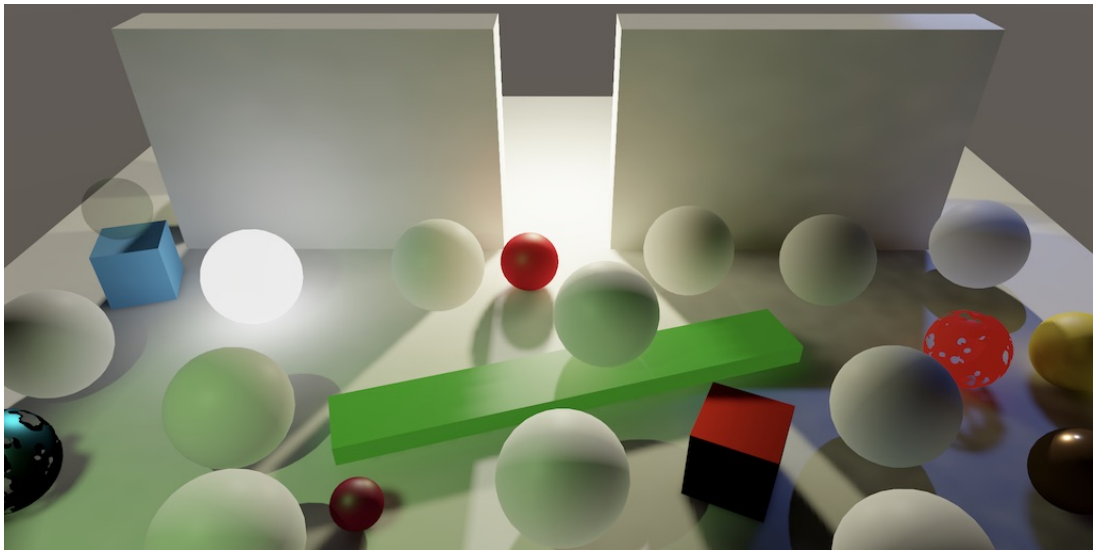
Image Quality Render scale, MSAA, and HDR

Adjust the render scale.
Support MSAA.
Enable HDR, with optional tone mapping.

This is the twelfth installment of a tutorial series covering Unity's scriptable render pipeline. It's about improving image quality, by adjusting the render scale, applying MSAA, and rendering to HDR buffers in combination with tone mapping.

This tutorial is made with Unity 2018.4.6f1.



HDR, MSAA, and render scale working together.

1 Render Scale

The camera determines the width and height of the image that gets rendered, that's out of control of the pipeline. But we can do whatever we want before rendering to the camera's target. We can render to intermediate textures, which we can give any size we like. For example we could render everything to a smaller texture, followed by a final blit to the camera's target to scale it up to the desired size. That reduces the image quality, but speeds up rendering because there are fewer fragment to process. The Lightweight/Universal pipeline has a *Render Scale* option to support this, so let's add it to our own pipeline as well.

1.1 Scaling Down

Add a slider for the render scale to `MyPipelineAsset`, initially with a range of ¼–1. Reducing the resolution to a quarter drops the quality by a lot—pixel count gets divided by 16—and is most likely unacceptable, unless the original resolution is very high.

```
	[SerializeField, Range(0.25f, 1f)]
	float renderScale = 1f;
```

Render scale slider set to its minimum.

Pass the render scale to the pipeline instance.

```
	protected override IRenderPipeline InternalCreatePipeline () {
		…
		return new MyPipeline(
			…
			(int)shadowCascades, shadowCascadeSplit,
			renderScale
		);
	}
```

And have `MyPipeline` keep track of it.

```
	float renderScale;

	public MyPipeline (
		…
		int shadowCascades, Vector3 shadowCascasdeSplit,
		float renderScale
	) {
		…
		this.renderScale = renderScale;
	}
```

When rendering a camera in `Render`, determine whether we're using scaled rendering before we create render textures in case we have an active stack. We use scaled rendering when the render scale has been reduced, but only do so for a game camera, so the scene, preview, and other cameras remain unaffected. Keep track of this decision with a boolean variable so we refer back to it.

```
		var myPipelineCamera = camera.GetComponent<MyPipelineCamera>();
		MyPostProcessingStack activeStack = myPipelineCamera ?
			myPipelineCamera.PostProcessingStack : defaultStack;

		bool scaledRendering =
			renderScale < 1f && camera.cameraType == CameraType.Game;

		if (activeStack) {
			…
		}
```

Keep track of the render width and height in variables as well. They're determined by the camera by default, but must be adjusted when using scaled rendering.

```
		bool scaledRendering =
			renderScale < 1f && camera.cameraType == CameraType.Game;

		int renderWidth = camera.pixelWidth;
		int renderHeight = camera.pixelHeight;
		if (scaledRendering) {
			renderWidth = (int)(renderWidth * renderScale);
			renderHeight = (int)(renderHeight * renderScale);
		}
```

1.2 Rendering to a Scaled Texture

We must now render to an intermediate texture when either scaled rendering or post-processing is used. Keep track of this with a boolean as well and use the adjusted width and height when getting the textures.

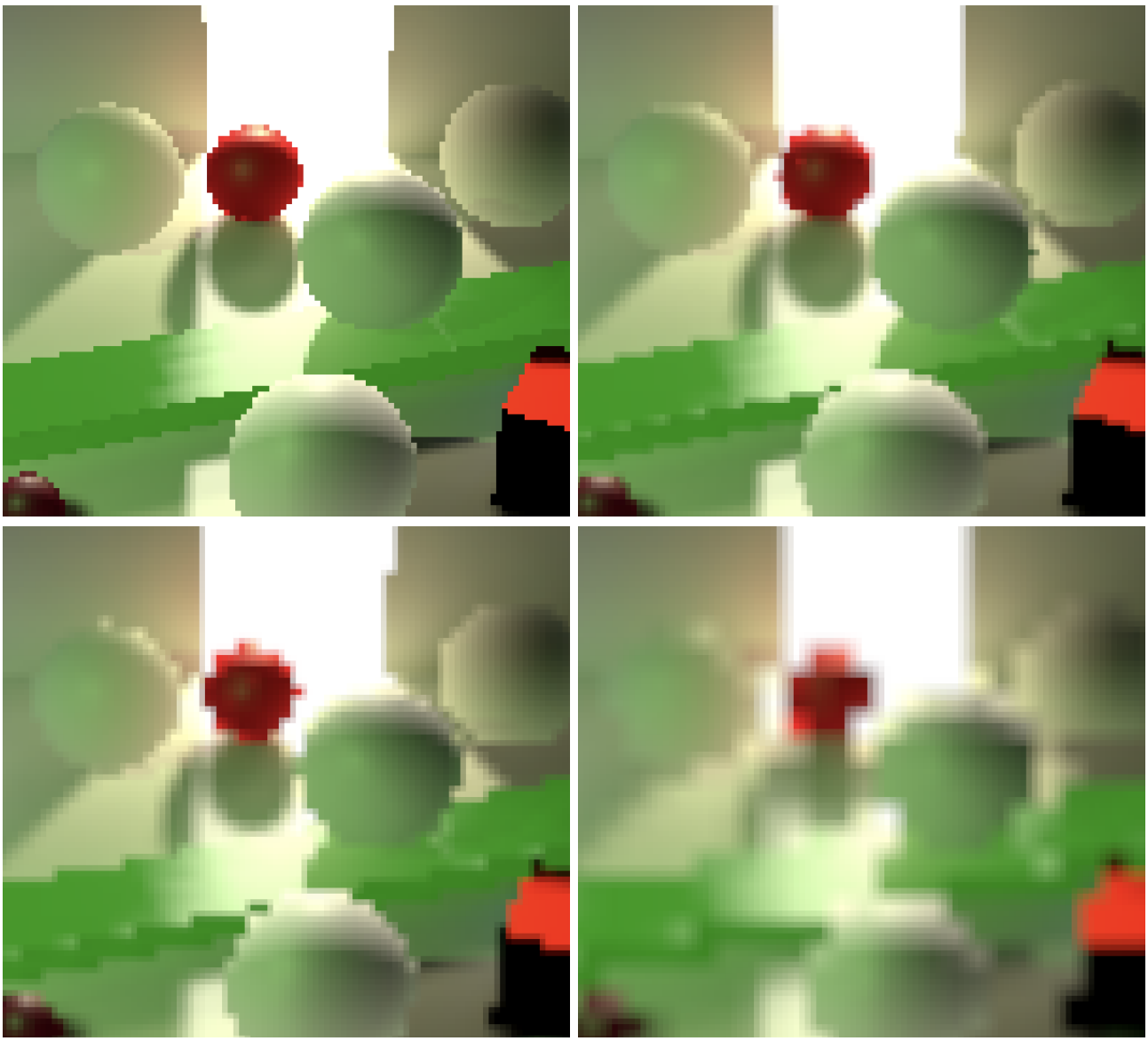
```
		bool renderToTexture = scaledRendering || activeStack;

		if (renderToTexture) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, renderWidth, renderHeight, 0,
				FilterMode.Bilinear
			);
			cameraBuffer.GetTemporaryRT(
				cameraDepthTextureId, renderWidth, renderHeight, 24,
				FilterMode.Point, RenderTextureFormat.Depth
			);
			…
		}
```

From now on the adjusted width and height must be passed to the active stack, when `RenderAfterOpaque` gets invoked.

```
		context.DrawSkybox(camera);

		if (activeStack) {
			activeStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				renderWidth, renderHeight
			);
			…
		}
```

The same is true for `RenderAfterTransparent`. Now we must always release the textures when we're rendering to them but only invoke `RenderAfterTransparent` when a stack is in use. If not we can use a regular blit to copy the scaled texture to the camera's target.

```
		DrawDefaultPipeline(context, camera);

		if (renderToTexture) {
			if (activeStack) {
				activeStack.RenderAfterTransparent(
					postProcessingBuffer, cameraColorTextureId,
					cameraDepthTextureId, renderWidth, renderHeight
				);
				context.ExecuteCommandBuffer(postProcessingBuffer);
				postProcessingBuffer.Clear();
			}
			else {
				cameraBuffer.Blit(
					cameraColorTextureId, BuiltinRenderTextureType.CameraTarget
				);
			}
			cameraBuffer.ReleaseTemporaryRT(cameraColorTextureId);
			cameraBuffer.ReleaseTemporaryRT(cameraDepthTextureId);
		}
```

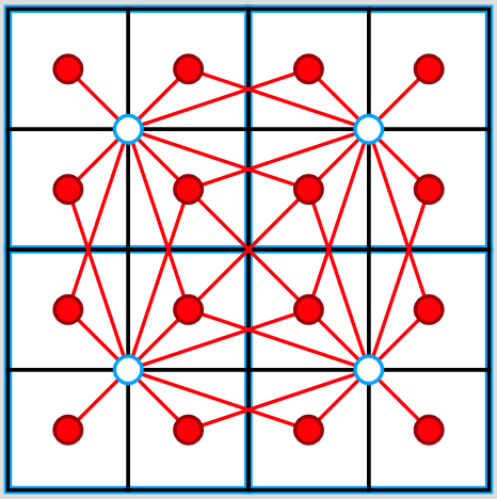


Render scale 1, 0.75, 0.5, and 0.25; zoomed in and without post-processing.

Adjusting the render scale affects everything that our pipeline renders, except shadows as they have their own size. A slight reduction of the render scale seems to apply a bit of anti-aliasing, although haphazardly. But further reduction makes it clear that this is just a loss of detail that gets smudged due to bilinear interpolation when blitting to the final render target.

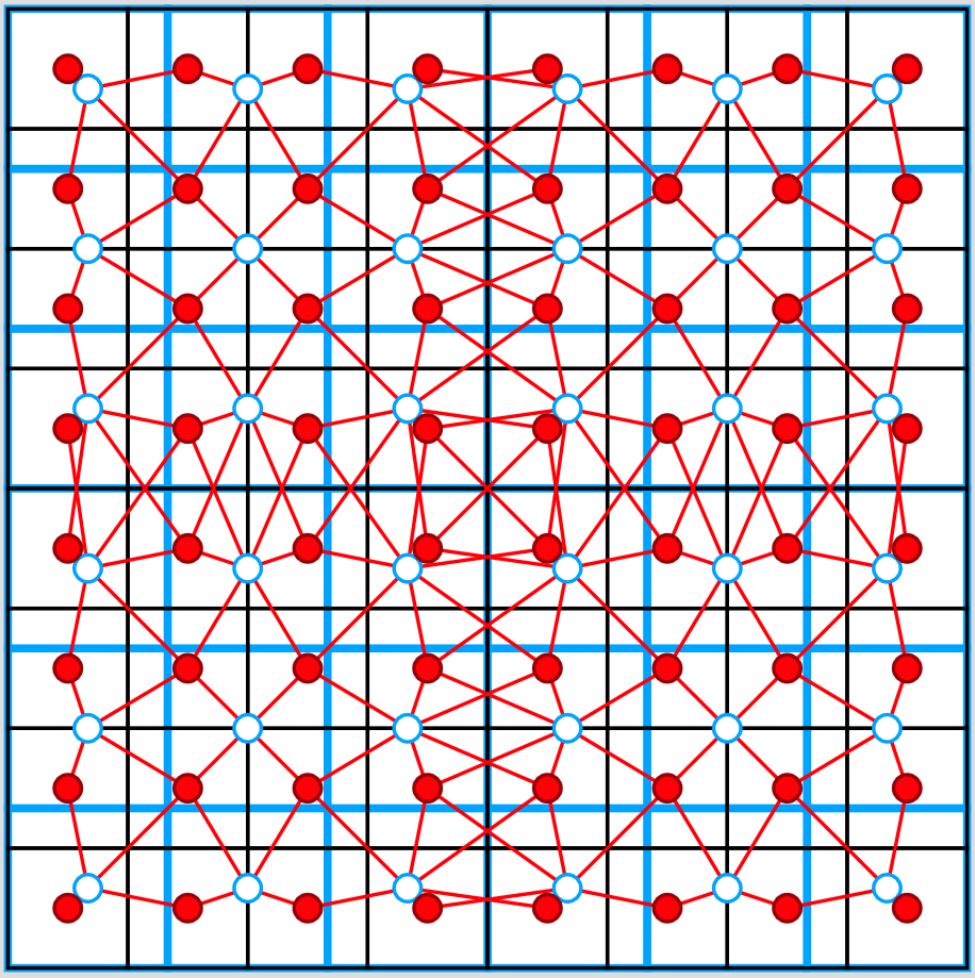
How does render scale interact with bilinear interpolation?

A render scale of 0.5 is most straightforward: we end up with a single pixel per block of 2×2 target pixels. Each final pixel uses the same four weights for interpolation, but there are four possible orientations.



Render scale 0.5 filtering.

Other render scales produce pixels with varying weight configurations, because the distance from source to target pixel varies, in a regular pattern depending on the scale.



Render scale 0.75 filtering.

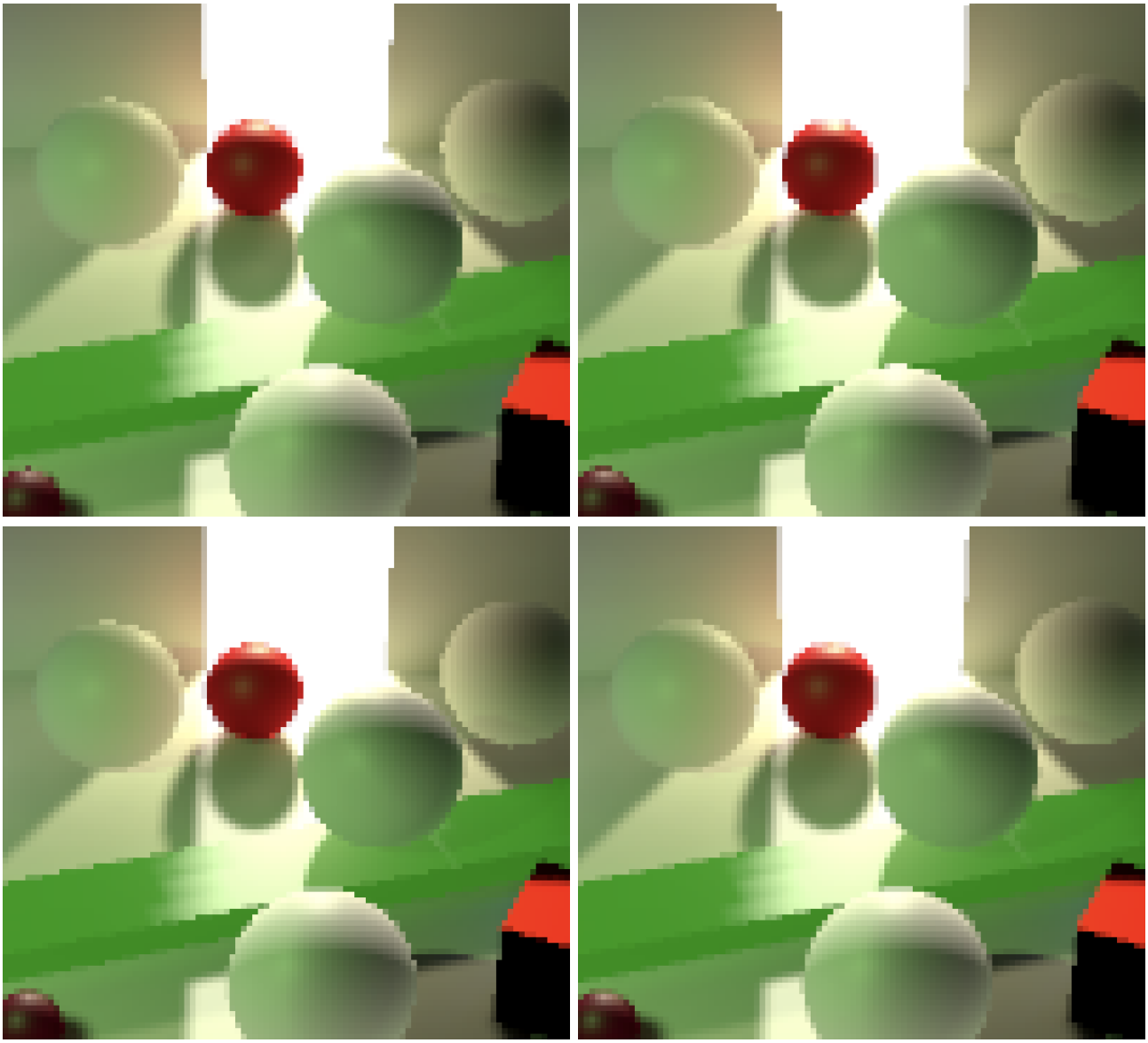
1.3 Scaling Up

We can scale down to improve performance at the cost of image quality. We can do the opposite as well: scale up to improve image quality at the cost of performance. To make this possible increase the maximum render scale to 2 in `MyPipelineAsset`.

```
	[SerializeField, Range(0.25f, 2f)]
	float renderScale = 1f;
```

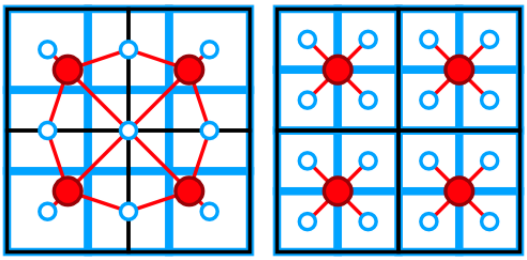
And also activate scaled rendering in `MyPipeline.Render` if the render scale is greater than 1.

```
		bool scaledRendering =
			(renderScale < 1f || renderScale > 1f) &&
			camera.cameraType == CameraType.Game;
```



Render scale 1.25, 1.5, 1.75, and 2.

The image quality indeed improves, but is only really good when the scale is set to 2. At this scale we end up averaging a dedicated 2×2 pixel blocks for each final pixel. This means that we're rendering four times as many pixels, which is the same as supersampling anti-aliasing, SSAA 2× using a regular grid.



Render scale 1.5 and 2 filtering.

Increasing the render scale further won't improve image quality. At 3 we end up with the same result as render scale 1, while at 4 we're back at 2×2 blocks per pixel but closer together. That's because a single bilinear blit can only average four pixels. Taking advantage of higher scales would require a pass that performs more than one texture sample per fragment. While that's possible it's impractical because the required work scales quadratically with the render scale. SSAA 4× would require use to render sixteen times as many pixels.

2 MSAA

An alternative to SSAA is MSAA: multi-sample anti-aliasing. The idea is the same, but the execution differs. MSAA keeps track of multiple samples per pixel, which don't have to placed in a regular grid. The big difference is that the fragment program is only invoked once per primitive per fragment, so at the original resolution. The result is then copied to all subsamples that are covered by the rasterized triangle. This significantly reduces the amount of work that has to be done, but it means that MSAA only affects triangle edges and nothing else. High-frequency surface patterns and alpha-clipped edges remain aliased.

> **What about alpha-to-coverage?**
>
> That's a trick to smooth alpha-clipped edges somewhat, which can produce decent results in some cases. It won't be covered in this tutorial.

2.1 Configuration

Add an option to select the MSAA mode to `MyPipelineAsset`. By default MSAA is off, the other options being 2×, 4×, and 8×, which can be represented with an enum. The enum values represent the amount of samples per pixel, so the default is 1.

```
	public enum MSAAMode {
		Off = 1,
		_2x = 2,
		_4x = 4,
		_8x = 8
	}

	…

	[SerializeField]
	MSAAMode MSAA = MSAAMode.Off;
```

MSAA mode.

> **Why not support MSAA 16×?**
>
> You can do that, but it is very expensive for little extra quality gain compared to 8×, and doesn't have widespread support.

Pass the amount of samples per pixel to the pipeline instance.

```
		return new MyPipeline(
			…
			renderScale, (int)MSAA
		);
```

And keep track of it in `MyPipeline`.

```
	int msaaSamples;

	public MyPipeline (
		…
		float renderScale, int msaaSamples
	) {
		…
		this.msaaSamples = msaaSamples;
	}
```

Not all platforms support MSAA and the maximum sample count also varies. Going above the maximum could result in a crash, so we have to make sure that we remain within the limit. We can do that by assigning the sample count to `QualitySettings.antiAliasing`. Our pipeline doesn't use this quality setting, but it takes care of enforcing the limit when assigned to it. So after assigning to it we copy it back to our own sample count. The only thing we have to be aware of is that it will yield zero when MSAA is unsupported, which we have to convert to a sample count of 1.

```
		QualitySettings.antiAliasing = msaaSamples;
		this.msaaSamples = Mathf.Max(QualitySettings.antiAliasing, 1);
```

2.2 Multisampled Render Textures

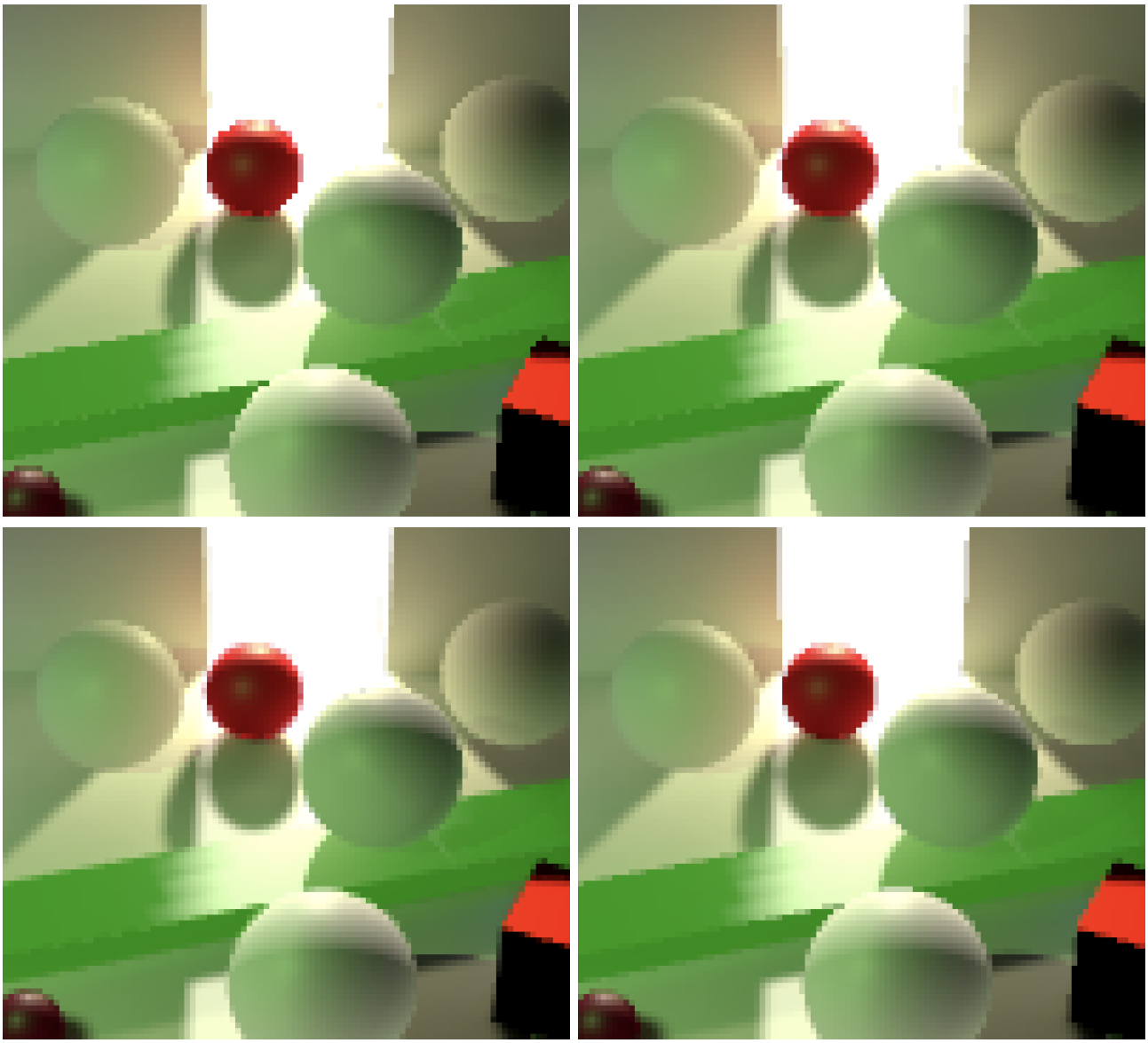
MSAA support is set per camera, so keep track of the samples used for rendering in `Render` and force it to 1 if the camera doesn't have MSAA enabled. Then if we end up with more than one sample per pixel we have to render to intermediate multi-sampled textures, MS textures for short.

```
		int renderSamples = camera.allowMSAA ? msaaSamples : 1;
		bool renderToTexture =
			scaledRendering || renderSamples > 1 || activeStack;
```

To configure the render textures correctly we have to add two more arguments to `GetTemporaryRT`. First the read-write mode, which is the default for the color buffer and is linear for the depth buffer. The next argument is the sample count.

```
		if (renderToTexture) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, renderWidth, renderHeight, 0,
				FilterMode.Bilinear, RenderTextureFormat.Default,
				RenderTextureReadWrite.Default, renderSamples
			);
			cameraBuffer.GetTemporaryRT(
				cameraDepthTextureId, renderWidth, renderHeight, 24,
				FilterMode.Point, RenderTextureFormat.Depth,
				RenderTextureReadWrite.Linear, renderSamples
			);
			…
		}
```

Try this out with all post-processing disabled.

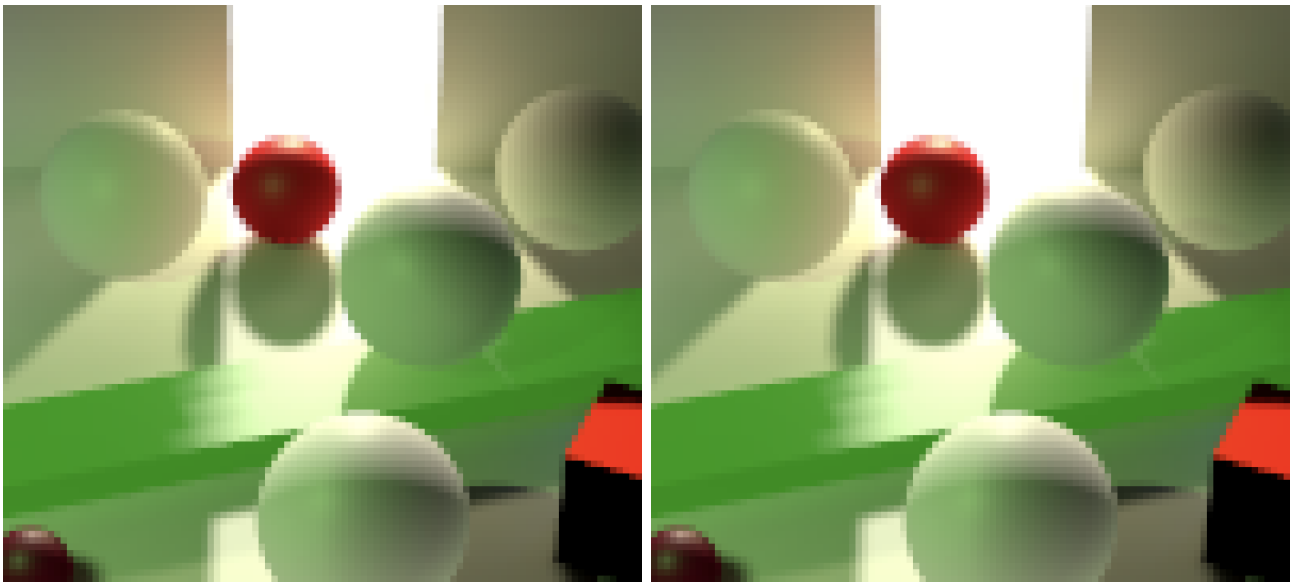


MSAA 2×, 4×, 8×, plus no MSAA with render scale 2 for comparison.

Does MSAA work with directional shadows?

It works fine for our render pipeline. Unity's pipelines have trouble because they use a screen-space pass for cascaded directional shadows. We'll encounter the same kind of problem a bit further in this tutorial.

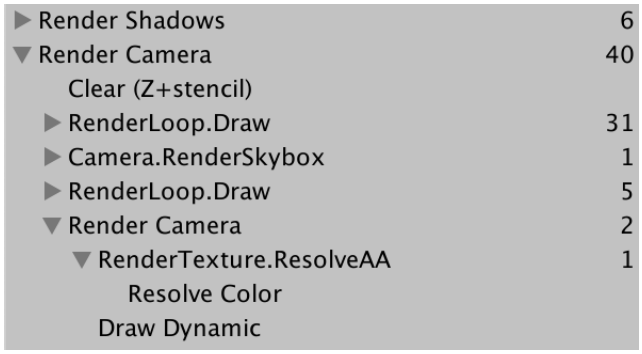
Compared to doubling the render scale, MSAA 4× ends up slightly better than render scale 2, with the caveat that the render scale affects everything and not just geometry edges. You could also combine both approaches. For example MSAA 4× at render scale 2 is roughly comparable to solely MSAA 8× at render scale 1, although it uses sixteen samples instead of eight per final pixel.



MSAA 4× and 8× both at render scale 2.

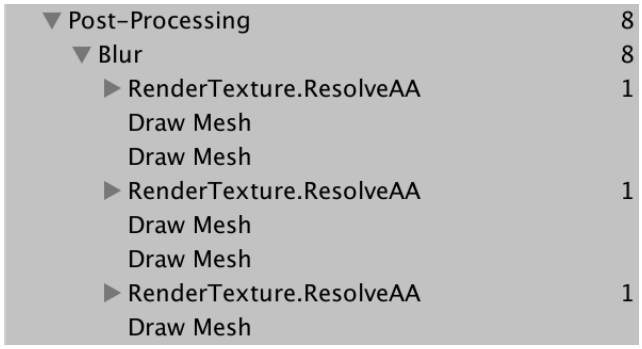
2.3 Resolving MS Textures

While we can render directly to MS textures, we cannot directly read from them the normal way. If we want to sample a pixel it must first be resolved, which means averaging all samples to arrive at the final value. The resolve happens for the entire texture at once in a special *Resolve Color* pass, which gets inserted automatically before a pass that samples it.



Resolving color before final blit.

Resolving the MS texture creates a temporary regular texture which remains valid until something new gets rendered to the MS texture. So if we sample from and then render to the MS texture multiple times we end up with extra resolve passes for the same texture. You can see this when activating a post-effect stack with blurring enabled. At strength 5 we get three resolve passes.



Resolving three times with blur strength 5.

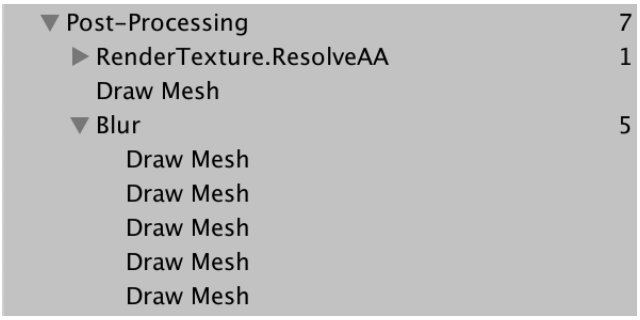
The additional resolve passes are useless, because our full-screen effects don't benefit from MSAA. To avoid needlessly rendering to an MS texture we can blit to an intermediate texture once and then use that instead of the camera target. To make this possible add a samples parameter to the `RenderAfterOpaque` and `RenderAfterTransparent` methods in **`MyPostProcessingStack`**. If blurring is enabled and MSAA is used then copy to a resolved texture and pass that to `Blur`.

```
	static int resolvedTexId =
		Shader.PropertyToID("_MyPostProcessingStackResolvedTex");

	…

	public void RenderAfterOpaque (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height, int samples
	) { … }

	public void RenderAfterTransparent (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height, int samples
	) {
		if (blurStrength > 0) {
			if (samples > 1) {
				cb.GetTemporaryRT(
					resolvedTexId, width, height, 0, FilterMode.Bilinear
				);
				Blit(cb, cameraColorId, resolvedTexId);
				Blur(cb, resolvedTexId, width, height);
				cb.ReleaseTemporaryRT(resolvedTexId);
			}
			else {
				Blur(cb, cameraColorId, width, height);
			}
		}
		else {
			Blit(cb, cameraColorId, BuiltinRenderTextureType.CameraTarget);
		}
	}
```

Add the render samples as arguments in `MyPipeline.Render`.

```
			activeStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				renderWidth, renderHeight, renderSamples
			);

		…

					activeStack.RenderAfterTransparent(
					postProcessingBuffer, cameraColorTextureId,
					cameraDepthTextureId, renderWidth, renderHeight,
					renderSamples
				);
```

The three resolve passes are now reduced to one, plus a simple blit.



Resolving once with blur strength 5.

2.4 No Depth Resolve

Color samples are resolved by averaging them, but this doesn't work for the depth buffer. Averaging adjacent depth values makes no sense and there is no universal approach that can be used, so multisampled depth doesn't get resolved at all. As as result the depth stripes effect doesn't work when MSAA is enabled.

The naive approach to get the effect working again is to simply not apply MSAA to the depth texture when depth stripes are enabled. First add a getter property to `MyPostProcessingStack` that indicates whether it needs to read from a depth texture. This is only required when the depth stripes effect is used.

```
	public bool NeedsDepth {
		get {
			return depthStripes;
		}
	}
```

Now we can keep track of whether we need an accessible depth texture in `MyPipeline.Render`. Only when we need depth do we have to get a separate depth texture, otherwise we can make do with setting the depth bits of the color texture. And if we do need a depth texture then let's explicitly always set its samples to 1 to disable MSAA for it.

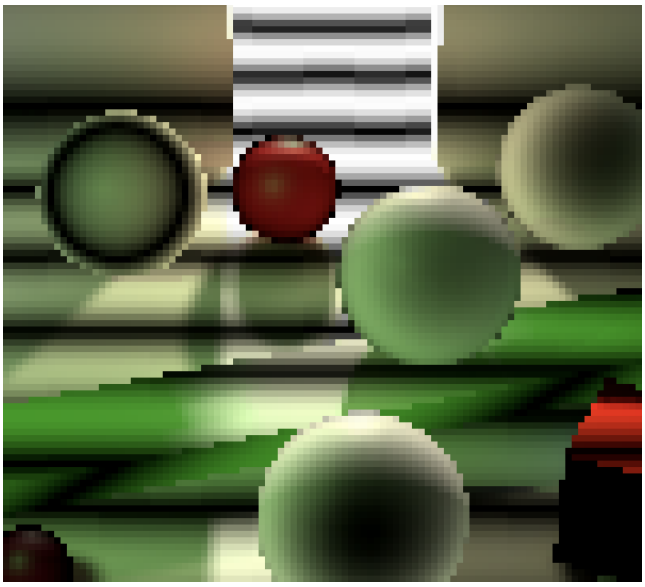
```
		bool needsDepth = activeStack && activeStack.NeedsDepth;

		if (renderToTexture) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, renderWidth, renderHeight,
				needsDepth ? 0 : 24,
				FilterMode.Bilinear, RenderTextureFormat.Default,
				RenderTextureReadWrite.Default, renderSamples
			);
			if (needsDepth) {
				cameraBuffer.GetTemporaryRT(
					cameraDepthTextureId, renderWidth, renderHeight, 24,
					FilterMode.Point, RenderTextureFormat.Depth,
					RenderTextureReadWrite.Linear, 1
				);
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store,
					cameraDepthTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
				);
			}
			else {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
				);
			}
		}
```

This also affects setting the render target after drawing opaque effects.

```
		context.DrawSkybox(camera);

		if (activeStack) {
			…

			if (needsDepth) {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.Load, RenderBufferStoreAction.Store,
					cameraDepthTextureId,
					RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
				);
			}
			else {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
				);
			}
			context.ExecuteCommandBuffer(cameraBuffer);
			cameraBuffer.Clear();
		}
```

And which textures need to get released at the end.

```
		DrawDefaultPipeline(context, camera);

		if (renderToTexture) {
			…
			cameraBuffer.ReleaseTemporaryRT(cameraColorTextureId);
			if (needsDepth) {
				cameraBuffer.ReleaseTemporaryRT(cameraDepthTextureId);
			}
		}
```



Depth stripes with MSAA 8×.

Depth stripes now show up when MSAA is enabled, but anti-aliasing appears to be broken. This happened because depth information is no longer affected by MSAA. We need a different approach.

2.5 Depth-Only Pass

We need an MS depth texture for regular rendering and a non-MS depth texture for the depth stripes effect. We could solve this problem by creating a custom resolve pass for the depth texture, but unfortunately support for that is very limited. The alternative is to render to depth twice, by adding a depth-only pass that renders to a regular depth texture. This is expensive but feasible. It is what Unity does when a depth buffer is used in combination with MSAA, for example when a screen-space shadow pass is needed for cascaded directional shadows.

Does this mean that Unity's depth-only pass doesn't affect regular rendering?

Indeed. It is not used to prime the depth buffer and we won't use it for that either.

First distinguish between whether we can directly use the depth texture or whether we need a depth-only pass because MSAA is active. The logic that we have right now for getting textures and setting render targets applies to when MSAA is not used.

```
		bool needsDepth = activeStack && activeStack.NeedsDepth;
		bool needsDirectDepth = needsDepth && renderSamples == 1;
		bool needsDepthOnlyPass = needsDepth && renderSamples > 1;

		if (renderToTexture) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, renderWidth, renderHeight,
				needsDirectDepth ? 0 : 24,
				FilterMode.Bilinear, RenderTextureFormat.Default,
				RenderTextureReadWrite.Default, renderSamples
			);
			if (needsDepth) {
				cameraBuffer.GetTemporaryRT(
					cameraDepthTextureId, renderWidth, renderHeight, 24,
					FilterMode.Point, RenderTextureFormat.Depth,
					RenderTextureReadWrite.Linear, 1
				);
			}
			if (needsDirectDepth) {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store,
					cameraDepthTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
				);
			}
			else {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
				);
			}
		}

		…
		context.DrawSkybox(camera);

		if (activeStack) {
			…

			if (needsDirectDepth) {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.Load, RenderBufferStoreAction.Store,
					cameraDepthTextureId,
					RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
				);
			}
			else {
				cameraBuffer.SetRenderTarget(
					cameraColorTextureId,
					RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
				);
			}
			context.ExecuteCommandBuffer(cameraBuffer);
			cameraBuffer.Clear();
		}
```

If needed, add a depth-only pass before invoking `RenderAfterOpaque`. This works like the opaque pass except that we use *DepthOnly* for the pass name, don't need a renderer configuration, and have to set and clear the depth texture as the render target.

```
		context.DrawSkybox(camera);

		if (activeStack) {
			if (needsDepthOnlyPass) {
				var depthOnlyDrawSettings = new DrawRendererSettings(
					camera, new ShaderPassName("DepthOnly")
				) {
					flags = drawFlags
				};
				depthOnlyDrawSettings.sorting.flags = SortFlags.CommonOpaque;
				cameraBuffer.SetRenderTarget(
					cameraDepthTextureId,
					RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
				);
				cameraBuffer.ClearRenderTarget(true, false, Color.clear);
				context.ExecuteCommandBuffer(cameraBuffer);
				cameraBuffer.Clear();
				context.DrawRenderers(
					cull.visibleRenderers, ref depthOnlyDrawSettings, filterSettings
				);
			}

			activeStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				renderWidth, renderHeight
			);
			…
		}
```

Add the required pass to the *Lit* shader. It's a copy of the default pass with all features except instancing, clipping, and LOD fading removed. It won't write color information, so give it a color mask set to zero. It always writes depth and relies on dedicated vertex and fragment functions from a separate *DepthOnly* HLSL file.

```
        Pass {
            Tags {
                "LightMode" = "DepthOnly"
            }

            ColorMask 0
            Cull [_Cull]
            ZWrite On

            HLSLPROGRAM

            #pragma target 3.5

            #pragma multi_compile_instancing
            //#pragma instancing_options assumeuniformscaling

            #pragma shader_feature _CLIPPING_ON
            #pragma multi_compile _ LOD_FADE_CROSSFADE

            #pragma vertex DepthOnlyPassVertex
            #pragma fragment DepthOnlyPassFragment

            #include "../ShaderLibrary/DepthOnly.hlsl"

            ENDHLSL
        }
```

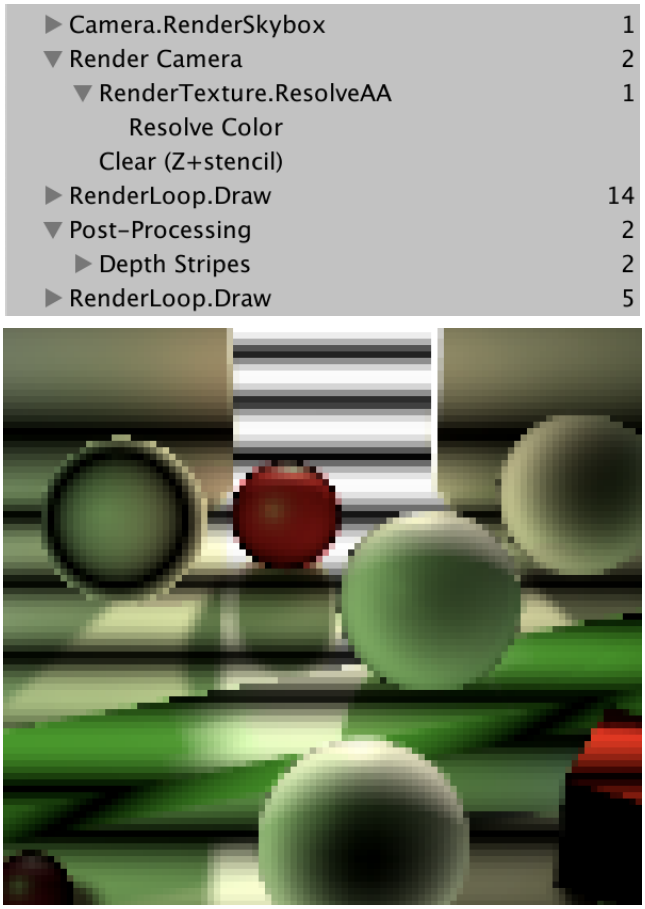
DepthOnly.hlsl is a copy of *Lit.hlsl* with all data removed that doesn't affect depth.

```
#ifndef MYRP_DEPTH_ONLY_INCLUDED
#define MYRP_DEPTH_ONLY_INCLUDED

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"

CBUFFER_START(UnityPerFrame)
	…
CBUFFER_END

CBUFFER_START(UnityPerCamera)
	…
CBUFFER_END

CBUFFER_START(UnityPerDraw)
	…
CBUFFER_END

CBUFFER_START(UnityPerMaterial)
	…
CBUFFER_END

TEXTURE2D(_MainTex);
SAMPLER(sampler_MainTex);

TEXTURE2D(_DitherTexture);
SAMPLER(sampler_DitherTexture);

#define UNITY_MATRIX_M unity_ObjectToWorld

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/UnityInstancing.hlsl"

UNITY_INSTANCING_BUFFER_START(PerInstance)
	UNITY_DEFINE_INSTANCED_PROP(float4, _Color)
UNITY_INSTANCING_BUFFER_END(PerInstance)
```

We only care about the position and UV coordinates. The fragment function only applies LOD and performs clipping. Its final result is simply zero.

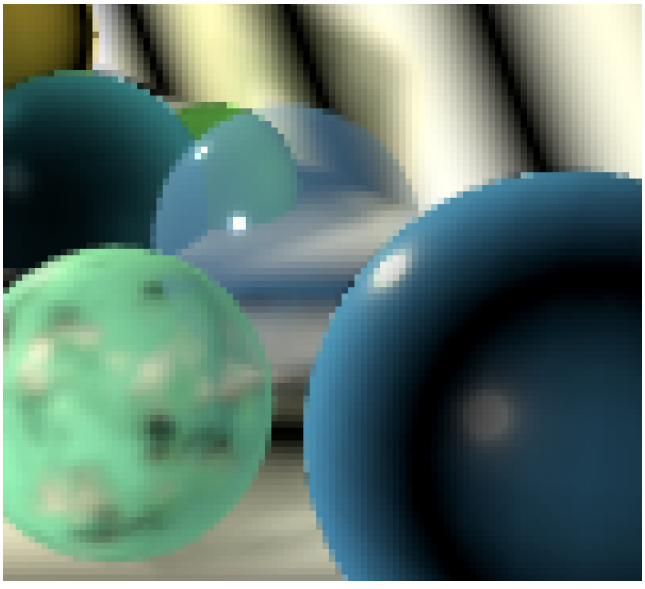
```
struct VertexInput {
	float4 pos : POSITION;
	float2 uv : TEXCOORD0;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
	float2 uv : TEXCOORD3;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

VertexOutput DepthOnlyPassVertex (VertexInput input) {
	VertexOutput output;
	UNITY_SETUP_INSTANCE_ID(input);
	UNITY_TRANSFER_INSTANCE_ID(input, output);
	float4 worldPos = mul(UNITY_MATRIX_M, float4(input.pos.xyz, 1.0));
	output.clipPos = mul(unity_MatrixVP, worldPos);
	output.uv = TRANSFORM_TEX(input.uv, _MainTex);
	return output;
}

void LODCrossFadeClip (float4 clipPos) {
	…
}

float4 DepthOnlyPassFragment (VertexOutput input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);

	#if defined(LOD_FADE_CROSSFADE)
		LODCrossFadeClip(input.clipPos);
	#endif

	float4 albedoAlpha = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv);
	albedoAlpha *= UNITY_ACCESS_INSTANCED_PROP(PerInstance, _Color);

	#if defined(_CLIPPING_ON)
		clip(albedoAlpha.a - _Cutoff);
	#endif

	return 0;
}

#endif // MYRP_DEPTH_ONLY_INCLUDED
```

Now we get a depth-only pass before the depth stripes and MSAA works again. Unfortunately the depth stripes effect itself doesn't benefit from MSAA, so still introduces aliasing where it strongly affects the image. Unity's cascaded directional shadows interfere with MSAA in the same way.



Depth stripes with functional MSAA 8×.

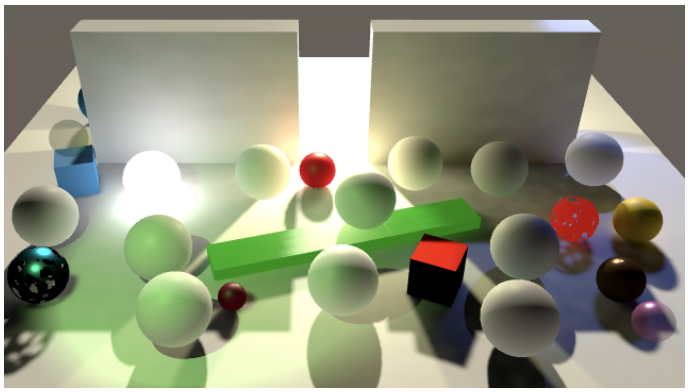
But transparent geometry that gets rendered later still benefits from MSAA.



Transparent with MSAA and depth stripes.

3 HDR

The final topic that we will cover is high-dynamic-range rendering. Up to this point each color channel of a texture has a range of 0–1, so it can represent light levels up to intensity 1. But the intensity of incoming light doesn't have an inherent upper bound. The sun is an example of an extremely bright light source, which is why you shouldn't look at it directly. Its intensity is far greater than we can perceive before our eyes get damaged. But many regular light sources also produce light with an intensity that can exceed the limits of the observer, especially when observed up close. For example, I increased the intensity of the point light in the scene to 100 and also boosted the intensity of the white sphere's emission by one step.



High-intensity light and emission.

The result is that over-bright pixels get blown out to uniform white and there is some color shifting near the edges. The point of HDR rendering is to prevent very bright parts of the image from degrading to uniform white. This will require both storing the bright data and converting it to visible colors.

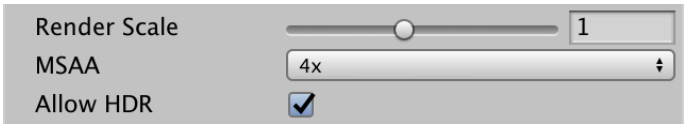
3.1 Configuration

Let's make it optional whether our pipeline supports high-dynamic-range rendering by adding an *Allow HDR* toggle to `MyPipelineAsset`, which we pass to the pipeline instance.

```
	[SerializeField]
	bool allowHDR;

	…

	protected override IRenderPipeline InternalCreatePipeline () {
		Vector3 shadowCascadeSplit = shadowCascades == ShadowCascades.Four ?
			fourCascadesSplit : new Vector3(twoCascadesSplit, 0f);
		return new MyPipeline(
			…
			renderScale, (int)MSAA, allowHDR
		);
	}
```

`MyPipeline` just needs to keep track of it.

```
	bool allowHDR;

	public MyPipeline (
		…
		float renderScale, int msaaSamples, bool allowHDR
	) {
		…
		this.allowHDR = allowHDR;
	}
```



HDR allowed.

3.2 Texture Format

To store color values that exceed 1 we need to change the texture format that we use for render textures. If HDR is enabled for both our pipeline and the camera then we need the default HDR format, otherwise we can make do with the regular default. The difference is that HDR textures contain floating-point values instead of 8-bit values for their color channels. So they require more memory, which means that you should only use HDR when you need it.

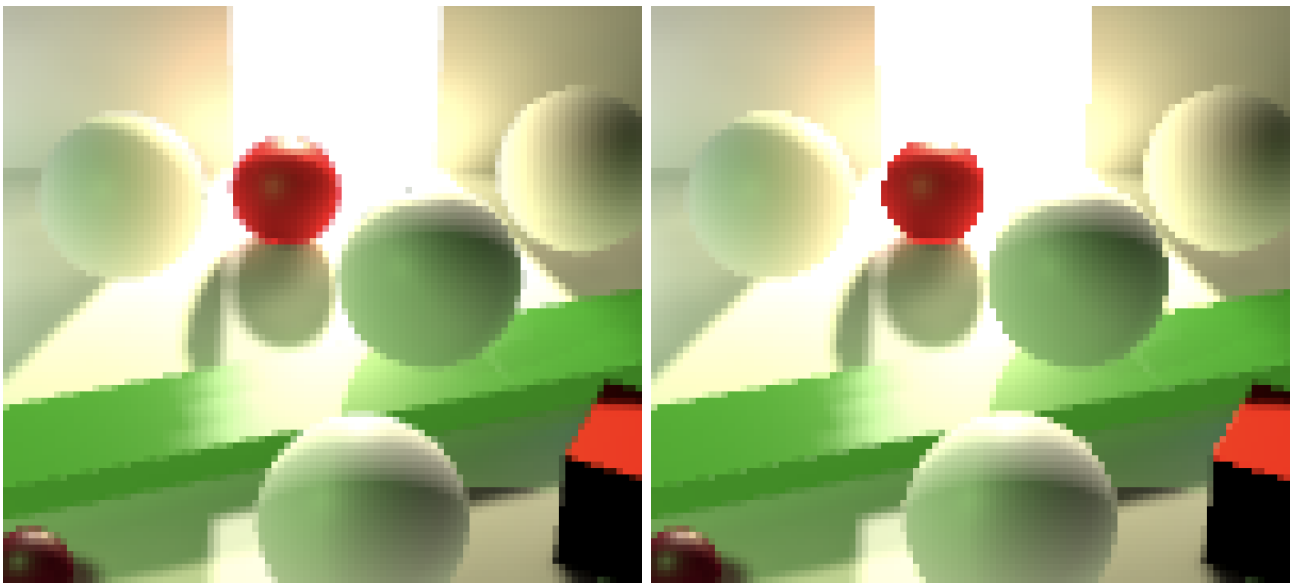
We don't have to base `renderToTexture` on whether HDR is enabled because there is no reason to use HDR when not also using post-processing, as the final camera target is LDR 8-bit per channel anyway.

```
		RenderTextureFormat format = allowHDR && camera.allowHDR ?
			RenderTextureFormat.DefaultHDR : RenderTextureFormat.Default;

		if (renderToTexture) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, renderWidth, renderHeight,
				needsDirectDepth ? 0 : 24,
				FilterMode.Bilinear, format,
				RenderTextureReadWrite.Default, renderSamples
			);
			…
		}
```

What about HDR displays?

Unity doesn't currently support HDR displays, so we can't either. Besides that, a problem of HDR displays is that games that use color grading have to do this after—or at the same time as—tone mapping, which produces a regular LDR image, which then has to be converted back to HDR before sending it to the display, which then does its own thing.

We refrain from using HDR when not using a render texture. So let's also use MSAA, with no other post-processing. The result initially doesn't look different, except that the anti-aliasing quality got worse. That happened because averaging colors only works well when all values are LDR, otherwise the very bright samples dominate the result. So MSAA degrades when HDR colors are involved.



HDR off and on, both with MSAA ×4.

We also have to make sure that we don't unintentionally blit to an intermediate LDR texture during post-processing, as that would eliminate the HDR data. So pass the format to both invocations of the active stack.

```
			activeStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				renderWidth, renderHeight, renderSamples, format
			);

		…

				activeStack.RenderAfterTransparent(
					postProcessingBuffer, cameraColorTextureId,
					cameraDepthTextureId, renderWidth, renderHeight,
					renderSamples, format
				);
```

Add the required parameters in `MyPostProcessingStack`. We only need to use it for the temporary texture in `DepthStripes`, as blurring has to be performed in LDR anyway for best results.

```
	public void RenderAfterOpaque (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height, RenderTextureFormat format
	) {
		InitializeStatic();
		if (depthStripes) {
			DepthStripes(cb, cameraColorId, cameraDepthId, width, height, format);
		}
	}

	public void RenderAfterTransparent (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height, int samples, RenderTextureFormat format
	) { … }

	…

	void DepthStripes (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height, RenderTextureFormat format
	) {
		cb.BeginSample("Depth Stripes");
		cb.GetTemporaryRT(tempTexId, width, height, 0, FilterMode.Point, format);
		…
	}
```

3.3 Tone Mapping

Conversion from HDR to LDR is known as tone mapping, which comes from photography and film development. Traditional photos and film also have a limited range, so many techniques have been developed to perform the conversion. There is no single correct way to deal with this problem. Different approaches can be used to set the mood of the final result, like the classical film look. However, tweaking colors falls under color grading, which comes after tone mapping. We'll only concern ourselves with toning down the brightness of the image so it ends up inside the LDR range.

Tone mapping is a post-processing effect which can be optional, so add a toggle for it to `MyPostProcessingStack`.

```
	[SerializeField]
	bool toneMapping;
```

Tone mapping enabled.

It is done via its own pass, so add an enum value and method for it, initially just blitting with its own profiler sample. Make the source and destination ID parameters `RenderTargetIdentifier` so we can be flexible with what we pass to it.

```
	enum Pass { Copy, Blur, DepthStripes, ToneMapping };

	…

	void ToneMapping (
		CommandBuffer cb,
		RenderTargetIdentifier sourceId, RenderTargetIdentifier destinationId
	) {
		cb.BeginSample("Tone Mapping");
		Blit(cb, sourceId, destinationId, Pass.ToneMapping);
		cb.EndSample("Tone Mapping");
	}
```

In `RenderAfterTransparent` perform tone mapping to the camera target instead of the regular blit, if blurring is disabled. Otherwise create the resolved texture when either tone mapping or MSAA is used, with the appropriate pass. So resolving in this case can mean either an MSAA resolve, a tone mapping resolve, or both.

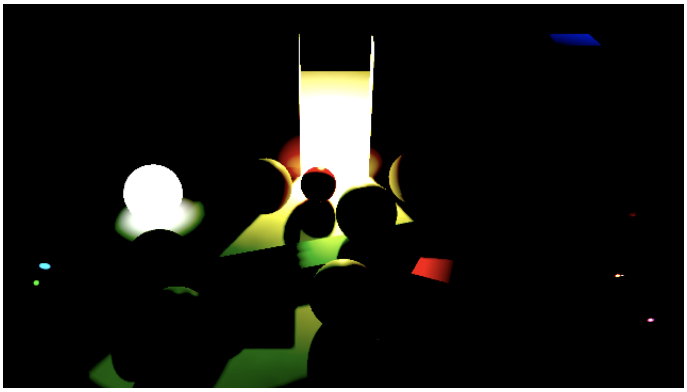
```
	public void RenderAfterTransparent (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height, int samples
	) {
		if (blurStrength > 0) {
			if (toneMapping || samples > 1) {
				cb.GetTemporaryRT(
					resolvedTexId, width, height, 0, FilterMode.Bilinear
				);
				if (toneMapping) {
					ToneMapping(cb, cameraColorId, resolvedTexId);
				}
				else {
					Blit(cb, cameraColorId, resolvedTexId);
				}
				Blur(cb, resolvedTexId, width, height);
				cb.ReleaseTemporaryRT(resolvedTexId);
			}
			else {
				Blur(cb, cameraColorId, width, height);
			}
		}
		else if (toneMapping) {
			ToneMapping(cb, cameraColorId, BuiltinRenderTextureType.CameraTarget);
		}
		else {
			Blit(cb, cameraColorId, BuiltinRenderTextureType.CameraTarget);
		}
	}
```

Add the new pass to the *PostEffectStack* shader.

```
		Pass { // 3 ToneMapping
			HLSLPROGRAM
			#pragma target 3.5
			#pragma vertex DefaultPassVertex
			#pragma fragment ToneMappingPassFragment
			ENDHLSL
		}
```

And add the required function to the HLSL file. Initially return the color minus 1, saturated. That gives us an indication of which pixels contain over-bright colors.

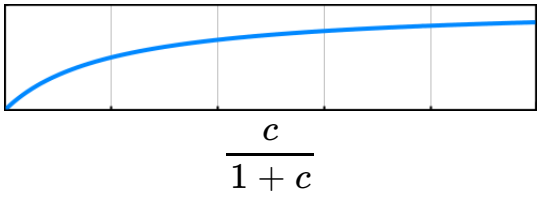
```
float4 ToneMappingPassFragment (VertexOutput input) : SV_TARGET {
	float3 color = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv).rgb;
	color -= 1;
	return float4(saturate(color), 1);
}
```



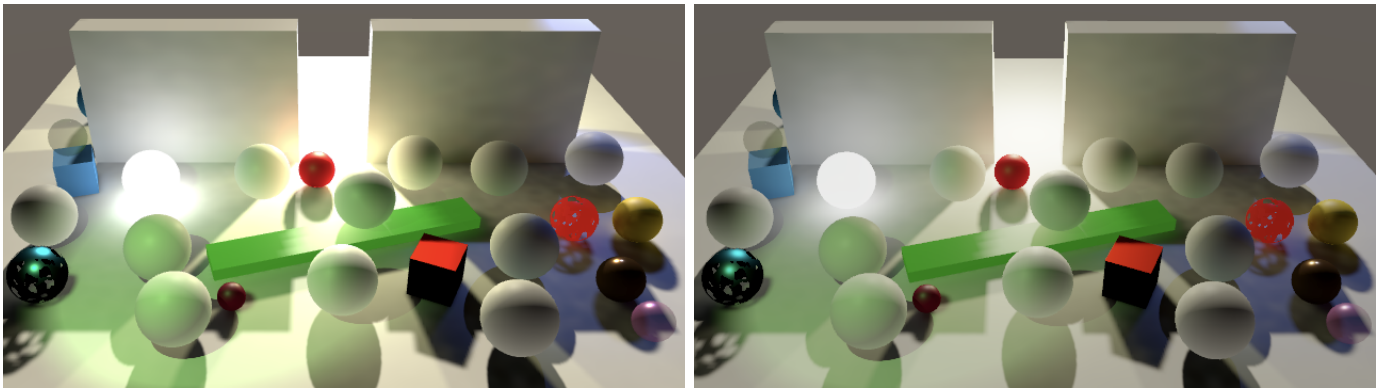
Only over-bright colors.

3.4 Reinhard

The goal of tone mapping is to reduce the brightness of the image so that otherwise uniform white regions show a variety of colors, revealing the details that were otherwise lost. It's like when your eyes adjust to a suddenly bright environment until you can see clearly again. But we don't want to scale down the entire image uniformly, because that would make darker colors indistinguishable, trading over-brightness for underexposure. So we need a nonlinear conversion that doesn't reduce dark values much but reduces high values a lot. At the extremes, zero remains zero and a value that approaches infinity is reduced to 1. A function that accomplishes that is $\frac{c}{1+c}$ where c is a color channel. That function is known as the Reinhard tone mapping operation, initially proposed by Mark Reinhard, except that he applies it to luminance while we'll apply it to individual color channels. Make our tone mapping pass use it.

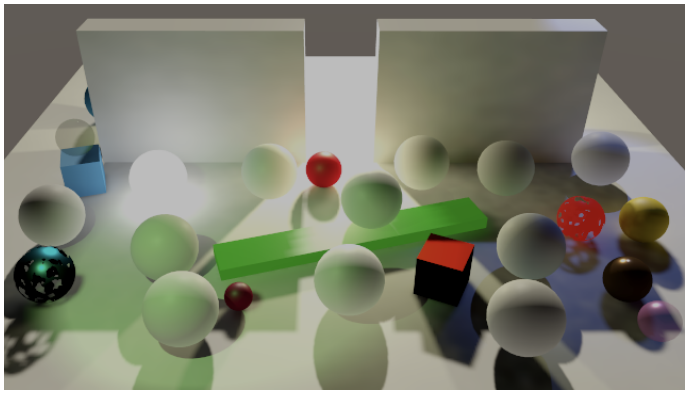



```
	color /= 1 + color;
```



Without and with Reinhard RGB tone mapping.

The result is an image that is guaranteed to not have any over-bright pixels, but the overall image has been desaturated and darkened somewhat. Pixels that were exactly at full intensity have been halved, which you can clearly see by applying tone mapping while HDR is disabled.



Tone mapping applied to LDR image.

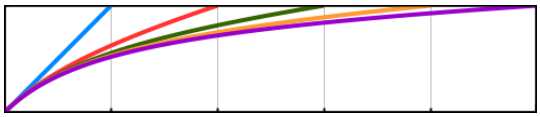
Couldn't we perform tone-mapping before resolving MS textures?

Yes, with a custom resolve pass, which would have to perform tone mapping on each sample before averaging them, which is more expensive than doing it once per pixel. Besides that, post-processing effects like bloom require HDR data. To make those work the resolve pass has to convert back to HDR after averaging, to approximate the original intensity.

3.5 Modified Reinhard

There are multiple approaches to tone mapping—and color grading can be used to tweak it further—but per-channel RGB Reinhard is the simplest, so we'll keep it. But a simple adjustment that we could make is to limit the strength of the effect, by adjusting the value range that gets compressed to LDR. Anything beyond that range remains over-bright. That allows us to reduce the adjustment for less-bright scenes, or accept some over-brightness to keep darker colors intact.

This adjustment is also described by Reinhard and transforms the function to $\frac{c\left(1+\frac{c}{w^2}\right)}{1+c}$ where w is the white point, or the maximum tone mapping range in our case. If w is infinite then we have the original function again, and when it is 1 then tone mapping does nothing.



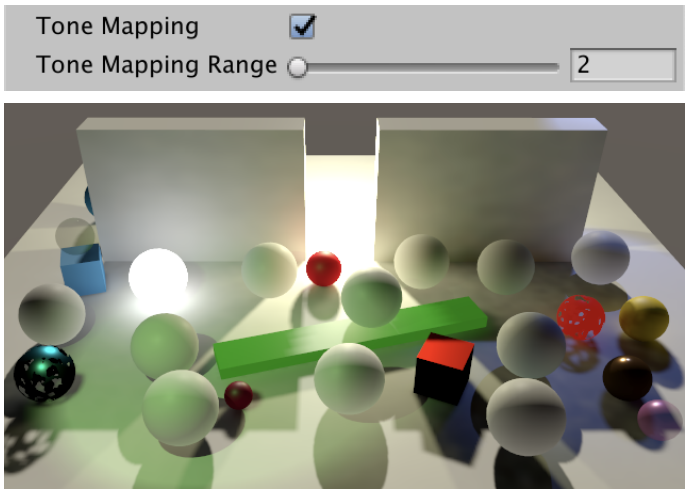
$\frac{c\left(1+\frac{c}{w^2}\right)}{1+c}$ *for w from 1 to 5.*

Add a configuration option for the range, with a minimum of 1 as lower values would over-expose the entire image. The maximum can be 100, which is plenty to approximate the original function. Then calculate $m = \frac{1}{w^2}$ and send that to the GPU as a modifier for the Reinhard function.

```
	[SerializeField, Range(1f, 100f)]
	float toneMappingRange = 100f;

	…

	void ToneMapping (
		CommandBuffer cb,
		RenderTargetIdentifier sourceId, RenderTargetIdentifier destinationId
	) {
		cb.BeginSample("Tone Mapping");
		cb.SetGlobalFloat(
			"_ReinhardModifier", 1f / (toneMappingRange * toneMappingRange)
		);
		Blit(cb, sourceId, destinationId, Pass.ToneMapping);
		cb.EndSample("Tone Mapping");
	}
```

The shader then only has to calculate $\frac{c(1+cm)}{1+c}$.

```
float _ReinhardModifier;

…

float4 ToneMappingPassFragment (VertexOutput input) : SV_TARGET {
	float3 color = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv).rgb;
	color *= (1 + color * _ReinhardModifier) / (1 + color);
	return float4(saturate(color), 1);
}
```



Tone mapping range reduced to 2.

This ends the original SRP tutorial series, which started when it was still experimental. A lot has changed in Unity 2019. There is a new Custom SRP series for that, which covers both old and new topics with a more modern approach.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

BECOME A PATRON

Or make a direct donation!

made by Jasper Flick