAA enabled. Basically there’s rarely a reason to use straight alpha testing ever!

**Conclusion**

You don’t need to live with the aliasing of alpha test, the sorting issues of alpha blend, or the cost of a two pass shader. There is a happy middle ground using Alpha to Coverage. If you’re using MSAA you should never need to use alpha test again!

**Appendix**

**The Shader**

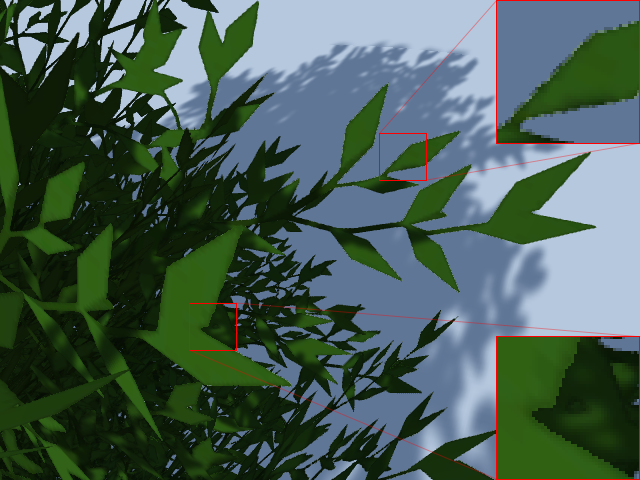
The Unity ShaderLab file for the final shader used in this article can be found below.

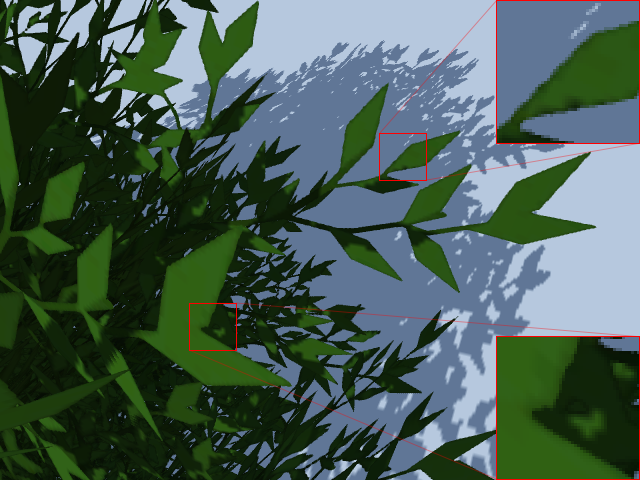
Shader "Custom/Alpha To Coverage"  
{  
 Properties  
 {  
 \_MainTex ("Texture", 2D) = "white" {}  
 \_Cutoff ("Alpha cutoff", Range(0.15,0.85)) = 0.4  
 \_MipScale ("Mip Level Alpha Scale", Range(0,1)) = 0.25  
 }  
 SubShader  
 {  
 Tags { "RenderQueue"="AlphaTest" "RenderType"="TransparentCutout" }  
 Cull Off  
   
 Pass  
 {  
 Tags { "LightMode"="ForwardBase" }  
 AlphaToMask On  
   
 CGPROGRAM  
 #pragma vertex vert  
 #pragma fragment frag  
   
 #include "UnityCG.cginc"  
 #include "Lighting.cginc"  
   
 struct appdata  
 {  
 float4 vertex : POSITION;  
 float2 uv : TEXCOORD0;  
 half3 normal : NORMAL;  
 };  
   
 struct v2f  
 {  
 float4 pos : SV\_POSITION;  
 float2 uv : TEXCOORD0;  
 half3 worldNormal : NORMAL;  
 };  
   
 sampler2D \_MainTex;  
 float4 \_MainTex\_ST;  
 float4 \_MainTex\_TexelSize;  
   
 fixed \_Cutoff;  
 half \_MipScale;  
   
 float CalcMipLevel(float2 texture\_coord)  
 {  
 float2 dx = ddx(texture\_coord);  
 float2 dy = ddy(texture\_coord);  
 float delta\_max\_sqr = max(dot(dx, dx), dot(dy, dy));  
   
 return max(0.0, 0.5 \* log2(delta\_max\_sqr));  
 }  
   
 v2f vert (appdata v)  
 {  
 v2f o;  
 o.pos = UnityObjectToClipPos(v.vertex);  
 o.uv = TRANSFORM\_TEX(v.uv, \_MainTex);  
 o.worldNormal = UnityObjectToWorldNormal(v.normal);  
 return o;  
 }  
   
 fixed4 frag (v2f i, fixed facing : VFACE) : SV\_Target  
 {  
 fixed4 col = tex2D(\_MainTex, i.uv);  
 // rescale alpha by mip level (if not using preserved coverage mip maps)  
 col.a \*= 1 + max(0, CalcMipLevel(i.uv \* \_MainTex\_TexelSize.zw)) \* \_MipScale;  
 // rescale alpha by partial derivative  
 col.a = (col.a - \_Cutoff) / max(fwidth(col.a), 0.0001) + 0.5;  
   
 half3 worldNormal = normalize(i.worldNormal \* facing);  
   
 fixed ndotl = saturate(dot(worldNormal, normalize(\_WorldSpaceLightPos0.xyz)));  
 fixed3 lighting = ndotl \* \_LightColor0;  
 lighting += ShadeSH9(half4(worldNormal, 1.0));  
   
 col.rgb \*= lighting;  
   
 return col;  
 }  
 ENDCG  
 }  
 }  
}

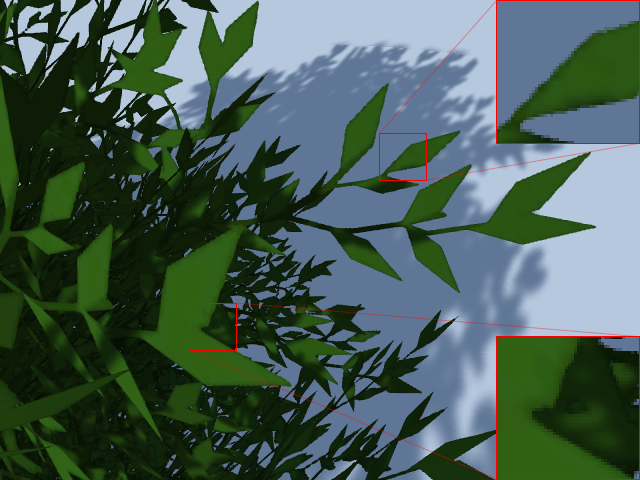
**Unity’s Surface Shaders & Directional Shadows**

Many peoples’ first inclination is probably going to be to try this technique in Unity with a surface shader by adding AlphaToMask On. This does work, but it breaks shadow casting and receiving if you use the addshadow surface option due to Unity’s default shadow caster passes outputing 0% alpha and AlphaToMask On still being enabled for the generated pass. You can use Fallback “Legacy Shaders/Transparent/Cutout/VertexLit” instead as fallback passes don’t inherent the parent shader file’s render settings, but it won’t do the mip level alpha rescaling so it might look odd in the distance unless you use a custom shadow caster pass. Even then Unity’s directional shadows will sometimes cause bright and dark noise around the edges or seemingly disable anti-aliasing since their directional shadows are rendered with out MSAA. This is one of the reasons why MSAA never seems to work as well as you might expect it to in Unity. You can disable shadow cascades in the Graphics settings, but then the shadows don’t look as nice. It does mean if you’re not using real time directional shadows on the PC or consoles, or you’re working on a mobile project, this technique works great. Upcoming changes to Unity’s forward renderer will also likely solve this.

There is a work around for getting shadow cascades and MSAA working together better, which I used for [Wayward Sky](http://www.uberent.com/waywardsky/) and [Dino Frontier](http://www.uberent.com/dinofrontier/) (both out now for the PSVR). But that is for another article. I posted the basic idea on the Unity forums [here](https://forum.unity3d.com/threads/fixing-screen-space-directional-shadows-and-anti-aliasing.379902/) for those curious.







artifacts and apparent loss of anti-aliasing from Unity’s cascaded shadow maps (left); cascades disabled in Graphics settings (center); cascades and MSAA with fix (right)

¹[^](https://bgolus.medium.com/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f#5de2)Rendering an object once using an alpha test pass, and again with an alpha blend pass is a good solution, especially if you don’t have MSAA. There are plenty of places that have discussed this technique, like Wolfire Games’ blog [here](http://blog.wolfire.com/2009/02/rendering-plants-with-smooth-edges/) (which also mentions an Alpha to Coverage like technique) and even Unity’s documentation on [Legacy Alpha Testing](https://docs.unity3d.com/Manual/SL-AlphaTest.html). Interestingly, Alpha to Coverage can be used leverage the two pass technique for softer blending than is normally possible with Alpha to Coverage alone, so effectively getting the benefits of Alpha to Coverage and alpha blending. It’s mentioned briefly on Luis Antonio’s site in a post about the clouds in The Witness. It’s actually a section written by Ignacio Castaño again.

<http://www.artofluis.com/3d-work/the-art-of-the-witness/clouds/>

²[^](https://bgolus.medium.com/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f#dec4)Two coverage samples result in something similar to 50% opaque, but only if using gamma (aka sRGB) color space rendering. Otherwise if you’re using linear color space rendering and not using a custom resolver it’s more like ~74% alpha, or ~188/255, which if you’re familiar with linear to gamma conversions might look familiar.

³[^](https://bgolus.medium.com/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f#41e8)Pixel derivatives work in blocks of 2x2 pixels, so fwidth() might be comparing the pixel above or below, and the pixel to the left or right, depending on which pixel in the 2x2 block the pixel/fragment shader is currently running. The ddx() and ddy() functions individually give the horizontal and vertical partial derivatives. These are always compared in the same order, so the ddx() and ddy() values for each pair of pixels is the same. Depending on the implementation it may even be the same values for all four pixels in the 2x2 grid. Basically 4 instances of a pixel/fragment shader is running in parallel always, even if a triangle is only covering one of those pixels.

[Programming](https://medium.com/tag/programming?source=post_page-----8b177335ae4f---------------programming-----------------)

[Shaders](https://medium.com/tag/shaders?source=post_page-----8b177335ae4f---------------shaders-----------------)

[Unity3d](https://medium.com/tag/unity3d?source=post_page-----8b177335ae4f---------------unity3d-----------------)

[Unity Game Development](https://medium.com/tag/unity-game-development?source=post_page-----8b177335ae4f---------------unity_game_development-----------------)

454

14