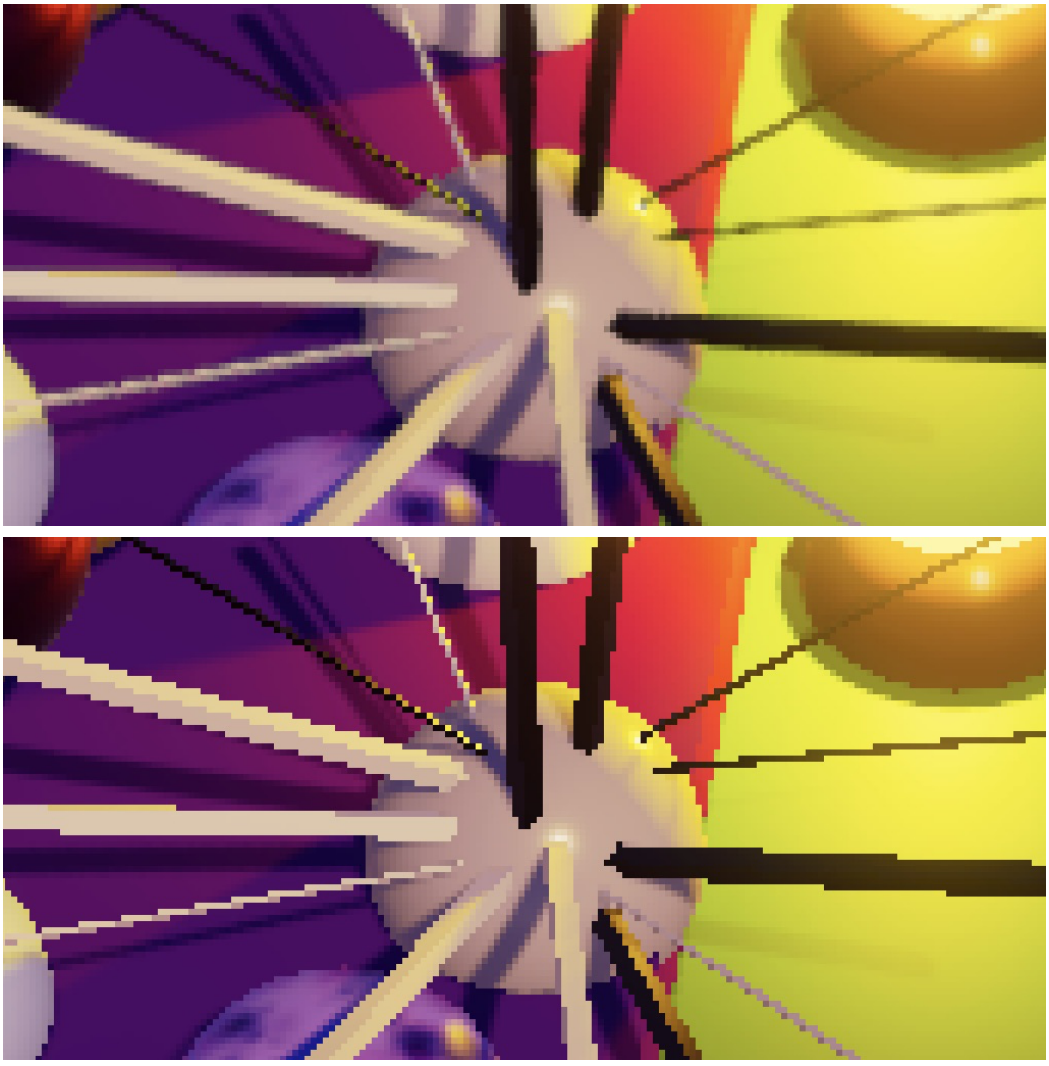
published 2021-02-28

FXAA Fast approXimate Anti-Aliasing

Calculate and store pixel luma, or fall back to green.
Find and blend high-contrast pixels.
Detect and smooth long edges.
Combine FXAA and render scale.

This is the 17th part of a tutorial series about creating a custom scriptable render pipeline. It covers the implementation of the FXAA antialiasing algorithm.

This tutorial is made with Unity 2019.4.20f1.



An image with and without FXAA, zoomed in.

Changes

I made the integer division and modulo operations in `GetLighting` faster by using their unsigned versions. The D3D compiler complains about it otherwise.

```
			int lightIndex = unity_LightIndices[(uint)j / 4][(uint)j % 4];
```

I also fixed an incorrectly interpolated normal in `LitPassFragment`.

```
	#if defined(_NORMAL_MAP)
		surface.normal = NormalTangentToWorld(
			GetNormalTS(config), input.normalWS, input.tangentWS
		);
		surface.interpolatedNormal = input.normalWS;
	#else
		surface.normal = normalize(input.normalWS);
		surface.interpolatedNormal = surface.normal;
	#endif
	//surface.interpolatedNormal = surface.normal;
```

Finally, I changed *PostFXStackPasses* to sidestep mip map level selection, as that is never needed for post effects.

```
float4 GetSource(float2 screenUV) {
	return SAMPLE_TEXTURE2D_LOD(_PostFXSource, sampler_linear_clamp, screenUV, 0);
}

…

float4 GetSource2(float2 screenUV) {
	return SAMPLE_TEXTURE2D_LOD(_PostFXSource2, sampler_linear_clamp, screenUV, 0);
}
```

And the same in *CameraRendererPasses*.

```
float4 CopyPassFragment (Varyings input) : SV_TARGET {
	return SAMPLE_TEXTURE2D_LOD(_SourceTexture, sampler_linear_clamp, input.screenUV, 0);
}

float CopyDepthPassFragment (Varyings input) : SV_DEPTH {
	return
		SAMPLE_DEPTH_TEXTURE_LOD(_SourceTexture, sampler_point_clamp, input.screenUV, 0);
}
```

And also in *Fragment*.

```
Fragment GetFragment (float4 positionSS) {
	…
	f.bufferDepth =
		SAMPLE_DEPTH_TEXTURE_LOD(_CameraDepthTexture, sampler_point_clamp, f.screenUV, 0);
	…
}

float4 GetBufferColor (Fragment fragment, float2 uvOffset = float2(0.0, 0.0)) {
	float2 uv = fragment.screenUV + uvOffset;
	return SAMPLE_TEXTURE2D_LOD(_CameraColorTexture, sampler_linear_clamp, uv, 0);
}
```

1 FXAA Post-FX

The limited resolution of the frame buffer introduces visual aliasing artifacts to the final image. These are often referred to as jaggies or stairsteps, which are visible along lines that are not aligned with the pixel grid. Besides that features smaller than a pixel either appear or not, which can produce temporal flickering artifacts when they are in motion.

In the previous tutorial we added the ability to apply SSAA by at most doubling the render scaled and downsampling afterwards. This smoothes our jaggies somewhat and doubles the resolution for the purpose of detecting and then smoothing tiny features. While doubling the render scale improves visual quality it also requires the GPU to process four times as many fragments, so it is very expensive and usually not feasible for realtime use. A render scale less than 2 could be used, but on its own that doesn't improve quality much.

An alternative to increasing resolution is to apply a post FX to the original image that smoothes out all aliasing artifacts. Various such algorithms have been developed, which are known as acronyms that always end with AA, such as FXAA, MLAA, SMAA, and TAA. In this tutorial we'll implement FXAA, which is the simplest and fastest approach.

The first post-FX antialiasing solution was morphological anti-aliasing, abbreviated to MLAA. It analyses the image to detect the edges of visual features and then selectively blurs those. FXAA is a simpler approach inspired by MLAA. It stands for fast approximate anti-aliasing. It was developed by Timothy Lottes at NVIDIA. Compared to MLAA, it trades quality for speed. While a common complaint of FXAA is that it blurs too much, that varies depending on which variant is used and how it is tuned. We'll create the latest version—FXAA 3.11—specifically the high-quality variant that also investigates long edges.

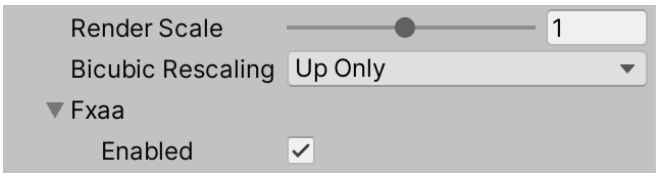
What about MSAA?

Like SSAA, MSAA renders to a higher resolution and later downsamples, but changes how fragments are rendered. Instead of rendering all fragments of a higher-resolution square it renders only a single fragment per triangle that covers that block, effectively copying the result to the higher-resolution pixels. This keeps the fill rate manageable. It also means that only the edges of triangles are affected, everything else remains unchanged. That's why MSAA doesn't smooth the edges of clipped surfaces and doesn't mitigate shader aliasing.

1.1 Enabling FXAA

Although FXAA is a post FX it globally affects the image quality like the render scale does, so we'll adds its configuration to `CameraBufferSettings`. Initially we only need a toggle to enable it, but we'll add some more configuration options for it later. So we'll group all FXAA settings in a new `CameraBufferSettings.FXAA` struct.

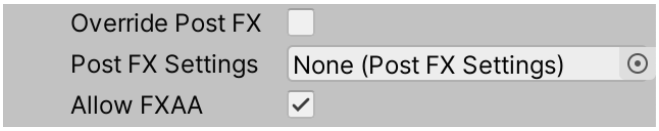
```
using System;
using UnityEngine;

[Serializable]
public struct CameraBufferSettings {

	…

	[Serializable]
	public struct FXAA {

		public bool enabled;
	}

	public FXAA fxaa;
}
```



FXAA enabled for the RP.

Again as we did with the render scale, we'll make it possible to control whether FXAA is used per camera, by adding a toggle to `CameraSettings` to control whether FXAA is allowed. It should be disallowed by default. This ensures that FXAA isn't applied to the scene window, material previews, or reflection probes.

```
	public bool allowFXAA = false;
```



FXAA enabled for the camera.

As FXAA is a post FX it is the responsibility of `PostFXStack` to apply it. This means that FXAA will only work if post FX are in use. Also, the FXAA configuration has to be passed to the stack, so add a parameter for it to `PostFXStack.Setup` and copy it to a field.

```
	CameraBufferSettings.FXAA fxaa;

	…

	public void Setup (
		…
		CameraBufferSettings.BicubicRescalingMode bicubicRescaling,
		CameraBufferSettings.FXAA fxaa
	) {
		this.fxaa = fxaa;
		…
	}
```

Pass the FXAA configuration to it in `CameraRenderer.Render`, after applying the camera's toggle to the global toggle.

```
		bufferSettings.fxaa.enabled &= cameraSettings.allowFXAA;
		postFXStack.Setup(
			context, camera, bufferSize, postFXSettings, useHDR, colorLUTResolution,
			cameraSettings.finalBlendMode, bufferSettings.bicubicRescaling,
			bufferSettings.fxaa
		);
```

Note that we can directly modify the buffer settings struct field because it contains a copy of the RP settings struct, not a reference to the original.

1.2 FXAA Pass

We need a pass to apply FXAA, so add it to the *PostFXStack* shader, along with a corresponding entry in the `PostFXStack.Pass` enum. The pass is a copy of *Final Rescale* renamed to *FXAA* and with its fragment function set to `FXAAPassFragment`. Besides that, we'll put the FXAA shader code in a separate *FXAAPass* HLSL file and include it in the pass itself only.

```
		Pass {
			Name "FXAA"

			Blend [_FinalSrcBlend] [_FinalDstBlend]

			HLSLPROGRAM
				#pragma target 3.5
				#pragma vertex DefaultPassVertex
				#pragma fragment FXAAPassFragment
				#include "FXAAPass.hlsl"
			ENDHLSL
		}
```

Create the new *FXAAPass.hlsl* file, initially containing only the `FXAAPAssFragment` function that returns the source pixel without modification.

```
#ifndef CUSTOM_FXAA_PASS_INCLUDED
#define CUSTOM_FXAA_PASS_INCLUDED

float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	return GetSource(input.screenUV);
}

#endif
```

We have to apply FXAA after color grading, for the same reason that the final rescale must happen after color grading. As we now have multiple scenarios in which the *Final* pass is no longer the true final let's rename it to *Apply Color Grading*, as that's what it actually does. Do this both in the shader and the `Pass` enum. Rename its fragment function to `ApplyColorGradingPassFragment`.

Let's also rename the `PostFXStack.DoColorGradingAndToneMapping` method to `DoFinal`, because it's now doing a lot more than just color grading and tone mapping.

When FXAA is enabled we first have to perform color grading and then apply FXAA on top of that. So we have to store the color-grading result in a temporary render texture. Add a shader property identifier for it to `PostFXStack`.

```
		colorGradingResultId = Shader.PropertyToID("_ColorGradingResult"),
		finalResultId = Shader.PropertyToID("_FinalResult"),
```

In `DoFinal`, check whether FXAA is enabled right before we move on to the final drawing phase. If FXAA is enabled, immediately perform color grading and store the result in a new temporary LDR texture.

```
		buffer.SetGlobalVector(colorGradingLUTParametersId,
			new Vector4(1f / lutWidth, 1f / lutHeight, lutHeight - 1f)
		);

		if (fxaa.enabled) {
			buffer.GetTemporaryRT(
				colorGradingResultId, bufferSize.x, bufferSize.y, 0,
				FilterMode.Bilinear, RenderTextureFormat.Default
			);
			Draw(sourceId, colorGradingResultId, Pass.ApplyColorGrading);
		}

		if (bufferSize.x == camera.pixelWidth) {
			DrawFinal(sourceId, Pass.ApplyColorGrading);
		}
```

Like we did for an adjusted render scale, we have to make sure that the final blend mode for color grading is set to `One Zero`. As this can now happen in two places let's simply always reset it before we begin drawing.

```
		buffer.SetGlobalFloat(finalSrcBlendId, 1f);
		buffer.SetGlobalFloat(finalDstBlendId, 0f);
		if (fxaa.enabled) {
			…
		}

		if (bufferSize.x == camera.pixelWidth) {
			DrawFinal(sourceId, Pass.ApplyColorGrading);
		}
		else {
			buffer.GetTemporaryRT(
				finalResultId, bufferSize.x, bufferSize.y, 0,
				FilterMode.Bilinear, RenderTextureFormat.Default
			);
			//buffer.SetGlobalFloat(finalSrcBlendId, 1f);
			//buffer.SetGlobalFloat(finalDstBlendId, 0f);
			…
		}
```

Next, if the buffer isn't scaled we again have to check whether FXAA is enabled. If so the final draw is with the FXAA pass and the color grading result, after which the color grading result has to be released. Otherwise color grading is the final pass, as before.

```
		if (bufferSize.x == camera.pixelWidth) {
			if (fxaa.enabled) {
				DrawFinal(colorGradingResultId, Pass.FXAA);
				buffer.ReleaseTemporaryRT(colorGradingResultId);
			}
			else {
				DrawFinal(sourceId, Pass.ApplyColorGrading);
			}
		}
```

In case of an adjusted render scale we still have to first render to an intermediate final result texture. If FXAA is enabled we do this with a regular draw of the FXAA pass with the color grading result, after which we release the color grading result. Otherwise it's the regular draw that applies color grading to the original source.

```
		else {
			buffer.GetTemporaryRT(
				finalResultId, bufferSize.x, bufferSize.y, 0,
				FilterMode.Bilinear, RenderTextureFormat.Default
			);

			if (fxaa.enabled) {
				Draw(colorGradingResultId, finalResultId, Pass.FXAA);
				buffer.ReleaseTemporaryRT(colorGradingResultId);
			}
			else {
				Draw(sourceId, finalResultId, Pass.ApplyColorGrading);
			}

			…
		}
```

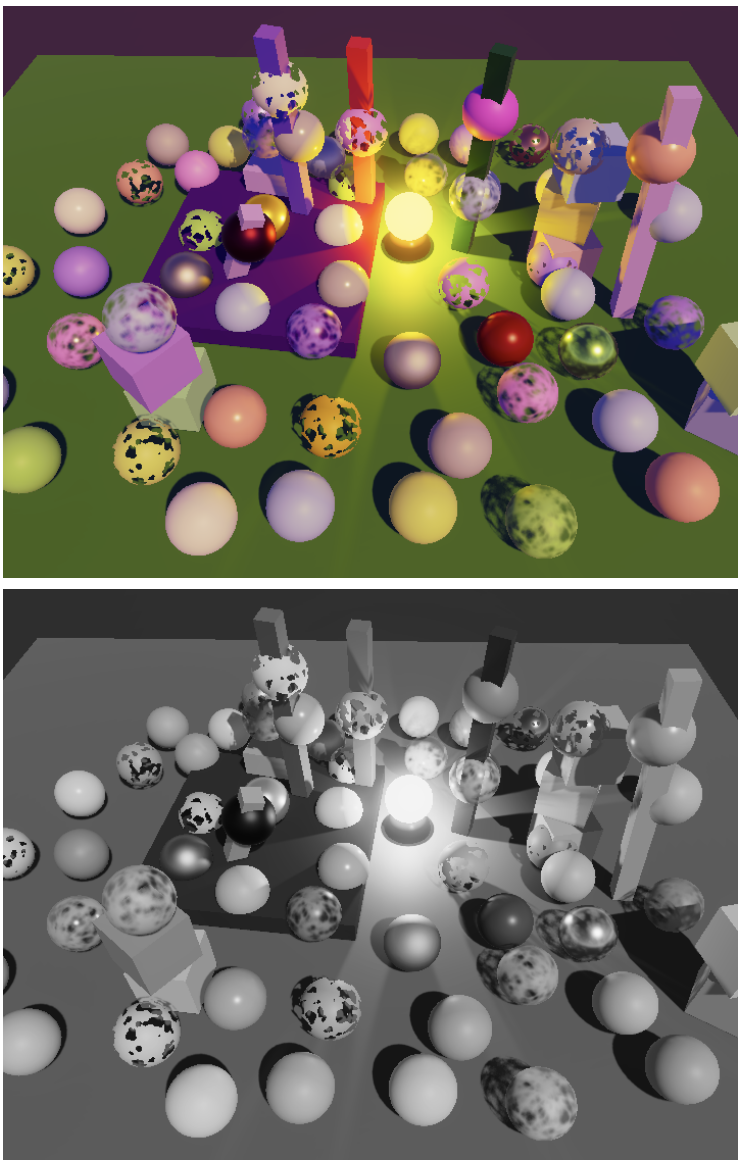
At this point our RP still produces the same results as before, but when FXAA is enabled the frame debugger will reveal an extra draw step with the FXAA pass.

1.3 Luma

FXAA works by selectively reducing the contrast of the image, smoothing out visually obvious jaggies and isolated pixels. Contrast is determined by comparing the perceived intensity of pixels. As the goal is to reduce artifacts that we perceive FXAA is only concerned with perceived brightness, which is gamma-adjusted luminance, known as luma. The exact color of the pixels doesn't matter, it's their luma that counts. Thus FXAA analyses a grayscale image. This means that hard transitions between different colors won't be smoothed out much when their luma is similar. Only visually obvious transitions are strongly affected.

Add a `GetLuma` function to *FXAAPass* that returns a luma value for some UV coordinates. Initially have it return the linear luminance of the source. Then make the FXAA pass return that. Note that FXAA works on LDR data after color grading and tone mapping, so this represents the luma of the final image.

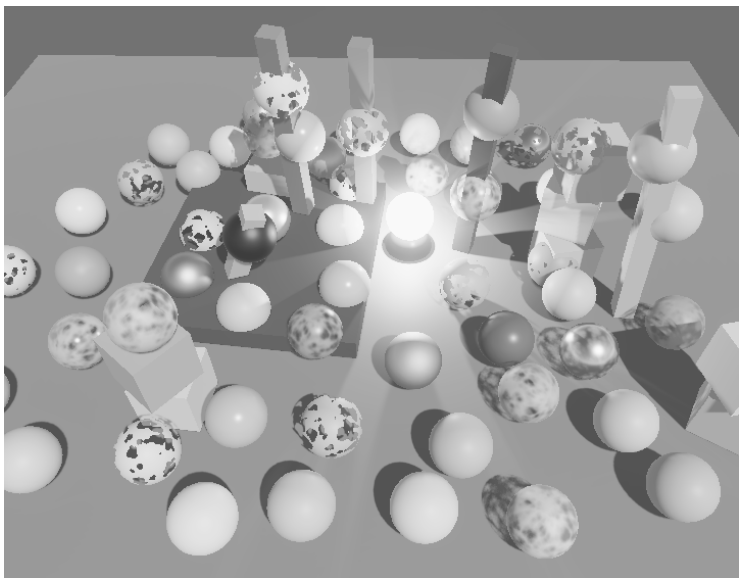
```
float GetLuma (float2 uv) {
	return Luminance(GetSource(uv);
}

float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	return GetLuma(input.screenUV);
}
```



Original color and linear luminance.

Because we're more perceptive to changes of dark colors that light ones we have to apply a gamma adjustment to the luminance to get an appropriate luma value. A gamma of 2 is sufficiently accurate, which we get by taking the square root of the linear luminance.

```
float GetLuma (float2 uv) {
	return sqrt(Luminance(GetSource(uv)));
}
```

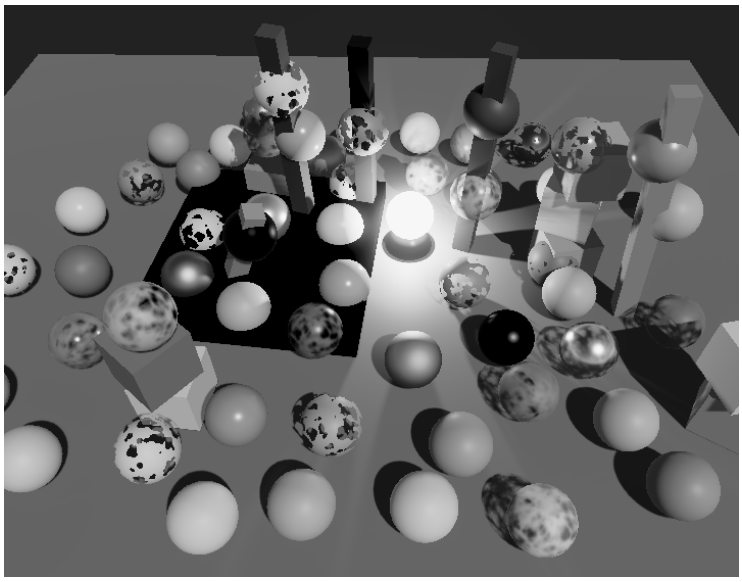


Gamma 2.0 luma.

1.4 Green for Luma

FXAA works by detecting contrast and edges, which requires multiple samples per fragment. Calculating luma for each sample would make it too expensive. Because we are visually most sensitive to green, a common alternative to calculating luma is to directly use the green color channel instead. This lowers the quality but avoids a dot product and square root operation.

```
float GetLuma (float2 uv) {
	return GetSource(uv).g;
}
```



Green as luma.

1.5 Storing Luma in the Alpha Channel

Calculating luma produces much better results that relying on just the green color channel, but we don't want to calculate it each time we sample the source. A solution is to calculate it once, when we apply color grading. We have to store the luma somewhere, for which we can use the alpha channel of the color-grading result texture. However, this is not possible if the transparency stored in the alpha channel is needed later, for example when layering cameras.

As the alpha channel is typically unused we'll add another pass to *PostFXStack* that both applies color grading and calculates luma, while also keeping the original pass.

```
		Pass {
			Name "Apply Color Grading With Luma"

			HLSLPROGRAM
				#pragma target 3.5
				#pragma vertex DefaultPassVertex
				#pragma fragment ApplyColorGradingWithLumaPassFragment
			ENDHLSL
		}
```

The new fragment function is a copy of `ApplyColorGradingPassFragment` that also calculates luma and stores it in the alpha channel.

```
float4 ApplyColorGradingWithLumaPassFragment (Varyings input) : SV_TARGET {
	float4 color = GetSource(input.screenUV);
	color.rgb = ApplyColorGradingLUT(color.rgb);
	color.a = sqrt(Luminance(color.rgb));
	return color;
}
```

We now need two version of our FXAA pass, one for when the alpha channel contains luma and one for when luma isn't available. We'll keep the current *FXAA* pass and add another *FXAA With Luma* pass for when luma is available. In this case we'll define *FXAA_ALPHA_CONTAINS_LUMA* instead of creating a separate fragment function for it. This works because we're including *FXAAPass* in the pass block itself, so we add the definition before including the file.

```
		Pass {
			Name "FXAA With Luma"

			Blend [_FinalSrcBlend] [_FinalDstBlend]

			HLSLPROGRAM
				#pragma target 3.5
				#pragma vertex DefaultPassVertex
				#pragma fragment FXAAPassFragment
				#define FXAA_ALPHA_CONTAINS_LUMA
				#include "FXAAPass.hlsl"
			ENDHLSL
		}
```

Now we can use conditional compilation to make `GetLuma` return the appropriate color channel: alpha when luma is stored in it and green otherwise.

```
float GetLuma (float2 uv) {
	#if defined(FXAA_ALPHA_CONTAINS_LUMA)
		return GetSource(uv).a;
	#else
		return GetSource(uv).g;
	#endif
}
```

1.6 Keeping Alpha

We prefer to calculate luma, so that will be the default. We'll only switch to green if the data in the alpha channel must be kept unchanged, no matter the reason. This depends on what the rendered image is used for, so must be configurable per camera. Add a toggle option to keep alpha to `CameraSettings` for this purpose, which is disabled by default.

```
	public bool keepAlpha = false;
```

Toggle for keeping alpha.

Pass this toggle along when setting up the post FX stack in `CameraRenderer.Render`. It's related to the HDR toggle as both settings deal with the nature of texture data, so place it before the HDR toggle argument.

```
		postFXStack.Setup(
			context, camera, bufferSize, postFXSettings, cameraSettings.keepAlpha, useHDR,
			colorLUTResolution, cameraSettings.finalBlendMode,
			bufferSettings.bicubicRescaling, bufferSettings.fxaa
		);
```

Then keep track of the toggle in `PostFXStack`.

```
	bool keepAlpha, useHDR;

	…

	public void Setup (
		ScriptableRenderContext context, Camera camera, Vector2Int bufferSize,
		PostFXSettings settings, bool keepAlpha, bool useHDR, int colorLUTResolution,
		…
	) {
		…
		this.keepAlpha = keepAlpha;
		this.useHDR = useHDR;
		…
	}
```

Now `DoFinal` must use the appropriate passes when FXAA is enabled. If we have to keep alpha then we stick with the current passes, otherwise we can switch to the color grading and FXAA passes that include luma in the alpha channel.

```
		if (fxaa.enabled) {
			…
			Draw(
				sourceId, colorGradingResultId,
				keepAlpha ? Pass.ApplyColorGrading : Pass.ApplyColorGradingWithLuma
			);
		}

		if (bufferSize.x == camera.pixelWidth) {
			if (fxaa.enabled) {
				DrawFinal(
					colorGradingResultId, keepAlpha ? Pass.FXAA : Pass.FXAAWithLuma
				);
				buffer.ReleaseTemporaryRT(colorGradingResultId);
			}
			…
		}
		else {
			…

			if (fxaa.enabled) {
				Draw(
					colorGradingResultId, finalResultId,
					keepAlpha ? Pass.FXAA : Pass.FXAAWithLuma
				);
				buffer.ReleaseTemporaryRT(colorGradingResultId);
			}
			…
		}
```

You can check whether this work by toggling the camera's *Keep Alpha* setting. When alpha must be kept our RP is forced to fall back to relying on green instead of luma, which will produce a darker grayscale image. Currently the only reason to keep alpha is when multiple cameras are stacked with transparency.

2 Subpixel Blending

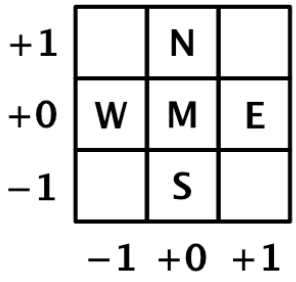
FXAA works by blending adjacent pixels that have a high contrast. So this is not a straightforward uniform blurring of the image. First, the local contrast—the range from lowest to highest luma—has to be calculated around the source pixel. Second—if there is enough contrast—a blend factor has to be chosen based on the contrast. Third, the local contrast gradient has to be investigated to determine a blend direction. Finally, a blend is performed between the original pixel its appropriate neighbor.

2.1 Luma Neighborhood

The local contrast is found by sampling the luma of the pixels in the neighborhood of the source pixel. To make this easy add two optional offset parameters to `GetLuma`, so it can be offset in units of pixels along the U and V dimensions.

```
float GetLuma (float2 uv, float uOffset = 0.0, float vOffset = 0.0) {
	uv += float2(uOffset, vOffset) * GetSourceTexelSize().xy;
	…
}
```

Besides the source pixel, we also have to sample its directly adjacent neighbors, which we'll identify with compass directions. So we end up with five luma values: the middle source pixel plus north, east, south, and west.



Neighborhood samples.

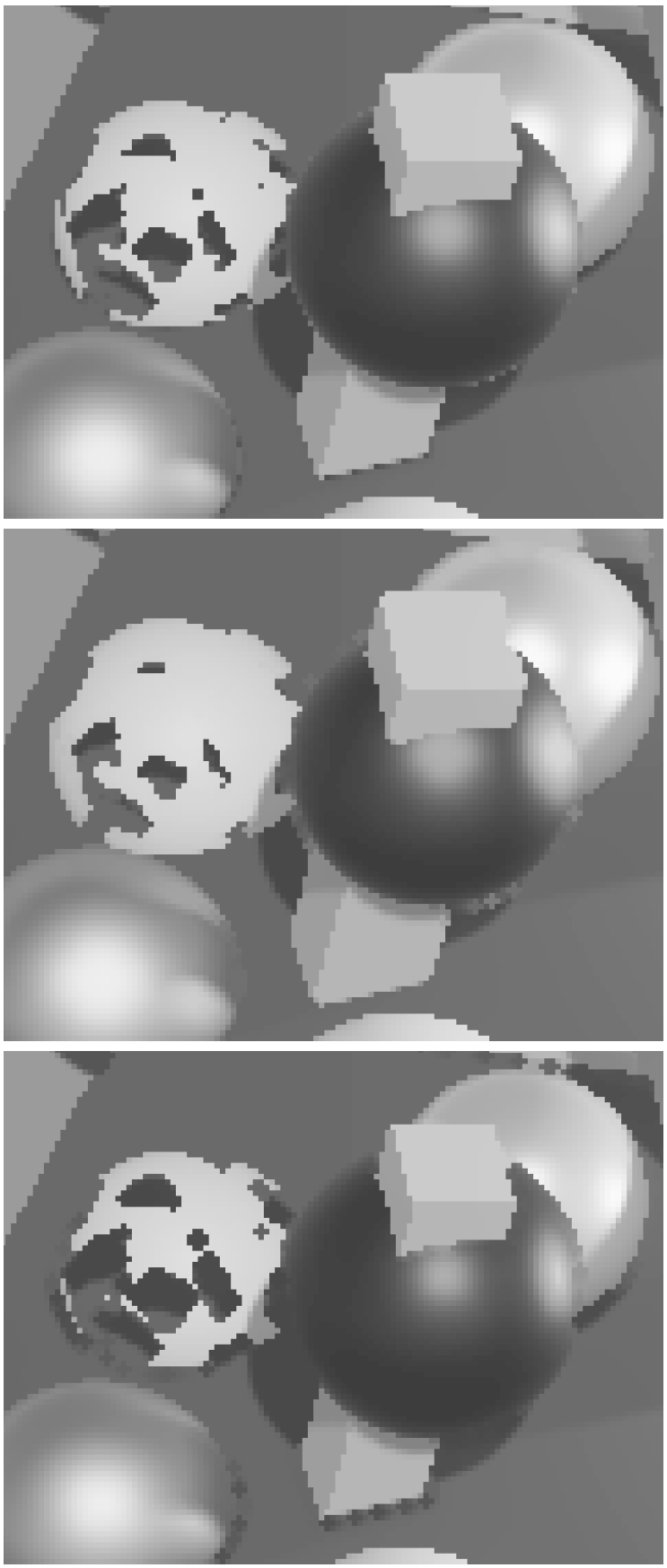
Define a `LumaNeighborhood` struct to keep track of all these together and add a `GetLumaNeighborhood` function that returns that neighborhood. Invoke it in the fragment pass, initially still only returning the middle luma value.

```
struct LumaNeighborhood {
	float m, n, e, s, w;
};

LumaNeighborhood GetLumaNeighborhood (float2 uv) {
	LumaNeighborhood luma;
	luma.m = GetLuma(uv);
	luma.n = GetLuma(uv, 0.0, 1.0);
	luma.e = GetLuma(uv, 1.0, 0.0);
	luma.s = GetLuma(uv, 0.0, -1.0);
	luma.w = GetLuma(uv, -1.0, 0.0);
	return luma;
}

float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	LumaNeighborhood luma = GetLumaNeighborhood(input.screenUV);
	return luma.m;
}
```

To determine the luma range in this neighborhood we need to know what its highest and lowest luma values are. Calculate them and also store them in the neighborhood struct.

```
struct LumaNeighborhood {
	float m, n, e, s, w;
	float highest, lowest;
};

LumaNeighborhood GetLumaNeighborhood (float2 uv) {
	…

	luma.highest = max(max(max(max(luma.m, luma.n), luma.e), luma.s), luma.w);
	luma.lowest = min(min(min(min(luma.m, luma.n), luma.e), luma.s), luma.w);
	return luma;
}
```

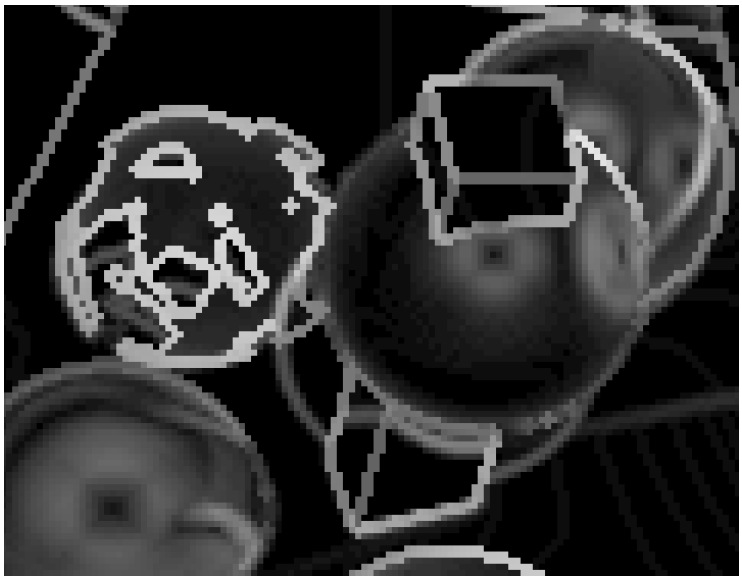
You can observe these and other values by using them for the result of the fragment function, but I don't show the temporary code changes needed for that.



Middle, highest, and lowest luma in neighborhood; zoomed in.

Now also add the luma range to the neighborhood, which is the highest luma minus the lowest luma.

```
struct LumaNeighborhood {
	float m, n, e, s, w;
	float highest, lowest, range;
};

LumaNeighborhood GetLumaNeighborhood (float2 uv) {
	…

	luma.highest = max(max(max(max(luma.m, luma.n), luma.e), luma.s), luma.w);
	luma.lowest = min(min(min(min(luma.m, luma.n), luma.e), luma.s), luma.w);
	luma.range = luma.highest - luma.lowest;
	return luma;
}
```



Luma range in neighborhood.

Note that the luma range visually reveals edges in the image as lines. The lines are two pixels wide because there's a pixel on both sides of every edge. The higher the luma contrast of an edge, the brighter it appears.

2.2 Fixed Threshold

We don't need to blend every pixel, only those whose neighborhood has a high enough contrast. The easiest way to make this distinction is by introducing a contrast threshold. If the neighborhood luma range doesn't reach this threshold then the pixel needs no blending. We'll name this the fixed threshold, because there's also a relative threshold.

Add a slider to configure a fixed threshold to our `CameraBufferSettings.FXAA` struct. The original FXAA algorithm has this threshold as well, with the following code documentation:

```
// Trims the algorithm from processing darks.
//   0.0833 - upper limit (default, the start of visible unfiltered edges)
//   0.0625 - high quality (faster)
//   0.0312 - visible limit (slower)
```

Although the documentation mentions that it trims dark areas, it trims based on contrast, so regardless whether it's bright or dark. We will use the same range as indicated by the original FXAA documentation.

```
	public struct FXAA {

		public bool enabled;

		[Range(0.0312f, 0.0833f)]
		public float fixedThreshold;
	}
```

Let's also use the same default as the original, which we set in `CustomRenderPipelineAsset`.

```
	CameraBufferSettings cameraBuffer = new CameraBufferSettings {
		allowHDR = true,
		renderScale = 1f,
		fxaa = new CameraBufferSettings.FXAA {
			fixedThreshold = 0.0833f
		}
	};
```

Fixed threshold slider.

Next, add an *_FXAAConfig* shader property identifier to `PostFXStack`.

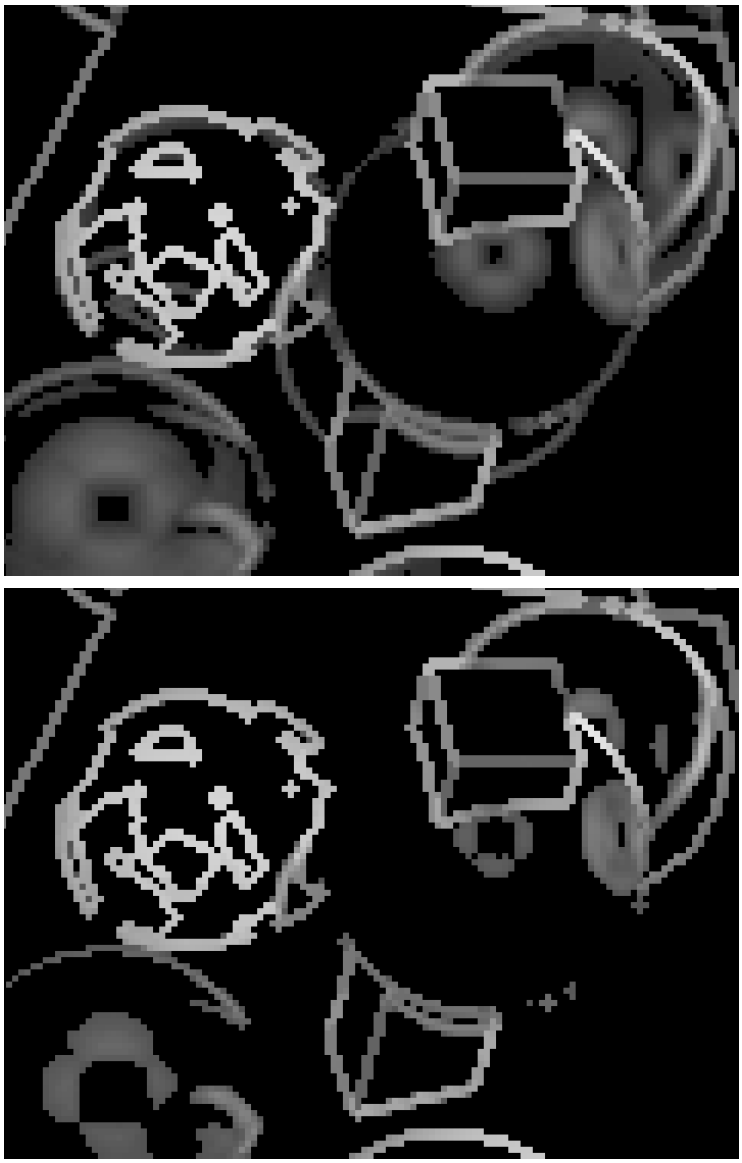
```
	int fxaaConfigId = Shader.PropertyToID("_FXAAConfig");
```

We'll send the FXAA configuration to the GPU as a vector, initially with only the fixed threshold in its first component. Do this in `DoFinal` if FXAA is enabled.

```
		if (fxaa.enabled) {
			buffer.SetGlobalVector(fxaaConfigId, new Vector4(fxaa.fixedThreshold, 0f));
			buffer.GetTemporaryRT(
				colorGradingResultId, bufferSize.x, bufferSize.y, 0,
				FilterMode.Bilinear, RenderTextureFormat.Default
			);
			…
		}
```

Add the *_FXAAConfig* vector to *FXAAPass*, along with a `CanSkipFXAA` function that takes a `LumaNeighborhood` and returns whether its range is less than the fixed threshold. Then return zero in `FXAAPassFragment` if we can skip FXAA.

```
float4 _FXAAConfig;

…

bool CanSkipFXAA (LumaNeighborhood luma) {
	return luma.range < _FXAAConfig.x;
}

float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	LumaNeighborhood luma = GetLumaNeighborhood(input.screenUV);

	if (CanSkipFXAA(luma)) {
		return 0.0;
	}

	return luma.range;
}
```



Fixed threshold set to minimum and maximum.

The pixels that are skipped by FXAA are now solid black. Low-contrast regions are now visually eliminated. How much remains depends on the threshold.

2.3 Relative Threshold

FXAA also has a second threshold, which is relative to the brightest luma of each neighborhood. The brighter the neighborhood, the higher the contrast must be to matter. The original FXAA code has the following documentation for its values:

```
// The minimum amount of local contrast required to apply algorithm.
//   0.333 - too little (faster)
//   0.250 - low quality
//   0.166 - default
//   0.125 - high quality
//   0.063 - overkill (slower)
```

Add a slider for this relative threshold to `CameraBufferSettings.FXAA` as well.

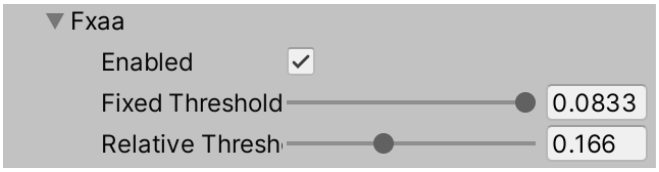
```
		[Range(0.063f, 0.333f)]
		public float relativeThreshold;
```

Again with the same default as the original in `CustomRenderPipelineAsset`.

```
		fxaa = new CameraBufferSettings.FXAA {
			fixedThreshold = 0.0833f,
			relativeThreshold = 0.166f
		}
```

Then put it in the second component of the FXAA config vector in `PostFXStack.DoFinal`.

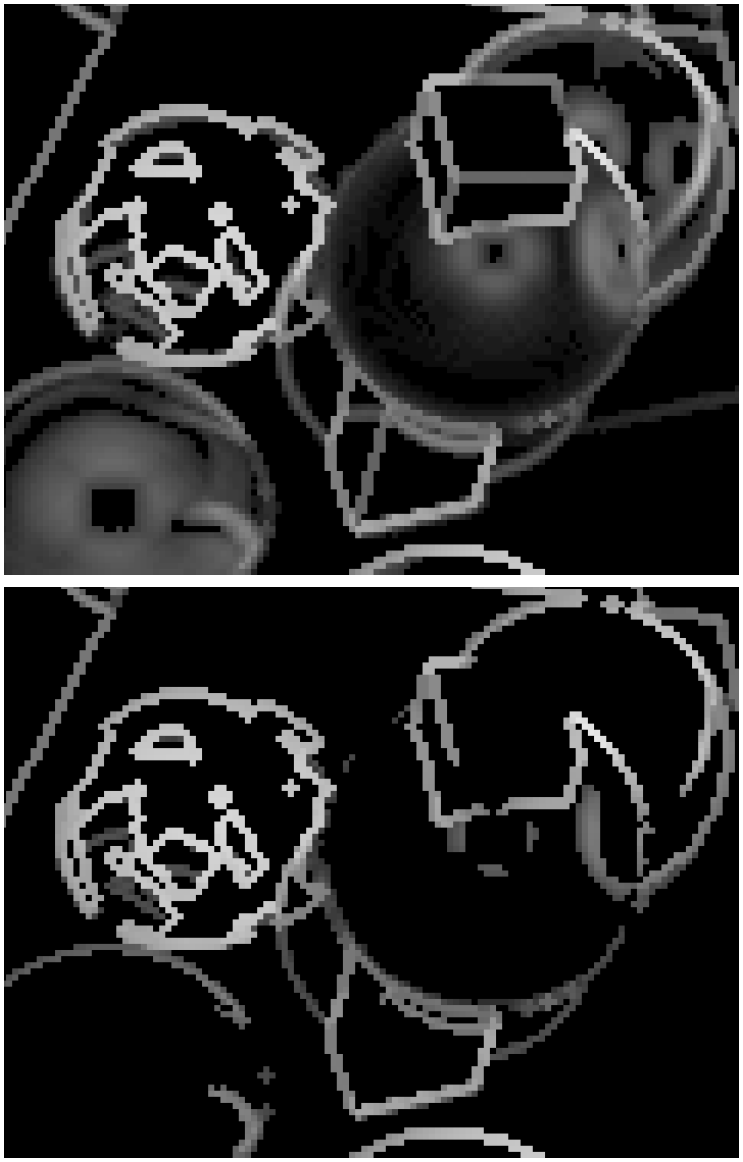
```
			buffer.SetGlobalVector(fxaaConfigId, new Vector4(
				fxaa.fixedThreshold, fxaa.relativeThreshold
			));
```



Two threshold sliders.

To apply the relative instead of the fixed threshold change `CanSkipFXAA` in *FXAAPass* so it checks whether the luma range is less than the second threshold scaled by the highest luma.

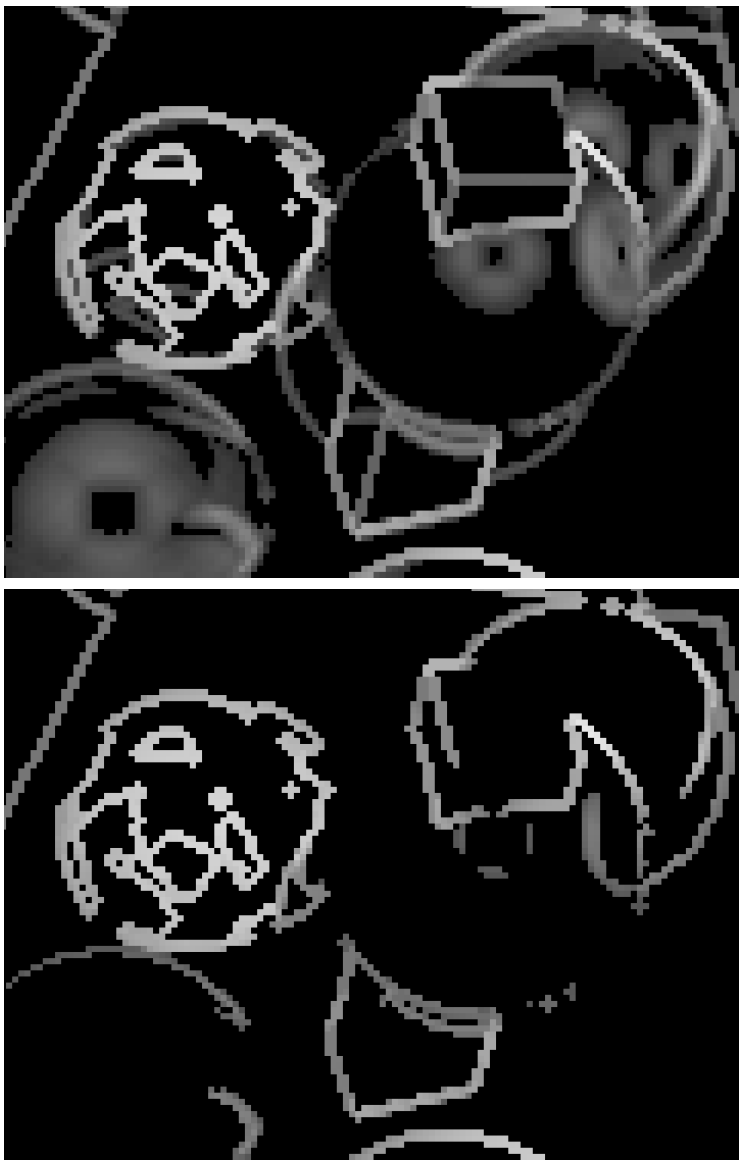
```
bool CanSkipFXAA (LumaNeighborhood luma) {
	return luma.range < _FXAAConfig.y * luma.highest;
}
```



Relative threshold set to minimum and maximum.

To apply both thresholds compare with the largest one.

```
bool CanSkipFXAA (LumaNeighborhood luma) {
	return luma.range < max(_FXAAConfig.x, _FXAAConfig.y * luma.highest);
}
```



Both thresholds set to minimum and maximum.

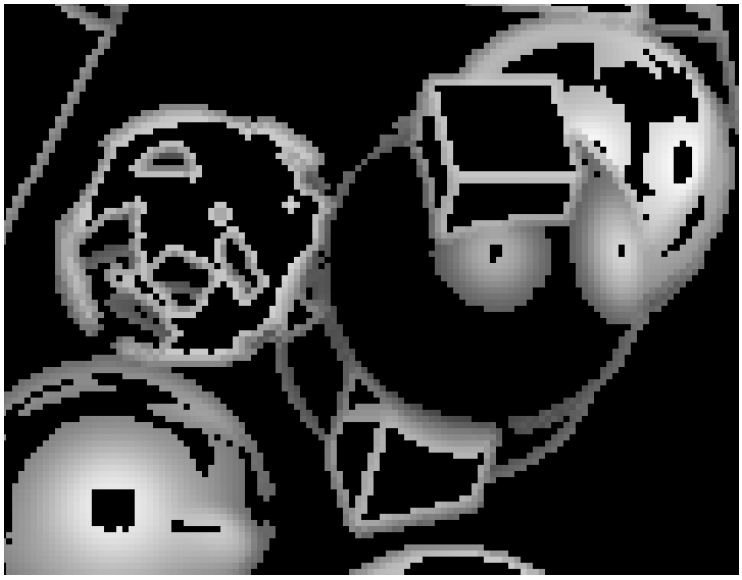
From now on I'll always use the lowest thresholds so that the most pixels are affected.

2.4 Blend Factor

The only correct way to increase the visual quality of an edge is to increase the resolution of the image. However, FXAA only has original image data to work with. The best that it can do is make a guess about the subpixel data that is missing. It does this by blending the middle pixel with one of its neighbors. At its most extreme this would be a simple average of both pixels, but the exact blend factor is the result of a filter that depends on the contrast of the pixel and the average of its neighbors. We'll visualize this in steps.

Begin by creating a `GetSubpixelBlendFactor` function that returns the average of the four neighbors in the neighborhood. Use that for the result of `FXAAPassFragment`.

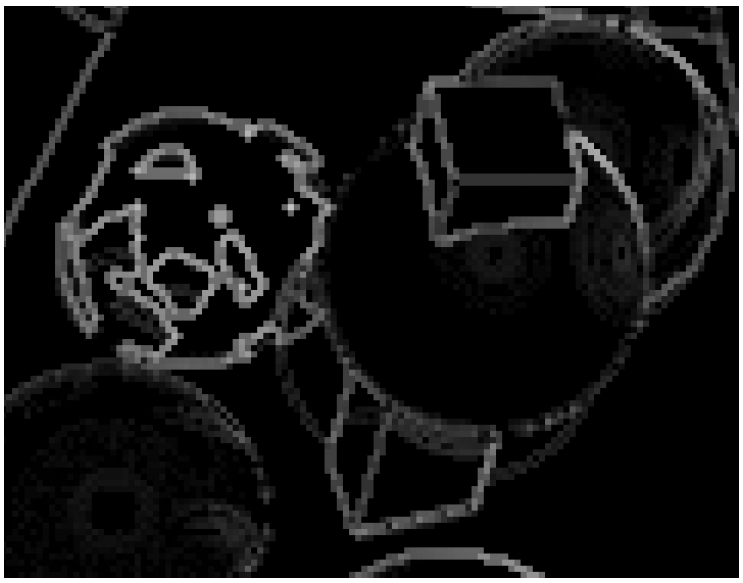
```
float GetSubpixelBlendFactor (LumaNeighborhood luma) {
	float filter = luma.n + luma.e + luma.s + luma.w;
	filter *= 1.0 / 4;
	return filter;
}

float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	…

	return GetSubpixelBlendFactor(luma);
}
```



Low-pass filter.

The result is a low-pass filter applied to the luma around those pixels that weren't skipped. The next step is to turn this into a high-pass filter, by taking the absolute difference between the neighbor average and the middle.

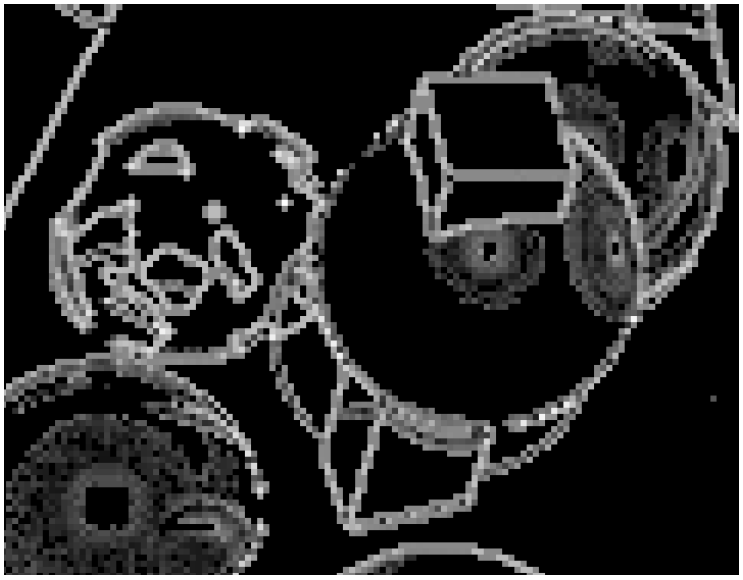
```
	filter = abs(filter - luma.m);
	return filter;
```



High-pass filter.

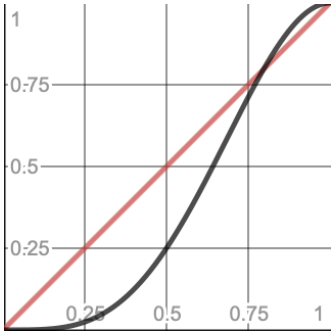
After that we normalize the filter by dividing it by the luma range.

```
	filter = filter / luma.range;
	return filter;
```



Normalized filter.

At this point the result is too strong to use as a blend factor. FXAA modifies the filter by applying the squared `smoothstep` function to it.



Linear and squared smoothstep.

```
	filter = smoothstep(0, 1, filter);
	return filter * filter;
```

Smoothed filter.

The quality of the filter can be improved by also incorporating the diagonal neighbors into it, so add their luma values to the neighborhood.

```
struct LumaNeighborhood {
	float m, n, e, s, w, ne, se, sw, nw;
	float highest, lowest, range;
};

LumaNeighborhood GetLumaNeighborhood (float2 uv) {
	LumaNeighborhood luma;
	luma.m = GetLuma(uv);
	luma.n = GetLuma(uv, 0.0, 1.0);
	luma.e = GetLuma(uv, 1.0, 0.0);
	luma.s = GetLuma(uv, 0.0, -1.0);
	luma.w = GetLuma(uv, -1.0, 0.0);
	luma.ne = GetLuma(uv, 1.0, 1.0);
	luma.se = GetLuma(uv, 1.0, -1.0);
	luma.sw = GetLuma(uv, -1.0, -1.0);
	luma.nw = GetLuma(uv, -1.0, 1.0);

	…
}
```

Because the diagonal neighbors are spatially farther away from the middle they should matter less that the direct neighbors. We factor this into our average by doubling the weights of the direct neighbors. This acts like a 3×3 tent filter, without the middle.

1	2	1
2		2
1	2	1

Neighbor weights.

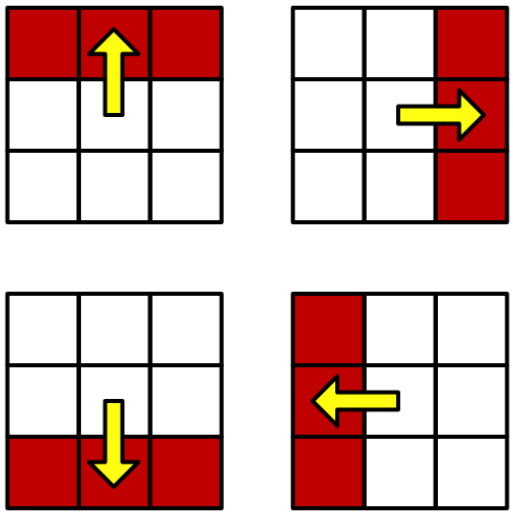
We now also have to saturate the normalized filter, because the highest value that we stored doesn't take the diagonal samples into account, thus after the division we could still have a value that exceeds 1.

```
float GetSubpixelBlendFactor (LumaNeighborhood luma) {
	float filter = 2.0 * (luma.n + luma.e + luma.s + luma.w);
	filter += luma.ne + luma.nw + luma.se + luma.sw;
	filter *= 1.0 / 12.0;
	filter = saturate(filter / luma.range);
	filter = smoothstep(0, 1, filter);
	return filter * filter;
}
```

Expanded filter and absolute difference between both filters.

2.5 Blend Direction

After determining the blend factor the next step is to decide which two pixels to blend. FXAA blends the middle pixel with one of its direct neighbors, so either the north, east, south, or west neighbor. Which of those four pixels is selected depends on the direction of the contrast gradient. In the simplest case, the middle pixel touches either a horizontal or a vertical edge between two contrasting regions. In case of a horizontal edge, it should be either the north or the south neighbor, depending on whether the middle is below or above the edge. Otherwise, it should be either the east or the west neighbor, depending on whether the middle is on the left or right side of the edge.



Four possible blend directions.

Edges often aren't perfectly horizontal or vertical, but we pick the best approximation, by comparing the horizontal and vertical contrast in the neighborhood. If there is a horizontal edge then there will be strong vertical contrast, either above or below the middle. We measure this by adding north and south, subtracting the middle twice, and taking the absolute of that. The same logic applies to vertical edges, but with east and west instead. If the horizontal result is greater than the vertical then we declare it a horizontal edge. Create a function that indicates this, given a neighborhood.

```
bool IsHorizontalEdge (LumaNeighborhood luma) {
	float horizontal = abs(luma.n + luma.s - 2.0 * luma.m);
	float vertical = abs(luma.e + luma.w - 2.0 * luma.m);
	return horizontal >= vertical;
}
```

We can improve the quality of our edge orientation detection by including the diagonal neighbors as well. For the horizontal edge, we perform the same calculation for the three pixels one step to the east and for the three pixels one step to the west, summing the results. Again, these additional values are further away from the middle, so we halve their relative importance by doubling the weight of the middle contrast. The same applies to the vertical edge contrast, but with north and south offsets.

```
bool IsHorizontalEdge (LumaNeighborhood luma) {
	float horizontal =
		2.0 * abs(luma.n + luma.s - 2.0 * luma.m) +
		abs(luma.ne + luma.se - 2.0 * luma.e) +
		abs(luma.nw + luma.sw - 2.0 * luma.w);
	float vertical =
		2.0 * abs(luma.e + luma.w - 2.0 * luma.m) +
		abs(luma.ne + luma.nw - 2.0 * luma.n) +
		abs(luma.se + luma.sw - 2.0 * luma.s);
	return horizontal >= vertical;
}
```

Now introduce an `FXAAEdge` struct to contain information about the detected edge. At this point that's only whether it is horizontal. Create a `GetFXAAEdge` method that returns that information, given a neighborhood.

```
struct FXAAEdge {
	bool isHorizontal;
};

FXAAEdge GetFXAAEdge (LumaNeighborhood luma) {
	FXAAEdge edge;
	edge.isHorizontal = IsHorizontalEdge(luma);
	return edge;
}
```

Get the edge data in `FXAAPassFragment`, then use it to visualize the orientation of the edges, for example by making horizontal ones red and vertical ones white. At this point we do not care about the blend factor.

```
float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	LumaNeighborhood luma = GetLumaNeighborhood(input.screenUV);

	if (CanSkipFXAA(luma)) {
		return 0.0;
	}

	FXAAEdge edge = GetFXAAEdge(luma);

	return edge.isHorizontal ? float4(1.0, 0.0, 0.0, 0.0) : 1.0;
}
```

Horizontal edges are red, vertical edges are white.

Knowing the edge orientation tells us in what dimension we have to blend. If it's horizontal then we'll blend vertically across the edge, otherwise it's vertical and we'll blend horizontally across the edge. How far it is to the next pixel in UV space depends on the pixel size, which depends on the blend direction. So let's add the size of the pixel step to `FXAAEdge` and initialize it in `GetFXAAEdge`.

```
struct FXAAEdge {
	bool isHorizontal;
	float pixelStep;
};

FXAAEdge GetFXAAEdge (LumaNeighborhood luma) {
	FXAAEdge edge;
	edge.isHorizontal = IsHorizontalEdge(luma);
	if (edge.isHorizontal) {
		edge.pixelStep = GetSourceTexelSize().y;
	}
	else {
		edge.pixelStep = GetSourceTexelSize().x;
	}

	return edge;
}
```

Next, we have to determine whether we should blend in the positive or negative direction. We do this by comparing the contrast—the luma gradient—on either side of the middle in the appropriate directions. If we have a horizontal edge then north is the positive neighbor and south is the negative one. If we have a vertical edge instead then east is the positive neighbor and west is the negative one.

If the positive gradient is smaller than the negative one then the middle is on the right side of the ege and we have to blend in the negative direction, which we do by negating the step.

```
	float lumaP, lumaN;
	if (edge.isHorizontal) {
		edge.pixelStep = GetSourceTexelSize().y;
		lumaP = luma.n;
		lumaN = luma.s;
	}
	else {
		edge.pixelStep = GetSourceTexelSize().x;
		lumaP = luma.e;
		lumaN = luma.w;
	}
	float gradientP = abs(lumaP - luma.m);
	float gradientN = abs(lumaN - luma.m);

	if (gradientP < gradientN) {
		edge.pixelStep = -edge.pixelStep;
	}
```

Now we can visualize the blend direction, for example by making positive edges red and negative edges white.

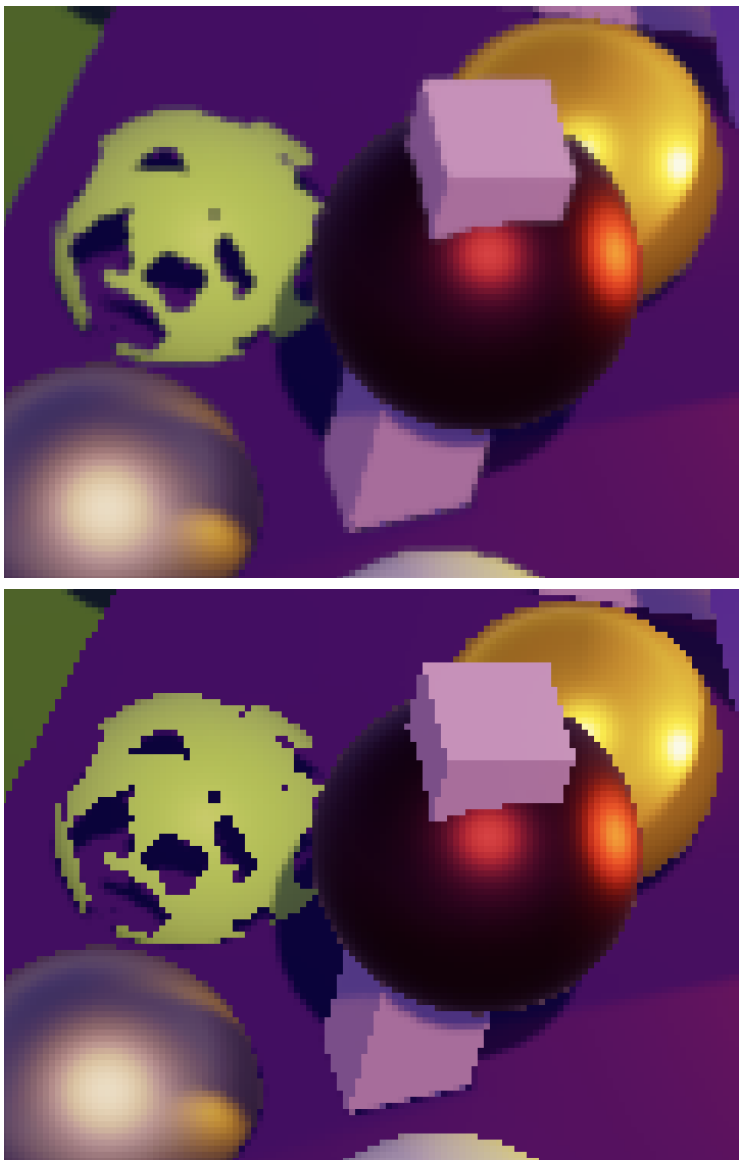
```
	return edge.pixelStep > 0.0 ? float4(1.0, 0.0, 0.0, 0.0) : 1.0;
```

Positive edges (left and bottom sides) are red, negative edges are white.

2.6 Final Blend

At this point we can get both the blend factor and know in which direction to blend. The final result is obtained by using the blend factor to linearly interpolate between the middle pixel and its neighbor in the appropriate direction. We can do this by simply sampling the image with an offset equal to the pixel step scaled by the blend factor. Also, we have to return the original pixel if we skip it.

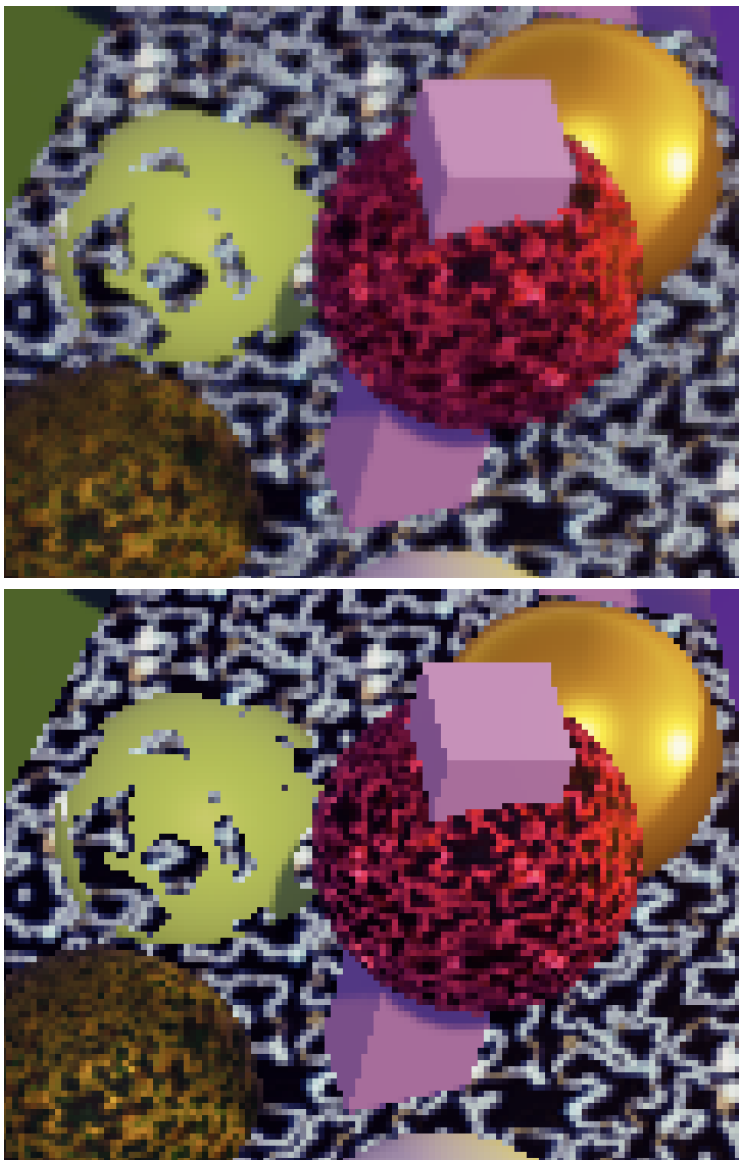
```
float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	LumaNeighborhood luma = GetLumaNeighborhood(input.screenUV);

	if (CanSkipFXAA(luma)) {
		return GetSource(input.screenUV);
	}

	FXAAEdge edge = GetFXAAEdge(luma);

	float blendFactor = GetSubpixelBlendFactor(luma);
	float2 blendUV = input.screenUV;
	if (edge.isHorizontal) {
		blendUV.y += blendFactor * edge.pixelStep;
	}
	else {
		blendUV.x += blendFactor * edge.pixelStep;
	}
	return GetSource(blendUV);
}
```



With and without blending.

2.7 Blend Strength

FXAA doesn't only affect obvious high-contrast edges, it blend anything that has high enough contrast, including isolated pixels. While this helps mitigate fireflies it also aggressively blurs small details, which is typically the biggest complaint against FXAA.



Circuitry materials with small details, with and without subpixel blending.

FXAA can control the strength of subpixel blending by simply scaling down its blend factor. Here is the original documentation for it:

```
// Choose the amount of sub-pixel aliasing removal.
// This can effect sharpness.
//   1.00 - upper limit (softer)
//   0.75 - default amount of filtering
//   0.50 - lower limit (sharper, less sub-pixel aliasing removal)
//   0.25 - almost off
//   0.00 - completely off
```

We make this configurable as well, by adding a 0–1 slider for subpixel blending to **`CameraSettings.FXAA`**.

```
		[Range(0f, 1f)]
		public float subpixelBlending;
```

Give it the same 75% strength default as the original in **`CustomRenderPipelineAsset`**.

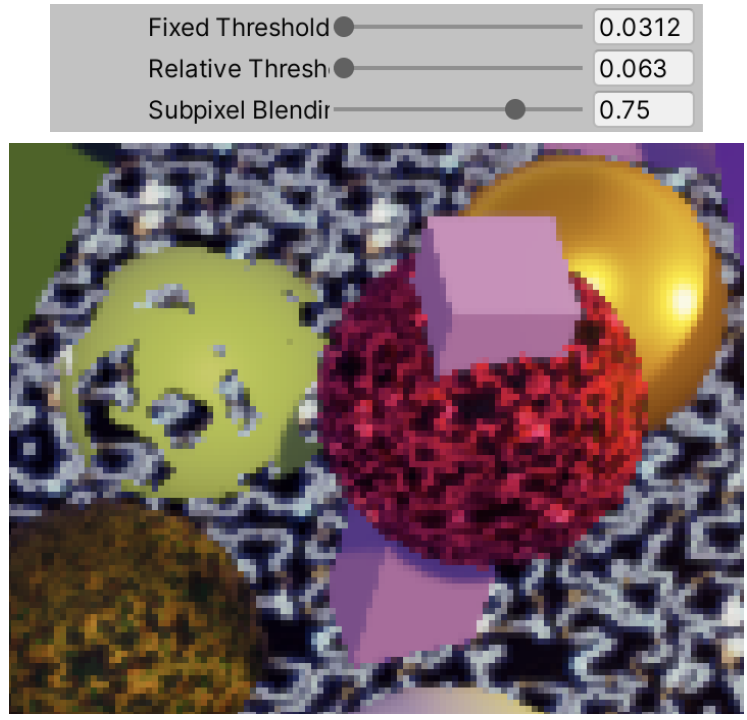
```
		fxaa = new CameraBufferSettings.FXAA {
			fixedThreshold = 0.0833f,
			relativeThreshold = 0.166f,
			subpixelBlending = 0.75f
		}
```

Then add it to the FXAA config vector in `PostFXStack.DoFinal`.

```
			buffer.SetGlobalVector(fxaaConfigId, new Vector4(
				fxaa.fixedThreshold, fxaa.relativeThreshold, fxaa.subpixelBlending
			));
```

And apply it to the blend factor at the end of `GeSubpixelBlendFactor`.

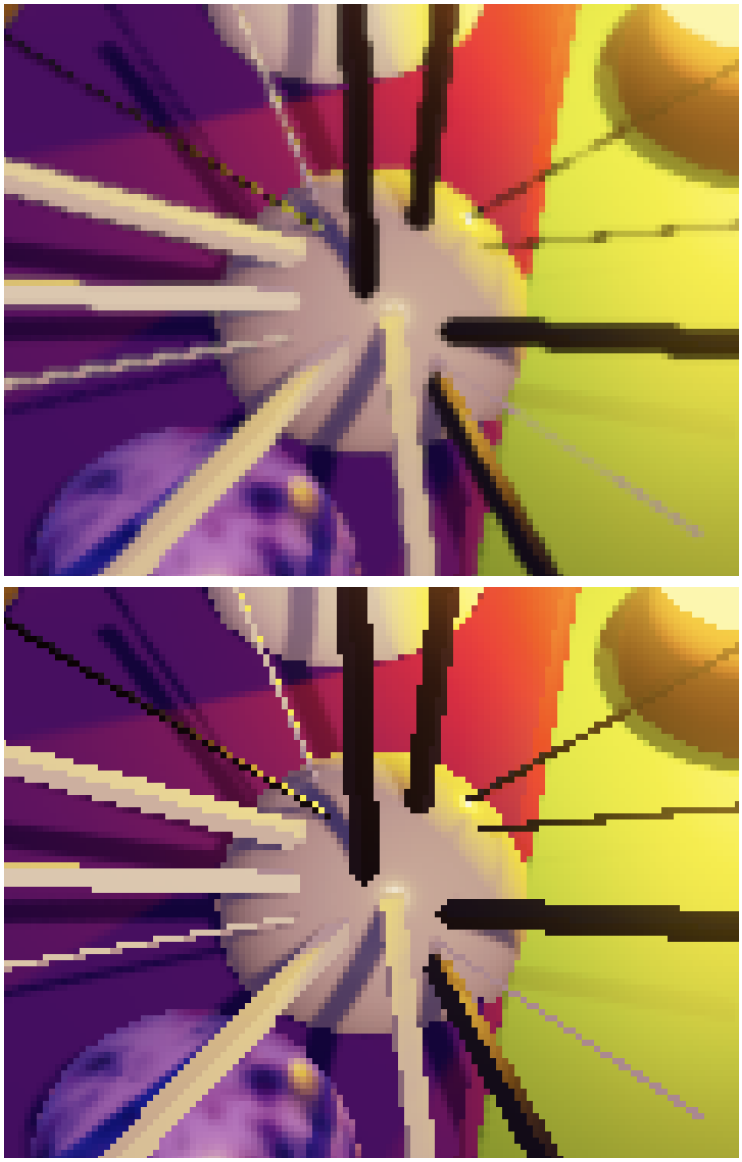
```
float GetSubpixelBlendFactor (LumaNeighborhood luma) {
	…
	return filter * filter * _FXAAConfig.z;
}
```



Subpixel blending reduced to 75%.

3 Blending Along Edges

Because the pixel blend factor is determined inside a 3×3 block it can only smooth out features of that scale. But edges can be longer than that. A pixel can end up somewhere on a long step of an angled edge staircase. While locally the edge is either horizontal or vertical, the true edge is often at another angle. If we knew this true edge then we could better match the blend factors of adjacent pixels, smoothing the edge across its entire length.



Pincushion geometry with and without subpixel blending at full strength.

In contrast, at render scale 2 we get better edges because the higher resolution can smooth staircases a bit across their entire lengths. It's also possible apply FXAA on top of an increased render scale in order to get even smoother results, but currently that doesn't make much of a difference for edge quality.

Render scale 2 without FXAA.

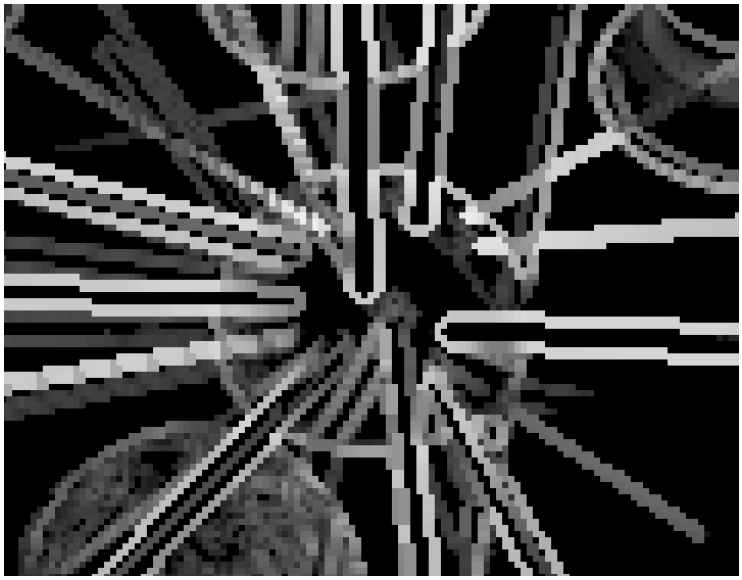
3.1 Edge Luma

To figure out what kind of edge we're dealing with we have to keep track of more information. We know that the middle pixel of the 3×3 block is on one side of the edge, while at least one of the other pixels is on the opposite side. To further identify the edge we need to know its luma gradient. We already figured this out in `GetFXAAEdge`. We now need to keep track of both this gradient and the luma on the other side of the edge, so add them to the edge data.

```
struct FXAAEdge {
	bool isHorizontal;
	float pixelStep;
	float lumaGradient, otherLuma;
};

FXAAEdge GetFXAAEdge (LumaNeighborhood luma) {
	…

	if (gradientP < gradientN) {
		edge.pixelStep = -edge.pixelStep;
		edge.lumaGradient = gradientN;
		edge.otherLuma = lumaN;
	}
	else {
		edge.lumaGradient = gradientP;
		edge.otherLuma = lumaP;
	}

	return edge;
}
```

Introduce a `GetEdgeBlendFactor` function that returns a separate blend factor for edges. It needs the luma neighborhood, edge data, and pixel UV coordinates to do this, so add parameters for those. We'll begin by returning the luma gradient of the edge. Adjust `FXAAPassFragment` so it visualized the new edge blend factor only.

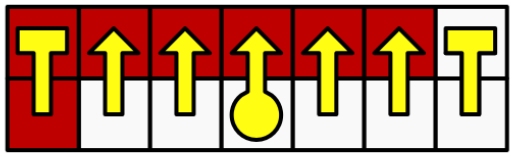
```
float GetEdgeBlendFactor (LumaNeighborhood luma, FXAAEdge edge, float2 uv) {
	return edge.lumaGradient;
}

float4 FXAAPassFragment (Varyings input) : SV_TARGET {
	LumaNeighborhood luma = GetLumaNeighborhood(input.screenUV);

	if (CanSkipFXAA(luma)) {
		return 0.0;
	}

	FXAAEdge edge = GetFXAAEdge(luma);

	float blendFactor = GetEdgeBlendFactor (luma, edge, input.screenUV);
	return blendFactor;
	float2 blendUV = input.screenUV;
	…
}
```



Edge gradients.

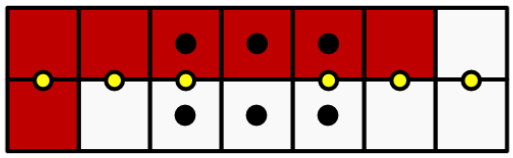
3.2 Tracing the Edge

We have to figure out the relative location of the pixel along the horizontal or vertical edge segment. The only way to do this is by walking along the edge in both directions until we find the end points. This can be done by sampling pixel pairs along the edge and checking whether they still resemble the edge that we initially detected.



Searching for the end points.

But we don't need to sample two pixels each step. We can make do with a single sample in between them, which gives us the average of their luma. This will be enough to determine the end of an edge.



Search (yellow) and neighborhood (black) samples.

To perform this search the first thing we have to do in `GetEdgeBlendFactor` is determine the UV coordinates for sampling on the edge. We have to offset the original UV coordinates half a pixel step toward the edge.

```
float GetEdgeBlendFactor (LumaNeighborhood luma, FXAAEdge edge, float2 uv) {
	float2 edgeUV = uv;
	if (edge.isHorizontal) {
		edgeUV.y += 0.5 * edge.pixelStep;
	}
	else {
		edgeUV.x += 0.5 * edge.pixelStep;
	}

	return edge.lumaGradient;
}
```

After that the UV offset for a single step along the edge depends on its orientation. It's either horizontal or vertical.

```
	float2 edgeUV = uv;
	float2 uvStep = 0.0;
	if (edge.isHorizontal) {
		edgeUV.y += 0.5 * edge.pixelStep;
		uvStep.x = GetSourceTexelSize().x;
	}
	else {
		edgeUV.x += 0.5 * edge.pixelStep;
		uvStep.y = GetSourceTexelSize().y;
	}
```

What we'll do is determine the contrast between the sampled luma values and the luma average on the originally detected edge. If this contrast becomes too great then we've gone off the edge. FXAA uses a quarter of the luma gradient of the edge as the threshold for this check. So we have to keep track of this threshold and the inital edge luma average.

```
	float edgeLuma = 0.5 * (luma.m + edge.otherLuma);
	float gradientThreshold = 0.25 * edge.lumaGradient;

	return edge.lumaGradient;
```

We start by going a single step in the positive direction. Determine the postive-offset UV coordinates, calculate the luma gradient between that offset and the original edge, and check whether it equals or exceeds the threshold. That tells us whether we're at the end of the edge in the positive direction. If we directly visualize this check then we'll see only those pixels that are directly next to the endpoint of an edge.

```
	float edgeLuma = 0.5 * (luma.m + edge.otherLuma);
	float gradientThreshold = 0.25 * edge.lumaGradient;

	float2 uvP = edgeUV + uvStep;
	float lumaGradientP = abs(GetLuma(uvP) - edgeLuma);
	bool atEndP = lumaGradientP >= gradientThreshold;

	return atEndP;
```

One step to the end in the positive direction.

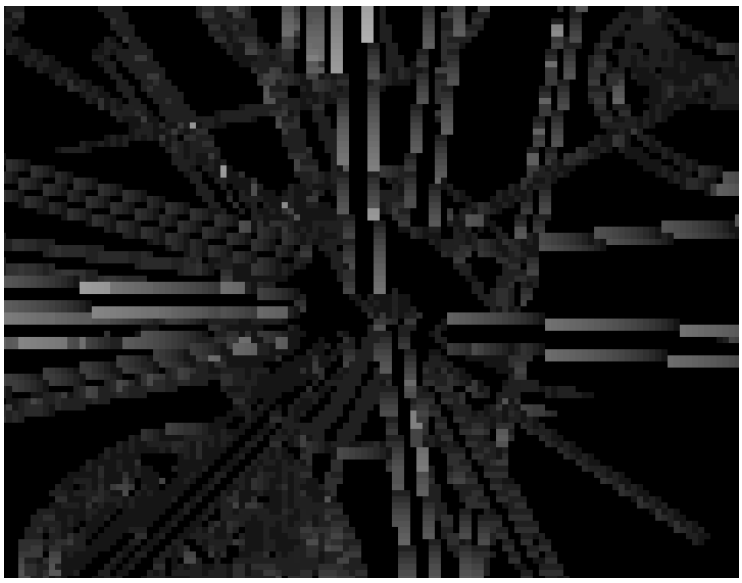
To walk the entire edge we have follow this step with a loop that repeats the process as long as we're not yet at the end. We also have to terminate this process at some point so it cannot go on forever, say after at most 100 steps, so the loops should allow 99 more.

```
	float2 uvP = edgeUV + uvStep;
	float lumaGradientP = abs(GetLuma(uvP) - edgeLuma);
	bool atEndP = lumaGradientP >= gradientThreshold;

	for (int i = 0; i < 99 && !atEndP; i++) {
		uvP += uvStep;
		lumaGradientP = abs(GetLuma(uvP) - edgeLuma);
		atEndP = lumaGradientP >= gradientThreshold;
	}
```

Up to 100 steps in the positive direction.

Once we're done searching we can find the distance to the positive end in UV space by subtracting the appropriate original UV coordinate component from the final offset component. We can then visualize the distance, scaling it up to make it easier to see.

```
	float distanceToEndP;
	if (edge.isHorizontal) {
		distanceToEndP = uvP.x - uv.x;
	}
	else {
		distanceToEndP = uvP.y - uv.y;
	}

	return 10.0 * distanceToEndP;
```



Distance to positive end in UV space, ×10.

3.3 Negative Direction

We also have to do the same in the negative direction, so duplicate the relevant code and adjust it appropriately.

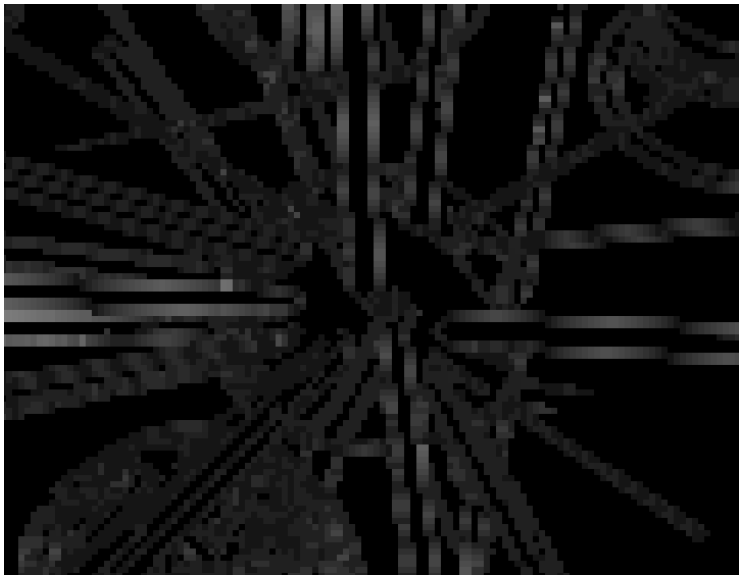
```
	float2 uvP = edgeUV + uvStep;
	float lumaGradientP = abs(GetLuma(uvP) - edgeLuma);
	bool atEndP = lumaGradientP >= gradientThreshold;

	for (int i = 0; i < 99 && !atEndP; i++) {
		uvP += uvStep;
		lumaGradientP = abs(GetLuma(uvP) - edgeLuma);
		atEndP = lumaGradientP >= gradientThreshold;
	}

	float2 uvN = edgeUV - uvStep;
	float lumaGradientN = abs(GetLuma(uvN) - edgeLuma);
	bool atEndN = lumaGradientN >= gradientThreshold;

	for (int i = 0; i < 99 && !atEndN; i++) {
		uvN -= uvStep;
		lumaGradientN = abs(GetLuma(uvN) - edgeLuma);
		atEndN = lumaGradientN >= gradientThreshold;
	}
```

Then determine the distance to the negative end, which works the same as for the positive end but negated.

```
	float distanceToEndP, distanceToEndN;
	if (edge.isHorizontal) {
		distanceToEndP = uvP.x - uv.x;
		distanceToEndN = uv.x - uvN.x;
	}
	else {
		distanceToEndP = uvP.y - uv.y;
		distanceToEndN = uv.y - uvN.y;
	}
```

We can now find the distance to the nearest end of the edge and visualize it.

```
	float distanceToNearestEnd;
	if (distanceToEndP <= distanceToEndN) {
		distanceToNearestEnd = distanceToEndP;
	}
	else {
		distanceToNearestEnd = distanceToEndN;
	}

	return 10.0 * distanceToNearestEnd;
```

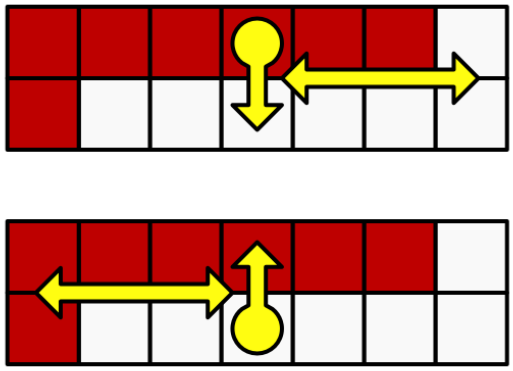


Distance to nearest end.

Note that the found distances appear to make sense in most cases but not always. Because FXAA is an approximation it will sometimes incorrectly guess or miss the end of an edge.

3.4 Blending on a Single Side

At this point we know the distance to the nearest end point of the edge, which we can use to determine the blend factor. But we'll only do that in the direction where the edge is slanting towards the region that contains the middle pixel. This ensures that we only blend pixels on one side of the edge.



Choosing which side to blend.

To determine the direction we need to know the direction of the last gradient we got while searching. To make this possible we'll change our code to keep track of the luma deltas instead of the absolute gradients.

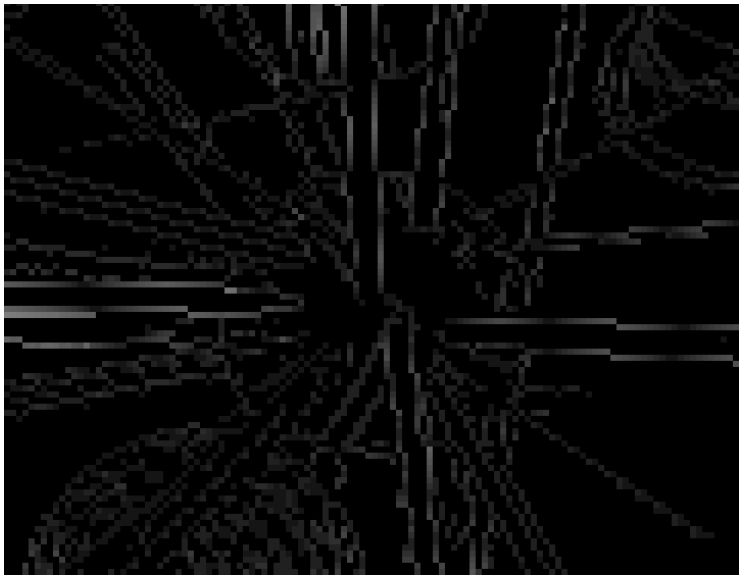
```
	float2 uvP = edgeUV + uvStep;
	float lumaDeltaP = GetLuma(uvP) - edgeLuma;
	bool atEndP = abs(lumaDeltaP) >= gradientThreshold;

	for (int i = 0; i < 99 && !atEndP; i++) {
		uvP += uvStep;
		lumaDeltaP = GetLuma(uvP) - edgeLuma;
		atEndP = abs(lumaDeltaP) >= gradientThreshold;
	}

	float2 uvN = edgeUV - uvStep;
	float lumaDeltaN = GetLuma(uvN) - edgeLuma;
	bool atEndN = abs(lumaDeltaN) >= gradientThreshold;

	for (int i = 0; i < 99 && !atEndN; i++) {
		uvN -= uvStep;
		lumaDeltaN = GetLuma(uvN) - edgeLuma;
		atEndN = abs(lumaDeltaN) >= gradientThreshold;
	}
```

Now we can determine the sign of the final delta. We can do this by checking whether it's greater than or equal to zero.

```
	float distanceToNearestEnd;
	bool deltaSign;
	if (distanceToEndP <= distanceToEndN) {
		distanceToNearestEnd = distanceToEndP;
		deltaSign = lumaDeltaP >= 0;
	}
	else {
		distanceToNearestEnd = distanceToEndN;
		deltaSign = lumaDeltaN >= 0;
	}
```

If the final sign matches the sign of the original edge then we're moving away from the edge and should skip blending, by returning zero.

```
    if (deltaSign == (luma.m - edgeLuma >= 0)) {
        return 0.0;
    }
    else {
        return 10.0 * distanceToNearestEnd;
    }
```



Distances for single side only.

3.5 Final Blend Factor

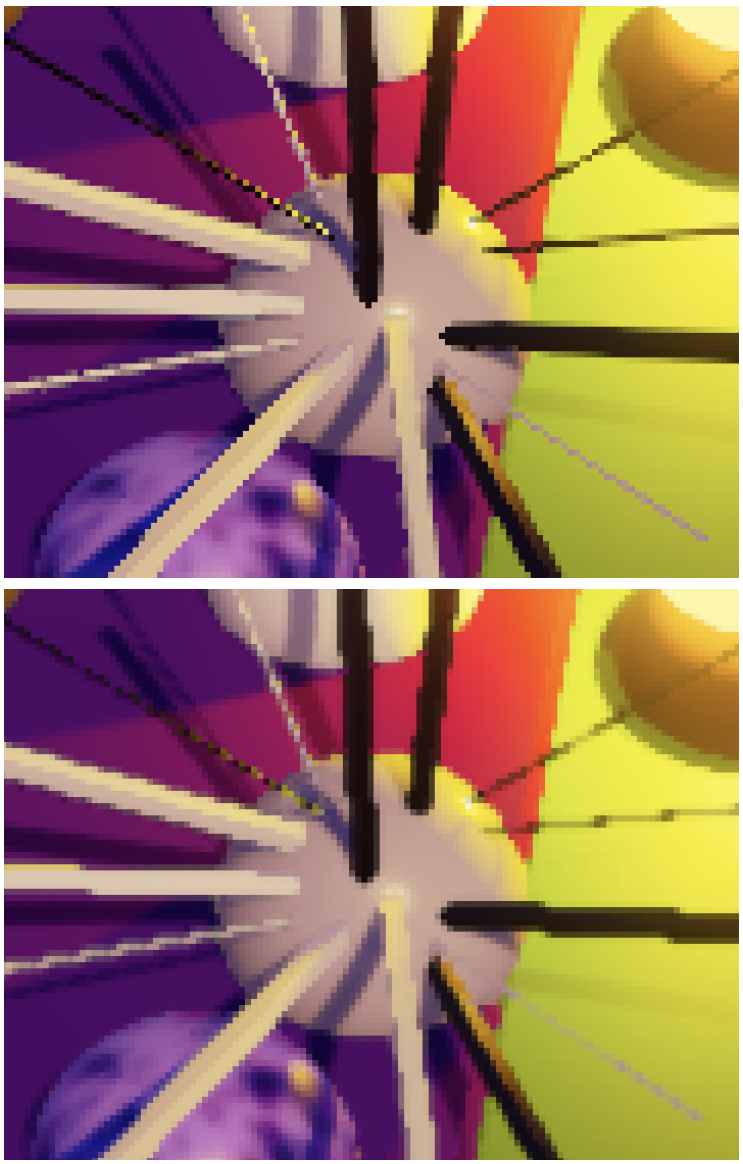
If we're on the correct side of the edge then we blend by a factor of 0.5 minus the relative distance to the nearest end point along the edge. This means that we blend more the closer we are to the end point and won't blend at all in the middle of the edge.

```
    if (deltaSign == (luma.m - edgeLuma >= 0)) {
        return 0.0;
    }
    else {
        return 0.5 - distanceToNearestEnd / (distanceToEndP + distanceToEndN);
    }
```

Edge blend factors.

Now adjust `FXAAPassFragment` so we can see the results of edge blending.

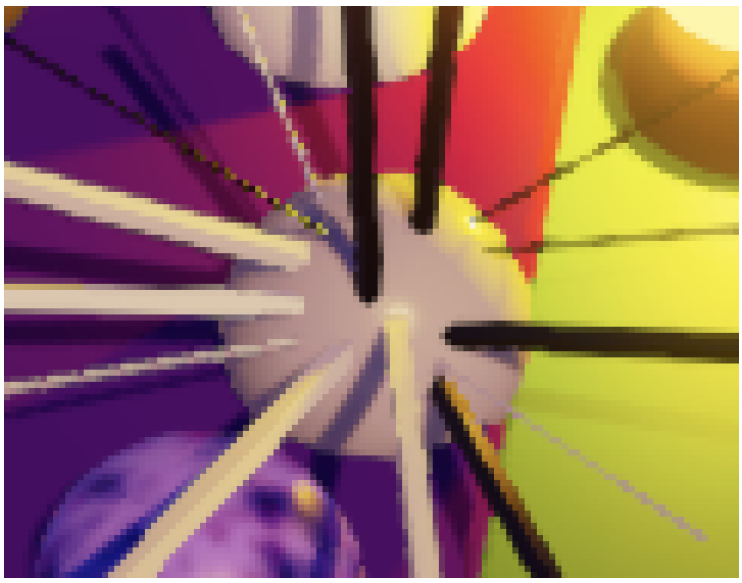
```
	if (CanSkipFXAA(luma)) {
		return GetSource(input.screenUV);
	}

	FXAAEdge edge = GetFXAAEdge(luma);

	float blendFactor = GetEdgeBlendFactor (luma, edge, input.screenUV);
	//return blendFactor;
	float2 blendUV = input.screenUV;
```



Only edge blending and only subpixel blending at full strength.

To apply both edge and subpixel blending we use the largest blend factor of both.

```
	float blendFactor = max(
		GetSubpixelBlendFactor(luma), GetEdgeBlendFactor (luma, edge, input.screenUV)
	);
```

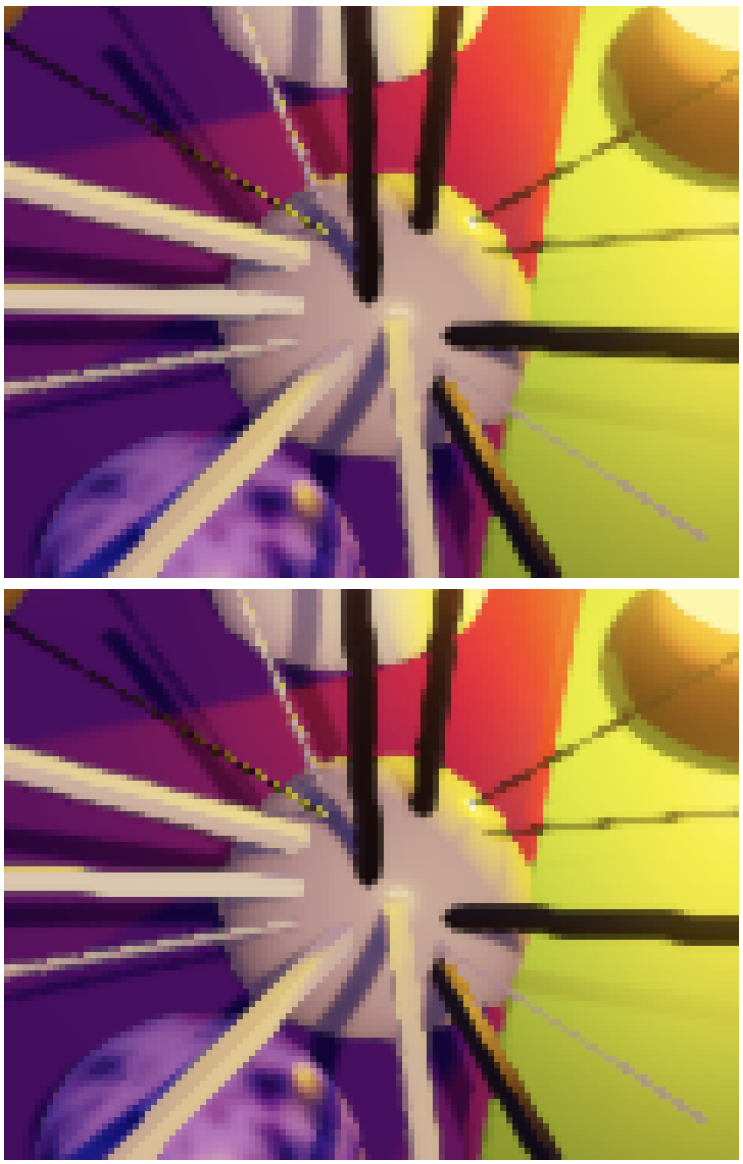


Edge blending together with subpixel blending at 0.75.

3.6 Limited Edge Search

Searching for the ends of edges can take a long time if they're nearly horizontal or vertical. Up to 100 samples in either direction is too much to guarantee acceptable performance. So we have to terminate the search sooner, but this will make it impossible for FXAA to detect longer edges. To clearly illustrate this reduce the search in `GetEdgeBlendFactor` to at most four pixels in either direction, so terminate the loop after three steps.

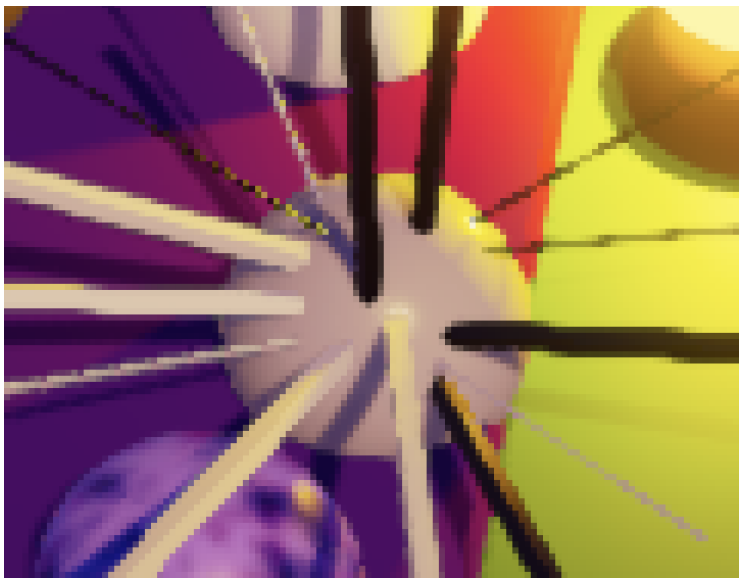
```
	for (int i = 0; i < 3 && !atEndP; i++) { … }

	…

	for (int i = 0; i < 3 && !atEndN; i++) { … }
```



Up to 100 steps and only up to four steps.

The result is that all endpoints further than four pixels away are treated as if they were four pixels away, which reduces the quality of FXAA. If the loop terminated before finding an edge then we underestimate the distance, as the end is at least one step further away. Thus we can improve the results somewhat by adding another step to guess the real distance if we didn't find it. If we didn't find it after four steps then we guess that the real distance is five.

```
	for (int i = 0; i < 3 && !atEndP; i++) { … }
	if (!atEndP) {
		uvP += uvStep;
	}

	…

	for (int i = 0; i < 3 && !atEndN; i++) { … }
	if (!atEndN) {
		uvN -= uvStep;
	}
```



Up to four steps with extra guess.

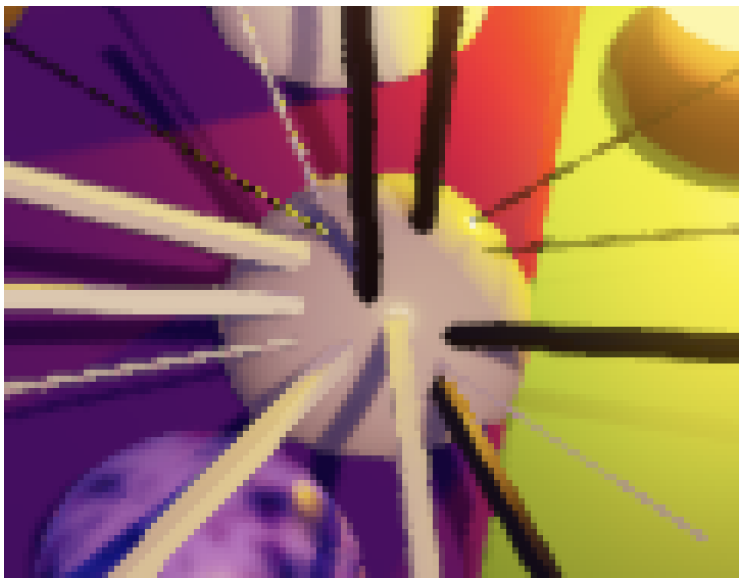
3.7 Edge Quality

How far we allow the edge search to go limits the quality of the result and how long it takes. So it is a tradeoff between quality and performance, which means that there isn't a single best choice. To make our approach configurable we'll introduce define statements for the amount of extra edge steps, a list of the offsets for the extra steps, and the offset for the last edge step guess that we use when we have to stop searching. Use these to create a static constant array for the edge step sizes and then use it all in `GetEdgeBlendFactor`.

```
#define EXTRA_EDGE_STEPS 3
#define EDGE_STEP_SIZES 1.0, 1.0, 1.0
#define LAST_EDGE_STEP_GUESS 1.0

static const float edgeStepSizes[EXTRA_EDGE_STEPS] = { EDGE_STEP_SIZES };

float GetEdgeBlendFactor (LumaNeighborhood luma, FXAAEdge edge, float2 uv) {
    …

    for (int i = 0; i < EXTRA_EDGE_STEPS && !atEndP; i++) {
        uvP += uvStep * edgeStepSizes[i];
        lumaDeltaP = GetLuma(uvP) - edgeLuma;
        atEndP = abs(lumaDeltaP) >= gradientThreshold;
    }
    if (!atEndP) {
        uvP += uvStep * LAST_EDGE_STEP_GUESS;
    }

    …

    for (int i = 0; i < EXTRA_EDGE_STEPS && !atEndN; i++) {
        uvN -= uvStep * edgeStepSizes[i];
        lumaDeltaN = GetLuma(uvN) - edgeLuma;
        atEndN = abs(lumaDeltaN) >= gradientThreshold;
    }
    if (!atEndN) {
        uvN -= uvStep * LAST_EDGE_STEP_GUESS;
    }

    …
}
```

We explicitly create an array for the step sizes so that we can vary them. For example, the original FXAA algorithm contains multiple quality presets that vary both the amount of steps and their sizes. Quality preset 22 is a fast low-quality preset that has three extra steps. The first extra step—after the initial offset of a single pixel—has an offset of 1.5. This extra half-pixel offset means that we end up sampling the average of a square of four pixels along the edge instead of a single pair. The two steps after that have size 2, each again sampling squares of four pixels instead of pairs. Thus it covers a distance of up to seven pixels with only four samples. And if it failed to detect the end it guesses that it is a least eight steps further away.

```
#define EXTRA_EDGE_STEPS 3
#define EDGE_STEP_SIZES 1.5, 2.0, 2.0
#define LAST_EDGE_STEP_GUESS 8.0
```

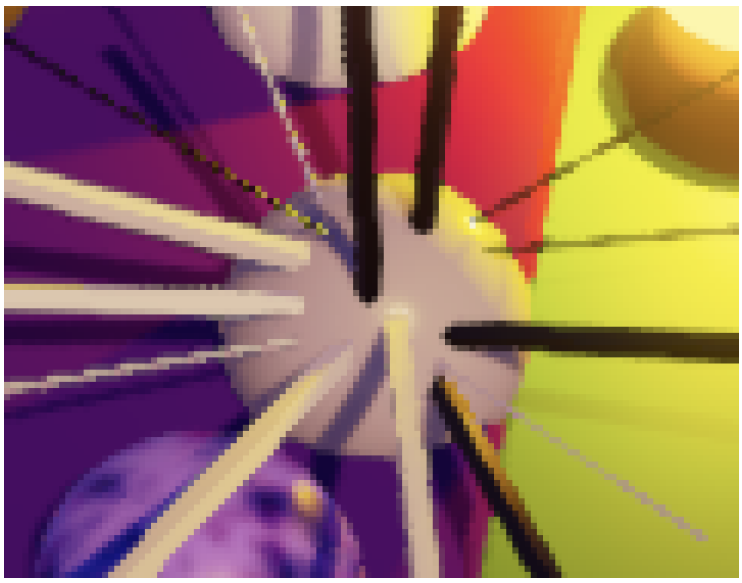


Low quality.

Using these settings we get low-quality results, but they're better able to deal with longer edges than if we fixed the extra step size at 1. The downsize is that the edges can appear to be a bit dithered. This is caused by the larger steps that produce less accurate results.

Let's use the current configuration for low-quality FXAA, only using it if `FXAA_QUALITY_LOW` is defined, which it currently isn't. Otherwise we'll use settings corresponding to quality preset 26. It uses the same approach as preset 22, but with eight extra samples, all with step size 2 except the last one, which has step size 4 so it skips a step to look further ahead.

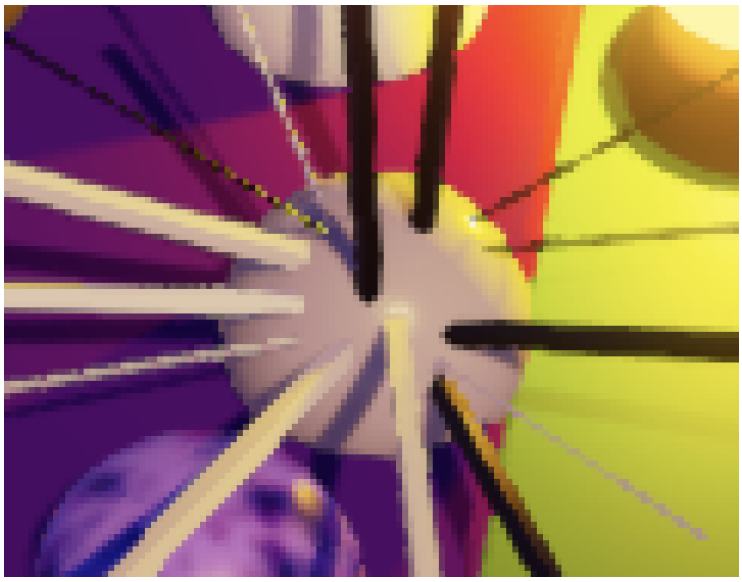
```
#if defined(FXAA_QUALITY_LOW)
	#define EXTRA_EDGE_STEPS 3
	#define EDGE_STEP_SIZES 1.5, 2.0, 2.0
	#define LAST_EDGE_STEP_GUESS 8.0
#else
	#define EXTRA_EDGE_STEPS 8
	#define EDGE_STEP_SIZES 1.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 4.0
	#define LAST_EDGE_STEP_GUESS 8.0
#endif
```



Medium quality.

Let's use this configuration for medium quality—for when `FXAA_QUALITY_MEDIUM` is defined—and add a final default configuration matching preset 39. This is a high-quality configuration that has ten extra steps and only switches to square block sampling after four extra pair samples, again with a skip for the final step and a guess of 8.

```
#if defined(FXAA_QUALITY_LOW)
    #define EXTRA_EDGE_STEPS 3
    #define EDGE_STEP_SIZES 1.5, 2.0, 2.0
    #define LAST_EDGE_STEP_GUESS 8.0
#elif defined(FXAA_QUALITY_MEDIUM)
    #define EXTRA_EDGE_STEPS 8
    #define EDGE_STEP_SIZES 1.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 4.0
    #define LAST_EDGE_STEP_GUESS 8.0
#else
    #define EXTRA_EDGE_STEPS 10
    #define EDGE_STEP_SIZES 1.0, 1.0, 1.0, 1.0, 1.5, 2.0, 2.0, 2.0, 2.0, 4.0
    #define LAST_EDGE_STEP_GUESS 8.0
#endif
```



High quality.

To allow selection of a quality level add a multi-compile directive to both FXAA passes of the *PostFXStack* shader. We only need keywords for the low and medium quality versions, using the default of no keyword for the high quality version.

```
			#pragma fragment FXAAPassFragment
			#pragma multi_compile _ FXAA_QUALITY_MEDIUM FXAA_QUALITY_LOW
```

Add a corresponding quality configuration option to `CameraBufferSettings.FXAA`.

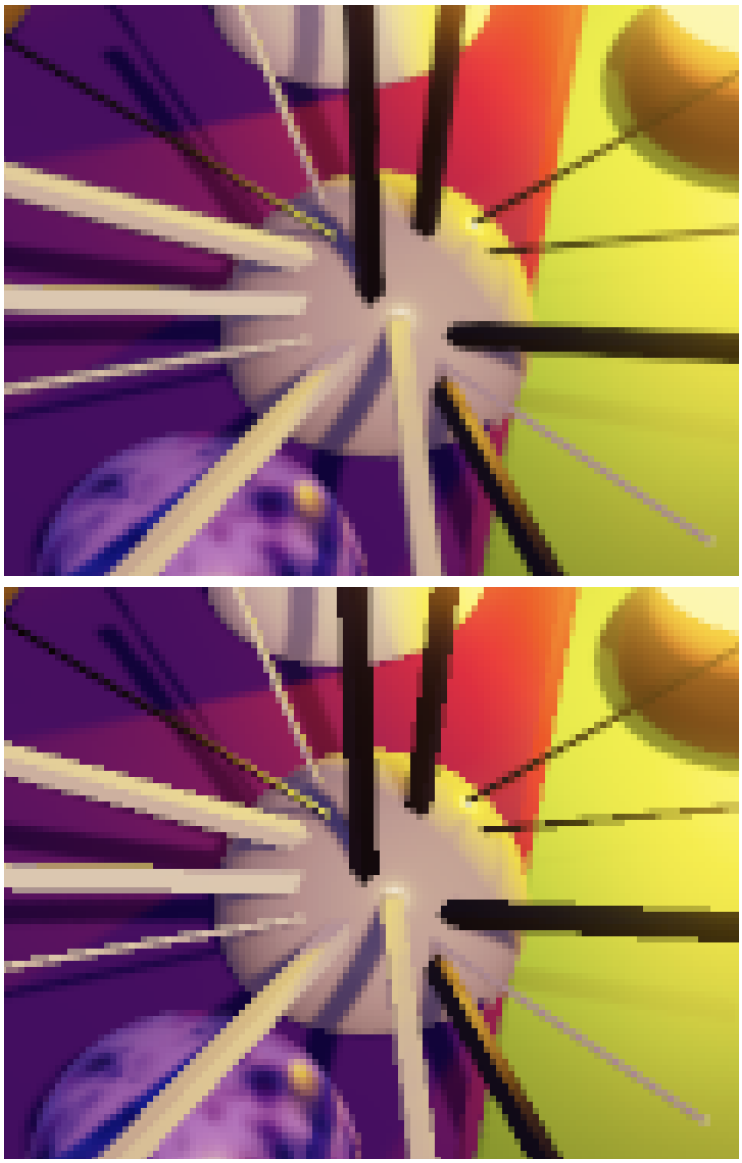
```
		public enum Quality { Low, Medium, High }

		public Quality quality;
```

Then enable or disable the appropriate keywords in `PostFXStack`. Do this in a new `ConfigureFXAA` method and also move the code that sets the configuration vector there. Then invoke it at the appropriate moment in `DoFinal`.

```
	const string
		fxaaQualityLowKeyword = "FXAA_QUALITY_LOW",
		fxaaQualityMediumKeyword = "FXAA_QUALITY_MEDIUM";

	…

	void ConfigureFXAA () {
		if (fxaa.quality == CameraBufferSettings.FXAA.Quality.Low) {
			buffer.EnableShaderKeyword(fxaaQualityLowKeyword);
			buffer.DisableShaderKeyword(fxaaQualityMediumKeyword);
		}
		else if (fxaa.quality == CameraBufferSettings.FXAA.Quality.Medium) {
			buffer.DisableShaderKeyword(fxaaQualityLowKeyword);
			buffer.EnableShaderKeyword(fxaaQualityMediumKeyword);
		}
		else {
			buffer.DisableShaderKeyword(fxaaQualityLowKeyword);
			buffer.DisableShaderKeyword(fxaaQualityMediumKeyword);
		}
		buffer.SetGlobalVector(fxaaConfigId, new Vector4(
			fxaa.fixedThreshold, fxaa.relativeThreshold, fxaa.subpixelBlending
		));
	}

	void DoFinal (int sourceId) {
		…
		if (fxaa.enabled) {
			ConfigureFXAA();
			buffer.GetTemporaryRT(
				colorGradingResultId, bufferSize.x, bufferSize.y, 0,
				FilterMode.Bilinear, RenderTextureFormat.Default
			);
			Draw(
				sourceId, colorGradingResultId,
				keepAlpha ? Pass.ApplyColorGrading : Pass.ApplyColorGradingWithLuma
			);
		}

		…
	}
```

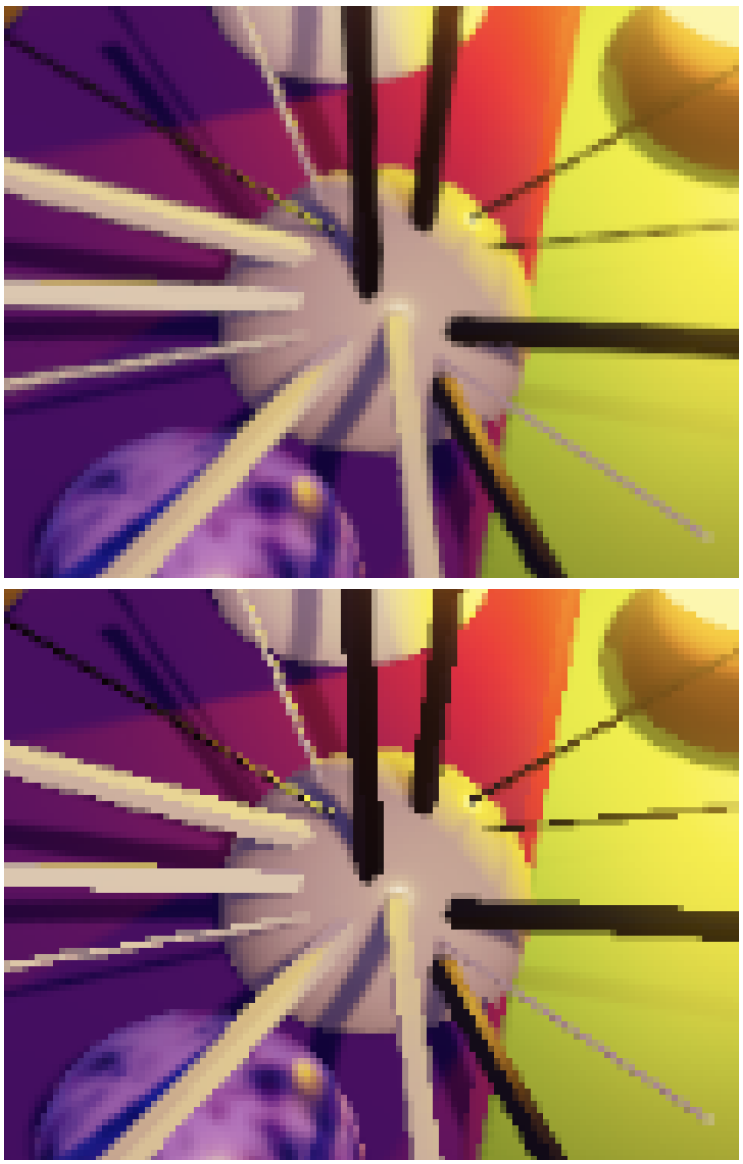
FXAA set to high quality.

These quality presets are just examples, you can configure them as you like. It is also possible to combine FXAA with doubling of the render scale for even better results. Keep in mind that FXAA operates at the adjusted render scale, so this is quite expensive.



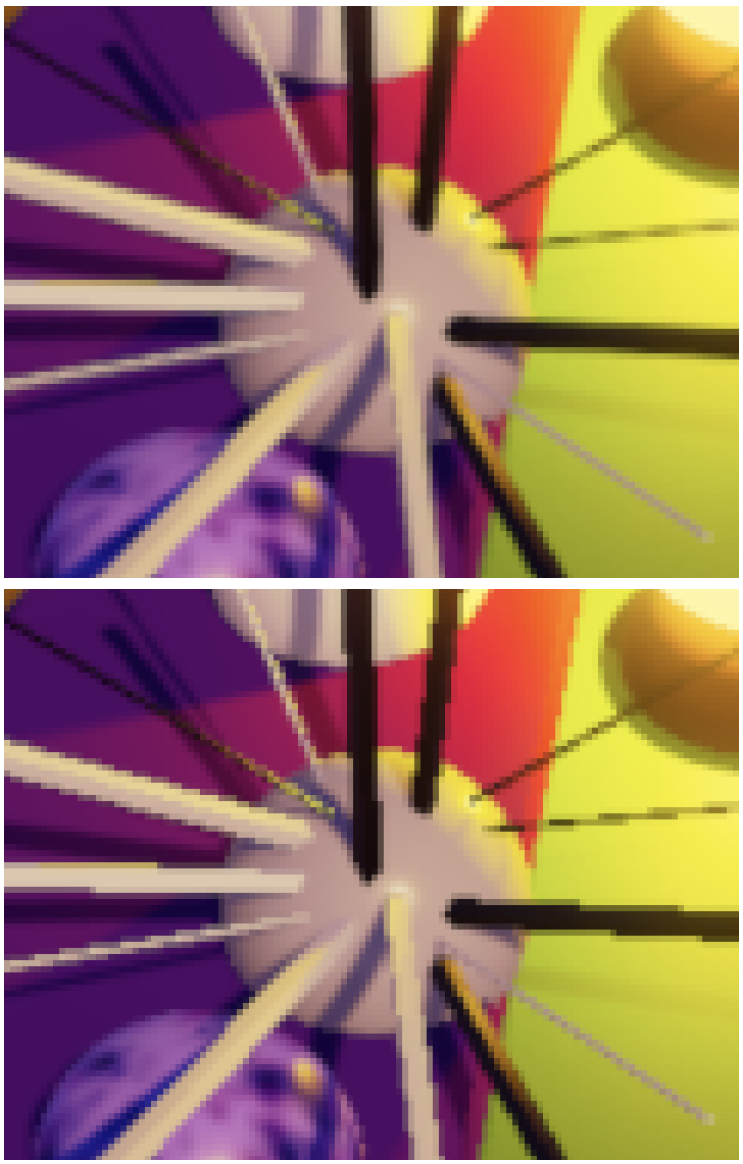
Render scale 2 with and without high-quality FXAA.

You don't need to go all the way. You could for example combine FXAA with render scale 4/3. That would increase the amount of pixels by 1.78 instead of by four. This was suggested by Timothy Lottes in his presentation *Filtering Approaches for Real-Time Anti-Aliasing* for SIGGRAPH2011.



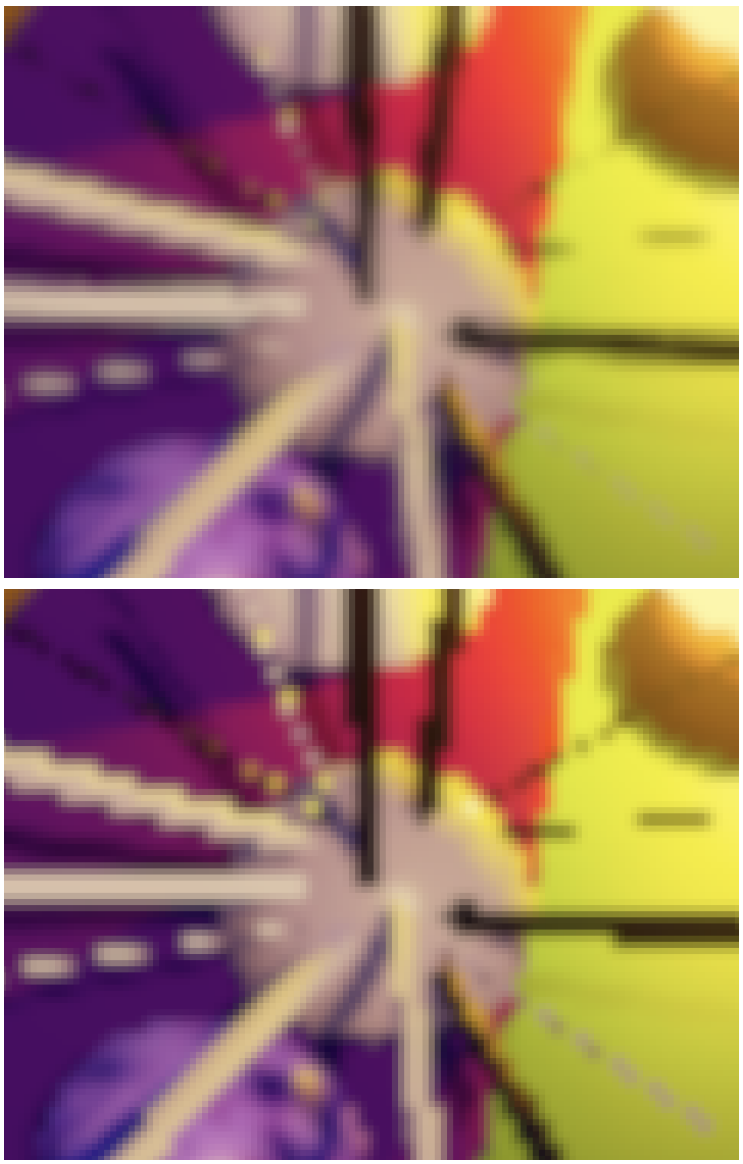
Bilinear render scale 1.333333 with and without high-quality FXAA.

This can be smoothed further by using bicubic rescaling.



Bicubic render scale 1.333333 with and without high-quality FXAA.

It is also possible to use FXAA to improve the results of a lowered render scale.



Bicubic render scale 0.5 with and without high-quality FXAA.

3.8 Unrolling Loops

Because our loops have a guaranteed maximum amount of iterations it is possible to unroll them, which means that we replace them with a sequence of conditional code blocks. We don't have to do this explicitly, we can let the shader compiler do this by placing `UNITY_UNROLL` before the loops. This adds the `unroll` attribute to them.

```
	UNITY_UNROLL
	for (int i = 0; i < EXTRA_EDGE_STEPS && !atEndP; i++) { … }
	…

	UNITY_UNROLL
	for (int i = 0; i < EXTRA_EDGE_STEPS && !atEndN; i++) { … }
```

This turns out to consistently improve performance a tiny bit, not expected to exceed a gain of 1 FPS. While this isn't much, it's for free.

The original FXAA algorithm also combines both loops, searching in both directions in lockstep. Each iteration, only the directions that haven't finished yet advance and sample again. This might be faster in some cases, but in my case two separate loops performed slightly better than a single loop. As always, if you want the absolute best performance, test it yourself, per project, per target platform.

Want to know when the next tutorial gets released? Keep tabs on my Patreon page!

license

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick