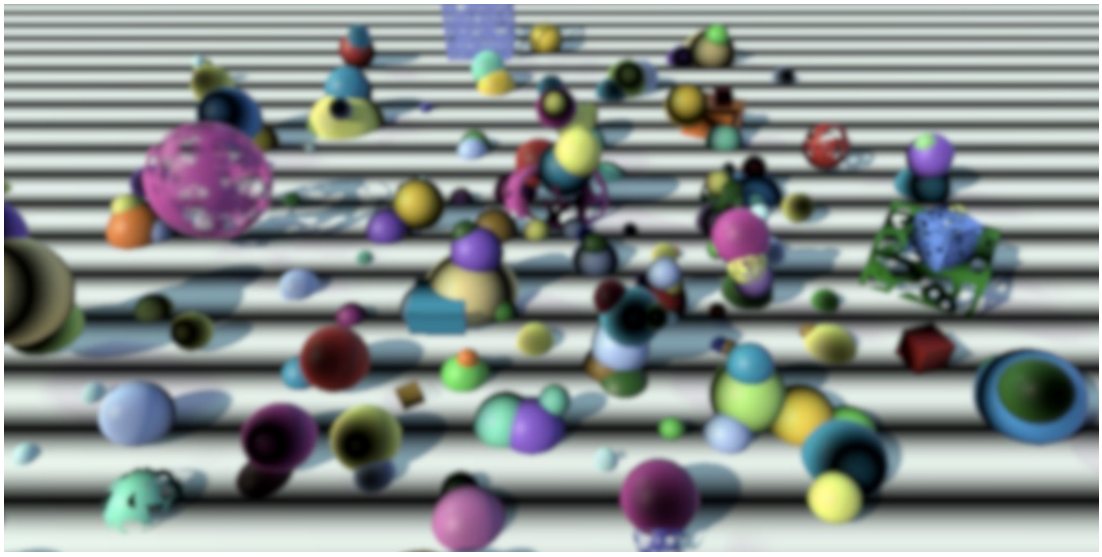
Post-Processing Full-Screen Effects

Create a post-processing stack asset.
Use render textures.
Draw a full-screen triangle.
Apply a multi-step blur effect and depth-based stripes.
Configure a stack per camera.

This is the eleventh installment of a tutorial series covering Unity's scriptable render pipeline. It covers the creation of a post-processing stack.

This tutorial is made with Unity 2018.4.4f1.



Messing with the image.

1 Post-Processing Stack

Besides rendering geometry that's part of the scene, it is also possible to alter the generated image afterwards. This is used to apply full-screen effects like ambient occlusion, bloom, color grading, and depth-of-field. Usually multiple post-processing steps are applied in a specific order, which is configured via one or multiple assets or components, collectively forming a post-processing stack. Unity has multiple implementations of such a stack.

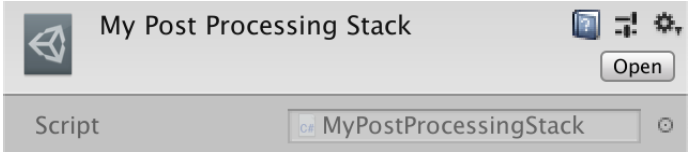
In this tutorial we'll create a simple post-processing stack of our own, with two effects to see it in action. You could extend it to support more useful effects, or alter the approach so you can connect to an existing solution.

1.1 Asset

We'll introduce a `MyPostProcessingStack` asset type to control post-processing. Give it a public `Render` method, with a `CommandBuffer` parameter that it can use to do its work. The idea is that the stack will fill the buffer with commands, but executing and clearing the buffer is the responsibility of the pipeline. Initially, just log that the stack's method got invoked.

```
using UnityEngine;
using UnityEngine.Rendering;

[CreateAssetMenu(menuName = "Rendering/My Post-Processing Stack")]
public class MyPostProcessingStack : ScriptableObject {

	public void Render (CommandBuffer cb) {
		Debug.Log("Rendering Post-Processing Stack");
	}
}
```

Create an asset for our stack. It doesn't have any configuration options yet, but we'll add some later.



Post-processing stack asset.

1.2 Default Stack

To use a stack `MyPipeline` needs a reference to it. Give it a field to keep track of a default stack, which is set via its constructor.

```
	MyPostProcessingStack defaultStack;

	public MyPipeline (
		bool dynamicBatching, bool instancing, MyPostProcessingStack defaultStack,
		Texture2D ditherTexture, float ditherAnimationSpeed,
		int shadowMapSize, float shadowDistance, float shadowFadeRange,
		int shadowCascades, Vector3 shadowCascasdeSplit
	) {
		…
		if (instancing) {
			drawFlags |= DrawRendererFlags.EnableInstancing;
		}

		this.defaultStack = defaultStack;

		…
	}
```

Give `MyPipelineAsset` a configuration option for a default stack as well, so it can pass it to the pipeline instance.

```
	[SerializeField]
	bool dynamicBatching;

	…

	protected override IRenderPipeline InternalCreatePipeline () {
		Vector3 shadowCascadeSplit = shadowCascades == ShadowCascades.Four ?
			fourCascadesSplit : new Vector3(twoCascadesSplit, 0f);
		return new MyPipeline(
			dynamicBatching, instancing, defaultStack,
			ditherTexture, ditherAnimationSpeed,
			(int)shadowMapSize, shadowDistance, shadowFadeRange,
			(int)shadowCascades, shadowCascadeSplit
		);
	}
```

Make our single stack asset the default.

Default stack assigned.

1.3 Rendering the Stack

To isolate rendering of the stack, add a command buffer dedicated to post-processing effects to `MyPipeline`. If a default stack exists, have it render with the buffer, then execute and clear the buffer. Post-processing happens after regular rendering is finished, so after invoking `DrawDefaultPipeline` in `Render`.

```
	CommandBuffer postProcessingBuffer = new CommandBuffer {
		name = "Post-Processing"
	};

	…

	void Render (ScriptableRenderContext context, Camera camera) {
		…

		DrawDefaultPipeline(context, camera);

		if (defaultStack) {
			defaultStack.Render(postProcessingBuffer);
			context.ExecuteCommandBuffer(postProcessingBuffer);
			postProcessingBuffer.Clear();
		}

		cameraBuffer.EndSample("Render Camera");
		context.ExecuteCommandBuffer(cameraBuffer);
		cameraBuffer.Clear();

		context.Submit();

		…
	}
```

At this point the stack should log that it gets invoked each time a frame is rendered.

2 Render Targets

To alter the rendered image we have to read from it. The simplest and most robust way to make that possible is to have our pipeline render to a texture. Up to this point we've always rendered to whatever the camera's target is. It's usually the frame buffer, but it can also be a render texture, for example when rendering the faces of a reflection probe. Unity also always renders to a texture for the scene window and its small camera preview when one is selected.

2.1 Rendering to a Texture

Before clearing the render target, we have to get a temporary render texture if there is a stack. This time we'll use `CommandBuffer.GetTemporaryRT` to schedule the acquisition of the texture, using the camera buffer. This approach requires us to supply a shader property ID, along with the width and height of the texture, which should match the camera's pixel dimensions. Let's use *_CameraColorTexture* for the shader property name.

```
	static int cameraColorTextureId = Shader.PropertyToID("_CameraColorTexture");

	…

	void Render (ScriptableRenderContext context, Camera camera) {
		…
		context.SetupCameraProperties(camera);

		if (defaultStack) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, camera.pixelWidth, camera.pixelHeight
			);
		}

		CameraClearFlags clearFlags = camera.clearFlags;

		…
	}
```

That will give us our texture, bound to the provided ID. Next, we have to make it the render target. That's done by invoking `SetRenderTarget` on the camera buffer with the ID as a parameter. The ID has be a `RenderTargetIdentifier`, but there is an implicit cast from `int` to that type, assuming that it is a shader property ID. Also, we can specify the load and store actions. We'll simply assume that we're working with a single camera, so don't care about the initial state of the texture, as we'll clear it next.

```
		if (defaultStack) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, camera.pixelWidth, camera.pixelHeight
			);
			cameraBuffer.SetRenderTarget(
				cameraColorTextureId,
				RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
			);
		}
```

We have to release the render texture after post-processing, if applicable. That's done by invoking `ReleaseTemporaryRT` on the camera buffer with the same ID. This isn't strictly necessary as textures claimed by the buffer should be released automatically once the camera is done rendering, but it's good practice to clean up explicitly as soon as possible.

```
		if (defaultStack) {
			defaultStack.Render(postProcessingBuffer);
			context.ExecuteCommandBuffer(postProcessingBuffer);
			postProcessingBuffer.Clear();
			cameraBuffer.ReleaseTemporaryRT(cameraColorTextureId);
		}
```

Could we cache the `RenderTargetIdentifier` for reuse?

Yes, that way the conversion only happens once, which is more efficient. However, I won't bother with that in this tutorial.

2.2 Blitting

At this point our scene appears to no longer get rendered, because we're rendering to a texture instead of the camera's target. To fix this we'll have `MyPostProcessingStack.Render` copy the texture's contents to the final target. We can do that by invoking `Blit` on the buffer with the source and destination IDs as arguments. Add the camera texture's color ID as a parameter for this purpose, and use `BuiltinRenderTextureType.CameraTarget` for the destination, which also implicitly gets converted to `RenderTargetIdentifier`.

```
	public void Render (CommandBuffer cb, int cameraColorId) {
		//Debug.Log("Rendering Post-Processing Stack");
		cb.Blit(cameraColorId, BuiltinRenderTextureType.CameraTarget);
	}
```

What does blit mean?

It comes from an old bit boundary block transfer routine name *BitBLT*, shortened to blit.

Add the color texture ID argument in `MyPipeline.Render`.

```
		if (defaultStack) {
			defaultStack.Render(postProcessingBuffer, cameraColorTextureId);
			context.ExecuteCommandBuffer(postProcessingBuffer);
			postProcessingBuffer.Clear();
			cameraBuffer.ReleaseTemporaryRT(cameraColorTextureId);
		}
```

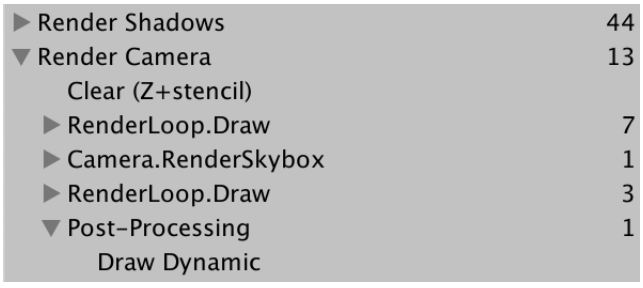
We see results again, but the skybox gets drawn on top of everything rendered before it, so only transparent objects remain visible. That happens because we're no longer using a depth buffer. We can reactive the depth buffer by adding another argument to `GetTemporaryRT` to specify the amount of bits used for depth. It's zero by default, which disables the depth buffer. We have to use 24 to reactivate it.

```
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, camera.pixelWidth, camera.pixelHeight, 24
			);
```

Why 24 bits?

The other option is 16 bits, but we want to use the highest possible precision for depth values, which is 24 bits. Sometimes the depth buffer precision is listed as 32, but the extra eight bits are for the stencil buffer, not depth. You could specify 32 but it would act the same as 24.

Our scene now appears to get rendered as usual. However, inspecting the frame debugger will reveal that another step was added. The nested execution of the post-processing command buffer automatically gets sampled. Inside its scope, the blit action is listed as *Draw Dynamic*.



Post-processing draw call.

2.3 Separate Depth Texture

Some post-processing effects rely on depth information, which they have to acquire by reading from the depth buffer. To make that possible we have to explicitly render depth information to a texture with its own ID, for which we'll use *_CameraDepthTexture*. Getting a depth texture works the same as the color one, except that we have to use a different texture format. This require us to invoke `GetTemporaryRT` a second time, with two extra arguments. First the filter mode, which should be the default `FilterMode.Point`, followed by `RenderTextureFormat.Depth`. The depth bits of the color texture should be set back to zero, which is the default but let's be explicit.

```
	static int cameraColorTextureId = Shader.PropertyToID("_CameraColorTexture");
	static int cameraDepthTextureId = Shader.PropertyToID("_CameraDepthTexture");

	…

	void Render (ScriptableRenderContext context, Camera camera) {
		…

		if (defaultStack) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, camera.pixelWidth, camera.pixelHeight, 0
			);
			cameraBuffer.GetTemporaryRT(
				cameraDepthTextureId, camera.pixelWidth, camera.pixelHeight, 24,
				FilterMode.Point, RenderTextureFormat.Depth
			);
			…
		}

		…
	}
```

Next, we have to invoke the variant of `SetRenderTarget` that allows us to specify a separate depth buffer, with its own load and store actions.

```
			cameraBuffer.SetRenderTarget(
				cameraColorTextureId,
				RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store,
				cameraDepthTextureId,
				RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
			);
```

Pass the ID for depth to the stack as well, and release the depth texture once we're done.

```
		if (defaultStack) {
			defaultStack.Render(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId
			);
			context.ExecuteCommandBuffer(postProcessingBuffer);
			postProcessingBuffer.Clear();
			cameraBuffer.ReleaseTemporaryRT(cameraColorTextureId);
			cameraBuffer.ReleaseTemporaryRT(cameraDepthTextureId);
		}
```

Add the required parameter to `MyPostProcessingStack.Render`. After that the scene should be rendered as normal again.

```
	public void Render (CommandBuffer cb, int cameraColorId, int cameraDepthId) {
		cb.Blit(cameraColorId, BuiltinRenderTextureType.CameraTarget);
	}
```

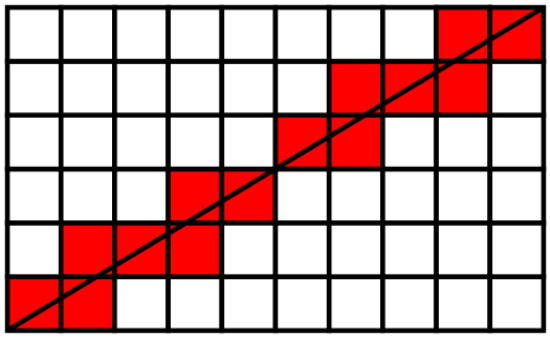
It's now also possible to use the depth texture as the source for the blit, which would show the raw depth information instead of colors. The result of that depends on the graphics API.

Raw depth.

3 Full-Screen Triangle

Blitting a texture is fundamentally the same as rendering regular geometry. It's done by rendering a full-screen quad with a shader that samples the texture based on its screen-space position. You can see a hint of this by inspecting the *Dynamic Draw* entry in the frame debugger. The color texture is assigned to *_MainTex* and it uses four vertices and indices.

So `Blit` renders a quad made from two triangles. This works, but could be done in a more efficient way, by using a single triangle that cover the entire screen instead. The obvious benefit of that is the reduction of vertices and indices to three. However, the more significant difference is that it eliminates the diagonal where the two triangles of the quad meet. Because GPUs render fragments parallel in small blocks, some fragments end up wasted along the edges of triangles. As the quad has two triangles, the fragment blocks along the diagonal get rendered twice, which is inefficient. Besides that rendering a single triangle can have better local cache coherency.



Redundant block rendering, exaggerated.

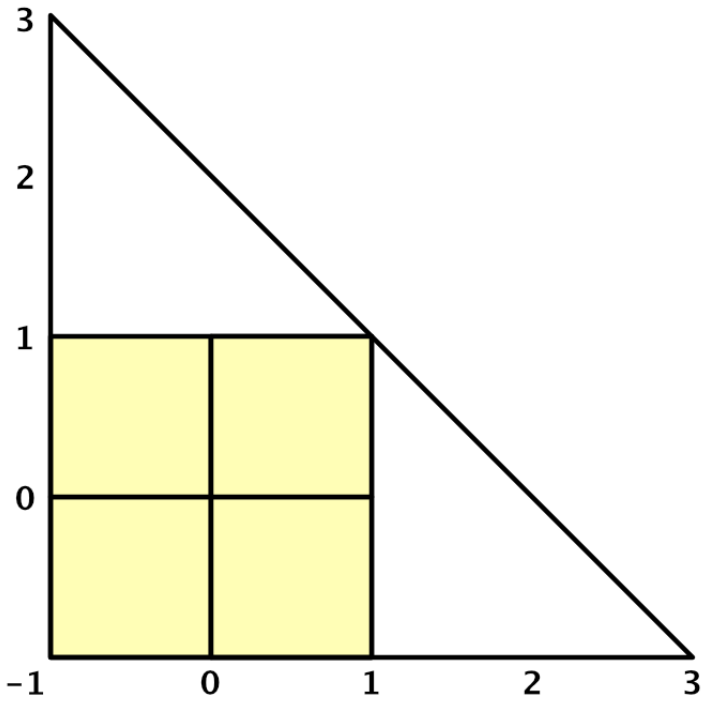
While the performance difference between a quad and single triangle might be tiny, it's enough that the standard approach nowadays is to go with the full-screen triangle, so we'll use it as well. However, Unity doesn't have a standard blit method for that, so we have to create one ourselves.

3.1 Mesh

The first step is to create the triangle. We'll keep track of it via a static `Mesh` field in `MyPostProcessingStack` and create it when needed via a static `InitializeStatic` method, which we invoke at the start of `Render`.

```
	static Mesh fullScreenTriangle;

	static void InitializeStatic () {
		if (fullScreenTriangle) {
			return;
		}
	}

	public void Render (CommandBuffer cb, int cameraColorId, int cameraDepthId) {
		InitializeStatic();
		cb.Blit(cameraColorId, BuiltinRenderTextureType.CameraTarget);
	}
```

The mesh needs three vertices and a single triangle. We'll draw it directly in clip space so we can skip a matrix multiplication and ignore the Z dimension. This means that the center of the screen is the origin and the XY coordinates are either -1 or 1 at the edges. The direction of the Y axis depends on the platform, but that doesn't matter for our triangle. To create a full-screen triangle, you can use vertices $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$, $\begin{bmatrix} -1 \\ 3 \end{bmatrix}$, and $\begin{bmatrix} 3 \\ -1 \end{bmatrix}$.



Triangle relative to clip space.

```
	static void InitializeStatic () {
		if (fullScreenTriangle) {
			return;
		}
		fullScreenTriangle = new Mesh {
			name = "My Post-Processing Stack Full-Screen Triangle",
			vertices = new Vector3[] {
				new Vector3(-1f, -1f, 0f),
				new Vector3(-1f,  3f, 0f),
				new Vector3( 3f, -1f, 0f)
			},
			triangles = new int[] { 0, 1, 2 },
		};
		fullScreenTriangle.UploadMeshData(true);
	}
```

3.2 Shader

The second step is to write a shader to copy the texture. Create a *Hidden/My Pipeline/PostEffectStack* shader for that with a single pass that doesn't perform culling and ignores depth. Have it use `CopyPassVertex` and `CopyPassFragment` functions, which we'll define in a separate *PostEffectStack.hlsl* include file.

```
Shader "Hidden/My Pipeline/PostEffectStack" {
	SubShader {
		Pass {
			Cull Off
			ZTest Always
			ZWrite Off

			HLSLPROGRAM
			#pragma target 3.5
			#pragma vertex CopyPassVertex
			#pragma fragment CopyPassFragment
			#include "../ShaderLibrary/PostEffectStack.hlsl"
			ENDHLSL
		}
	}
}
```

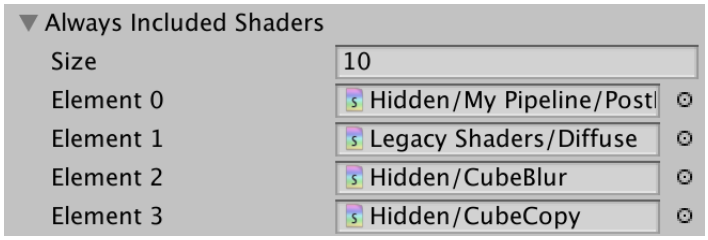
The shader code is short. We only need the vertex position, which doesn't have to be transformed. Besides that we'll output UV coordinates per vertex, which are simply the XY coordinates halved plus ½. We use those per fragment to sample the texture. We can directly sample *_CameraColorTexture*, so let's start with that.

```
#ifndef MYRP_POST_EFFECT_STACK_INCLUDED
#define MYRP_POST_EFFECT_STACK_INCLUDED

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

TEXTURE2D(_CameraColorTexture);
SAMPLER(sampler_CameraColorTexture);

struct VertexInput {
	float4 pos : POSITION;
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
	float2 uv : TEXCOORD0;
};

VertexOutput CopyPassVertex (VertexInput input) {
	VertexOutput output;
	output.clipPos = float4(input.pos.xy, 0.0, 1.0);
	output.uv = input.pos.xy * 0.5 + 0.5;
	return output;
}

float4 CopyPassFragment (VertexOutput input) : SV_TARGET {
	return SAMPLE_TEXTURE2D(
		_CameraColorTexture, sampler_CameraColorTexture, input.uv
	);
}

#endif // MYRP_POST_EFFECT_STACK_INCLUDED
```

Have `MyPostProcessingStack` keep track of a static material that uses this shader. `Shader.Find` is the simplest way to get a hold of it.

```
	static Material material;

	static void InitializeStatic () {
		…

		material =
			new material(Shader.Find("Hidden/My Pipeline/PostEffectStack")) {
				name = "My Post-Processing Stack material",
				hideFlags = HideFlags.HideAndDontSave
			};
	}
```

That always works in the editor, but will fail in a build if the shader is not included. We can enforce that by adding it to the *Always Included Shaders* array in the *Graphics* project settings. There are other ways to ensure that the shader gets included, but this is the approach that requires the least amount of code.



Always include post-processing shader.

3.3 Drawing

Now we can copy the color texture by invoking `CommandBuffer.DrawMesh` instead of `Blit`. At minimum, we need to specify the mesh, transformation matrix, and material to use. As we don't transform vertices any matrix will do.

```
	public void Render (CommandBuffer cb, int cameraColorId, int cameraDepthId) {
		InitializeStatic();
		//cb.Blit(cameraColorId, BuiltinRenderTextureType.CameraTarget);
		cb.DrawMesh(fullScreenTriangle, Matrix4x4.identity, material);
	}
```

But `Blit` does more than just draw a quad. It also sets the render target. We now have to do that ourselves.

```
		cb.SetRenderTarget(
			BuiltinRenderTextureType.CameraTarget,
			RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
		);
		cb.DrawMesh(fullScreenTriangle, Matrix4x4.identity, material);
```

We now render the final result with our own triangle, which you can verify via the frame debugger. The draw call is now listed as *Draw Mesh* and uses only three vertices and no matrix. The result looks good, except that it might appear upside down. That happens because Unity performs a vertical flip in some cases to get consistent results. For example, when not using OpenGL the scene view window and small camera preview will be flipped.

Our shader can detect whether a flip happens by checking the X component of the `_ProjectionParams` vector, which got set when our pipeline invoked `SetupCameraProperties`. If it is negative then we should flip the V coordinate.

```
float4 _ProjectionParams;

…

VertexOutput CopyPassVertex (VertexInput input) {
	…
	if (_ProjectionParams.x < 0.0) {
		output.uv.y = 1.0 - output.uv.y;
	}
	return output;
}
```

3.4 Variable Source Texture

`CommandBuffer.Blit` can work with any source texture. It does this by binding it to the *_MainTex* shader property. We can do the same by invoking `CommandBuffer.SetGlobalTexture` before drawing our triangle in `MyPostProcessingStack.Render`.

```
	static int mainTexId = Shader.PropertyToID("_MainTex");

	…

	public void Render (
		CommandBuffer cb, int cameraColorId, int cameraDepthId
	) {
		cb.SetGlobalTexture(mainTexId, cameraColorId);
		…
	}
```

Then adjust the shader so it samples *_MainTexture* instead of *_CameraColorTexture*. This way our stack no longer needs to know which shader property the pipeline uses.

```
TEXTURE2D(_MainTex);
SAMPLER(sampler_MainTex);

…

float4 CopyPassFragment (VertexOutput input) : SV_TARGET {
	return SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv);
}
```

4 Blurring

To see our post-processing stack in action, let's create a simple blur effect.

4.1 Shader

We'll put the code for all of our post-processing effects in the same shader, using a different pass for each. That way we can reuse code in the shader file and only have to deal with a single material. Begin by renaming `CopyPassVertex` to `DefaultPassVertex` in the HLSL file, because it's a simple vertex program that can be used for many effects. Then add a `BlurPassFragment`, initially a duplicate of `CopyPassFragment`.

```
VertexOutput DefaultPassVertex (VertexInput input) {
	…
}

float4 CopyPassFragment (VertexOutput input) : SV_TARGET {
	return SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv);
}

float4 BlurPassFragment (VertexOutput input) : SV_TARGET {
	return SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv);
}
```

Then adjust the shader file to match, adding a second pass for blurring. Move the culling and depth configuration up to the subshader level so we don't have to repeat that code. The include directive can also be shared this way, by putting it inside an `HLSLINCLUDE` block.

```
Shader "Hidden/My Pipeline/PostEffectStack" {
	SubShader {
		Cull Off
		ZTest Always
		ZWrite Off

		HLSLINCLUDE
		#include "../ShaderLibrary/PostEffectStack.hlsl"
		ENDHLSL

		Pass { // 0 Copy
			//Cull Off
			//ZTest Always
			//ZWrite Off

			HLSLPROGRAM
			#pragma target 3.5
			#pragma vertex DefaultPassVertex
			#pragma fragment CopyPassFragment
			ENDHLSL
		}

		Pass { // 1 Blur
			HLSLPROGRAM
			#pragma target 3.5
			#pragma vertex DefaultPassVertex
			#pragma fragment BlurPassFragment
			ENDHLSL
		}
	}
}
```

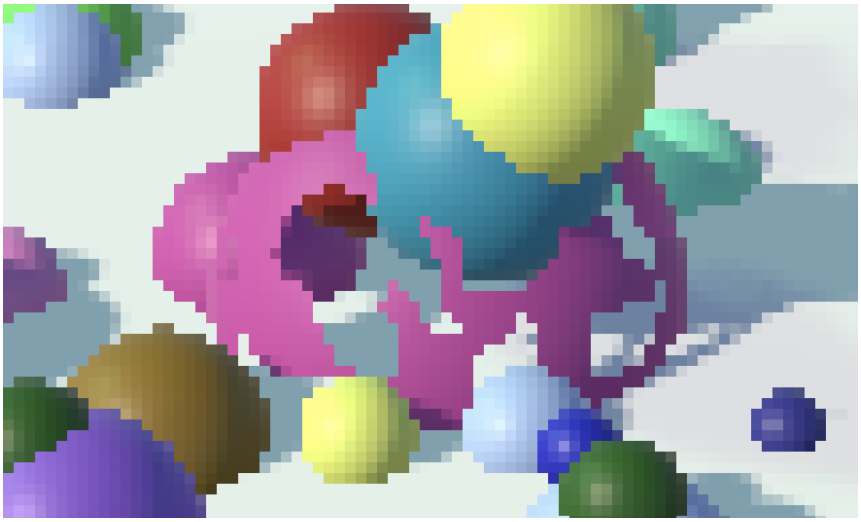
Now we can select the blur pass in `MyPostProcessingStack.Render`, by adding 1 as a fourth argument. The required third argument is the submesh index, which is always zero. To make it clearer which pass we are rendering, define a `Pass` enum inside `MyPostProcessingStack` for the copy and blur passes.

```
	enum Pass { Copy, Blur };

	…

	public void Render (CommandBuffer cb, int cameraColorId, int cameraDepthId) {
		…
		cb.DrawMesh(
			fullScreenTriangle, Matrix4x4.identity, material, 0, (int)Pass.Blur
		);
	}
```

4.2 Filtering

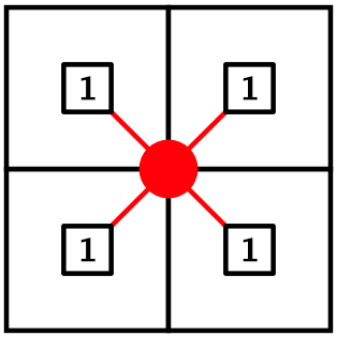
Blurring is done by filtering the image, which means sampling and combining multiple pixels of the source texture per rendered fragment. To make that easy, add a `BlurSample` function to the HLSL file that has parameters for the original UV coordinates plus separate U and V offsets. The offsets are defined in pixels. We can use the relevant screen-space derivatives of the U and V coordinates to convert the offsets to UV space. Begin by sampling the source texture without any offset. As the effect works at pixel scale, it's easiest to see by increasing the scale factor of the game window.

```
float4 BlurSample (float2 uv, float uOffset = 0.0, float vOffset = 0.0) {
	uv += float2(uOffset * ddx(uv.x), vOffset * ddy(uv.y));
	return SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, uv);
}

float4 BlurPassFragment (VertexOutput input) : SV_TARGET {
	return BlurSample(input.uv);
}
```



Unmodified image at ×10 scale.

The simplest blur operation is a 2×2 box filter, which averages a block of four pixels. We could do that by sampling four times, but we can also do it by sampling once at the corner of four pixels, by offsetting the UV coordinates half a pixel in both dimensions. Bilinear texture filtering will then take care of averaging for us.



2x2 box filter.

```
	return BlurSample(input.uv, 0.5, 0.5);
```

However, the default filter mode is point, which clamps to the nearest pixel, so that currently only moves the image. We have to change `MyPipeline.Render` so it uses bilinear filtering for its color texture. This change only matters when not sampling at the center of pixels.

```
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, camera.pixelWidth, camera.pixelHeight, 0,
				FilterMode.Bilinear
			);
```

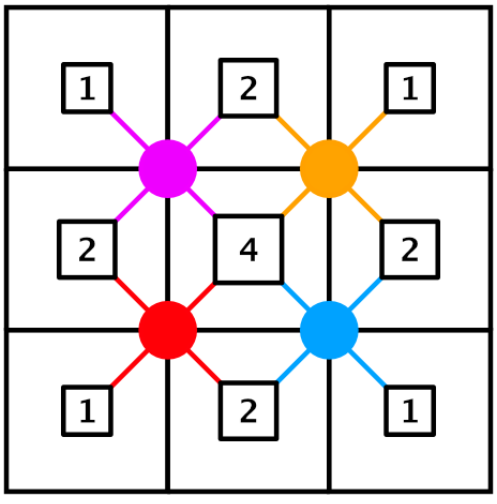
2x2 box filter applied.

While this blurs the image, it also moves it a bit due to the offset. We can eliminate the directional bias by sampling four times with offsets in all four diagonal directions, then averaging them. As this is the final rendering step we don't need the alpha channel so can set it to 1. That way we avoid calculating the average of the alpha channel.

```
float4 BlurPassFragment (VertexOutput input) : SV_TARGET {
	float4 color =
		BlurSample(input.uv, 0.5, 0.5) +
		BlurSample(input.uv, -0.5, 0.5) +
		BlurSample(input.uv, 0.5, -0.5) +
		BlurSample(input.uv, -0.5, -0.5);
	return float4(color.rgb * 0.25, 1);
}
```

Averaging four samples.

This covers a 3×3 pixel region with overlapping 2×2 samples, which means that pixels nearer to the center contribute more to the final color. This operation is known as 3×3 tent filter.



3x3 tent filter.

4.3 Blurring Twice

The blur effect might appear strong when zoomed in, but is subtle when zoomed out and might be hardly noticeable when rendering at a high resolution. We can strengthen the effect by increasing the filter region further, but that also makes the pass more complex. Another approach is to keep the filer that we have but apply it more than once. For example, performing a second blur pass would increase the filter size to 5×5. Let's do that.

First, put all code for a single blit in a separate `Blit` method so we can reuse it. Its parameters are the command buffer, source and destination IDs, and the pass.

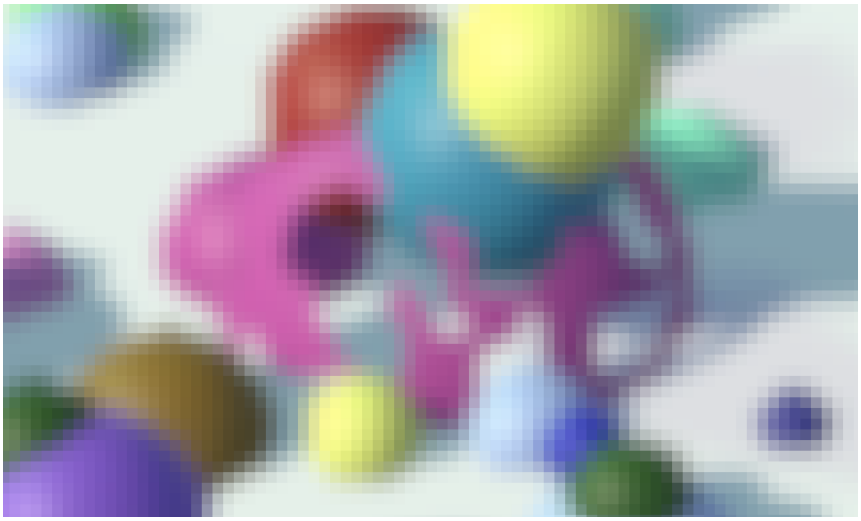
```
	public void Render (CommandBuffer cb, int cameraColorId, int cameraDepthId) {
		InitializeStatic();
		Blit(
			cb, cameraColorId, BuiltinRenderTextureType.CameraTarget, Pass.Blur
		);
	}

	void Blit (
		CommandBuffer cb,
		RenderTargetIdentifier sourceId, RenderTargetIdentifier destinationId,
		Pass pass = Pass.Copy
	) {
		cb.SetGlobalTexture(mainTexId, sourceId);
		cb.SetRenderTarget(
			destinationId,
			RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
		);
		cb.DrawMesh(
			fullScreenTriangle, Matrix4x4.identity, material, 0, (int)pass
		);
	}
```

Now we can blit twice in `Render`, but we cannot blit from the color texture to itself. The result would be undefined and differs per platform. So we have to get a temporary render texture to store the intermediate result. To be able to create this texture we have to add the width and height as parameters.

```
	static int tempTexId = Shader.PropertyToID("_MyPostProcessingStackTempTex");

	…

	public void Render (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		InitializeStatic();
		cb.GetTemporaryRT(tempTexId, width, height, 0, FilterMode.Bilinear);
		Blit(cb, cameraColorId, tempTexId, Pass.Blur);
		Blit(cb, tempTexId, BuiltinRenderTextureType.CameraTarget, Pass.Blur);
		cb.ReleaseTemporaryRT(tempTexId);
	}
```

Supply the width and height in `MyPipeline.Render`.

```
			defaultStack.Render(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				camera.pixelWidth, camera.pixelHeight
			);
```



Blurred twice.

4.4 Configurable Blur

Blurring twice produces softer results, but still won't be obvious at high resolutions. To make it stand out we'll have to add a few more passes. Let's make this configurable by adding a blur strength slider to `MyPostProcessingStack`.

```
	[SerializeField, Range(0, 10)]
	int blurStrength;
```

Move the blurring to a separate `Blur` method. Invoke it in `Render` only when the strength is positive, otherwise perform a regular copy.

```
	public void Render (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		InitializeStatic();
		if (blurStrength > 0) {
			Blur(cb, cameraColorId, width, height);
		}
		else {
			Blit(cb, cameraColorId, BuiltinRenderTextureType.CameraTarget);
		}
	}
```

Let's begin by always blurring twice when the strength is greater than one. If not we can suffice with a single blur straight to the camera target.

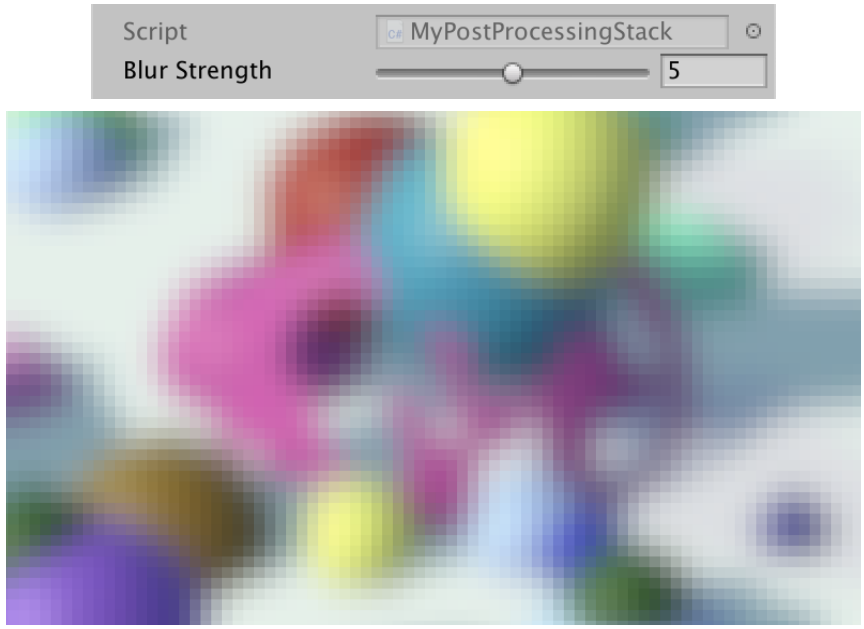
```
	void Blur (CommandBuffer cb, int cameraColorId, int width, int height) {
		cb.GetTemporaryRT(tempTexId, width, height, 0, FilterMode.Bilinear);
		}
		if (blurStrength > 1) {
			Blit(cb, cameraColorId, tempTexId, Pass.Blur);
			Blit(cb, tempTexId, BuiltinRenderTextureType.CameraTarget, Pass.Blur);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

We can make this work for any strength by beginning with a loop in which we perform a double blur until at most two passes remain. Inside that loop we can alternate between using the temporary texture and the original color texture as the render target.

```
		cb.GetTemporaryRT(tempTexId, width, height, 0, FilterMode.Bilinear);
		int passesLeft;
		for (passesLeft = blurStrength; passesLeft > 2; passesLeft -= 2) {
			Blit(cb, cameraColorId, tempTexId, Pass.Blur);
			Blit(cb, tempTexId, cameraColorId, Pass.Blur);
		}
		if (passesLeft > 1) {
			Blit(cb, cameraColorId, tempTexId, Pass.Blur);
			Blit(cb, tempTexId, BuiltinRenderTextureType.CameraTarget, Pass.Blur);
		}
```

And in the special case of only blurring once we can avoid getting the temporary texture.

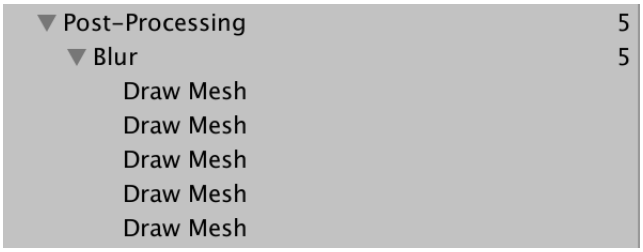
```
		if (blurStrength == 1) {
			Blit(
				cb, cameraColorId, BuiltinRenderTextureType.CameraTarget,Pass.Blur
			);
			return;
		}
		cb.GetTemporaryRT(tempTexId, width, height, 0, FilterMode.Bilinear);
```



Blur strength 5.

Let's wrap up our blur effect by grouping all its draw calls under a *Blur* entry in the frame debugger, by beginning and ending a nested sample in the `Blur` method.

```
	void Blur (CommandBuffer cb, int cameraColorId, int width, int height) {
		cb.BeginSample("Blur");
		if (blurStrength == 1) {
			Blit(
				cb, cameraColorId, BuiltinRenderTextureType.CameraTarget,Pass.Blur
			);
			cb.EndSample("Blur");
			return;
		}
		…
		cb.EndSample("Blur");
	}
```



Blurring in the frame debugger.

5 Using the Depth Buffer

As mentioned earlier, some post-processing effects depend on the depth buffer. We'll provide an example of how to do this by adding an effect that draws lines to indicate the depth.

5.1 Depth Stripes

Add a fragment function to the HLSL file for drawing depth stripes. Begin by sampling the depth, which we'll make available via *_MainTex*. We can use the `SAMPLE_DEPTH_TEXTURE` macro to make it work for all platforms.

```
float4 DepthStripesPassFragment (VertexOutput input) : SV_TARGET {
	return SAMPLE_DEPTH_TEXTURE(_MainTex, sampler_MainTex, input.uv);
}
```

We need the world-space depth—which is the distance from the near place, not the camera's position—which we can find via the `LinearEyeDepth` function. Besides the raw depth it also needs *_ZBufferParams*, which is another vector set by `SetupCameraProperties`.

```
float4 _ZBufferParams;

…

float4 DepthStripesPassFragment (VertexOutput input) : SV_TARGET {
	float rawDepth = SAMPLE_DEPTH_TEXTURE(_MainTex, sampler_MainTex, input.uv);
	return LinearEyeDepth(rawDepth, _ZBufferParams);
}
```

The simplest way to draw smooth stripes based on the depth d is to use $\sin^2 \pi d$. The result isn't beautiful but suffices to illustrate that the depth information is used.

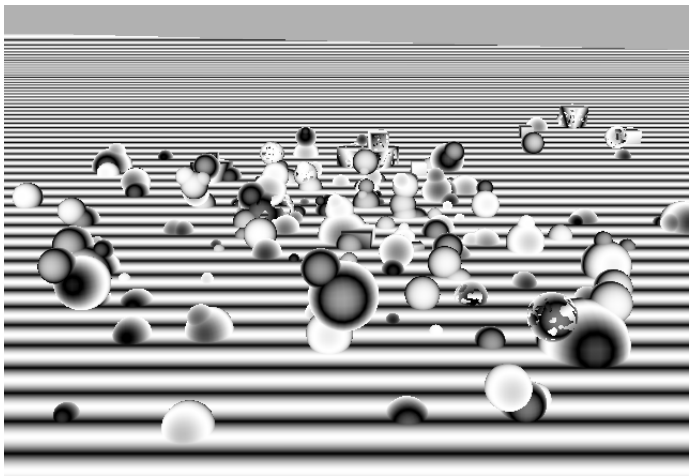
```
float4 DepthStripesPassFragment (VertexOutput input) : SV_TARGET {
	float rawDepth = SAMPLE_DEPTH_TEXTURE(_MainTex, sampler_MainTex, input.uv);
	float depth = LinearEyeDepth(rawDepth, _ZBufferParams);
	return pow(sin(3.14 * depth), 2.0);
}
```

Add a pass for the depth stripes to the shader.

```
		Pass { // 2 DepthStripes
			HLSLPROGRAM
			#pragma target 3.5
			#pragma vertex DefaultPassVertex
			#pragma fragment DepthStripesPassFragment
			ENDHLSL
		}
```

Also add the pass to the enum in `MyPostProcessingStack` and then blit from depth to color with it in `Render`. Do this before blurring, but set the blur strength to zero to disable it.

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

	…

	public void Render (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		InitializeStatic();
		Blit(cb, cameraDepthId, cameraColorId, Pass.DepthStripes);

		…
	}
```



Depth stripes.

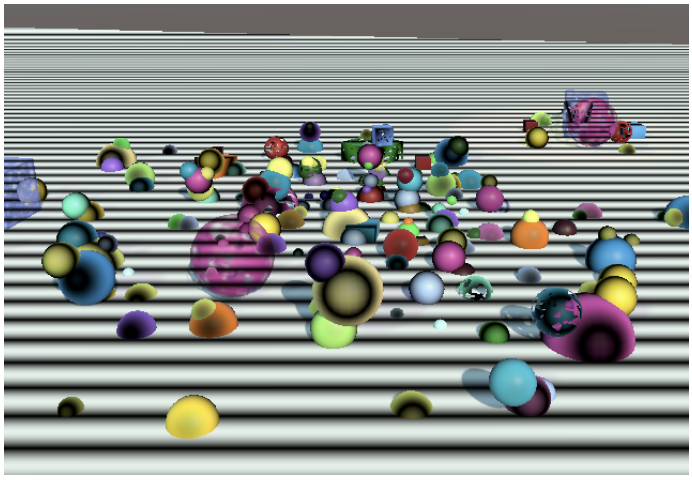
5.2 Mixing Depth and Color

Instead of completely replacing the original image we can factor the striped into it. This requires us to use two source textures. We could directly use `_CameraDepthTexture`, but let's keep the stack unaware of how exactly the pipeline renders depth and instead bind it to *_DepthTex* to accompany *_MainTex*. Also, to keep blurring working we have to render to the color texture, which requires a temporary texture and an extra copy. Put all that code in a separate `DepthStripes` method that groups the draws under *Depth Stripes*.

```
	static int depthTexId = Shader.PropertyToID("_DepthTex");

	…

	public void Render (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		InitializeStatic();
		//Blit(cb, depthTextureId, colorTextureId, Pass.DepthStripes);
		DepthStripes(cb, cameraColorId, cameraDepthId, width, height);

		…
	}

	…

	void DepthStripes (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		cb.BeginSample("Depth Stripes");
		cb.GetTemporaryRT(tempTexId, width, height);
		cb.SetGlobalTexture(depthTexId, cameraDepthId);
		Blit(cb, cameraColorId, tempTexId, Pass.DepthStripes);
		Blit(cb, tempTexId, cameraColorId);
		cb.ReleaseTemporaryRT(tempTexId);
		cb.EndSample("Depth Stripes");
	}
```

Then adjust `DepthStripesPassFragment` so it samples both the color texture and depth texture and multiplies the color with the striples.

```
TEXTURE2D(_DepthTex);
SAMPLER(sampler_DepthTex);

…

float4 DepthStripesPassFragment (VertexOutput input) : SV_TARGET {
	float rawDepth = SAMPLE_DEPTH_TEXTURE(_DepthTex, sampler_DepthTex, input.uv);
	float depth = LinearEyeDepth(rawDepth, _ZBufferParams);
	float4 color = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, input.uv);
	return color * pow(sin(3.14 * depth), 2.0);
}
```



Colored depth stripes.

5.3 Skipping the Sky Box

The stripes get applied to everything, also the sky box. But the sky box doesn't render to the depth buffer, which means that it ends up with the greatest possible depth value. However the results are unstable and if a lot of the sky is visible a big portion of the window can flicker terribly during camera movement. It is best to not to modify the sky. The default raw depth value is either zero or one, depending on whether the depth buffer is reversed, which is the case for non-OpenGL platforms. If so `UNITY_REVERSED_Z` is defined, which we can use to check whether the fragment has a valid depth. If not, return the original color.

```
	#if UNITY_REVERSED_Z
		bool hasDepth = rawDepth != 0;
	#else
		bool hasDepth = rawDepth != 1;
	#endif
	if (hasDepth) {
		color *= pow(sin(3.14 * depth), 2.0);
	}
	return color;
```

5.4 Opaque-Only Post-Processing

Besides the sky box, transparent geometry also doesn't write to the depth buffer. Thus stripes get applied on top of transparent surfaces based on what's behind them. Effects like depth-of-field behave in the same way. For some effects it's better that they aren't applied to transparent object at all. That can be accomplished by rendering them before transparent geometry, making them post-opaque pre-transparent effects.

We can make the depth stripes affect only opaque geometry by splitting `MyPostProcessingStack.Render` in two methods: `RenderAfterOpaque` and `RenderAfterTransparent`. The first initializes and does the stripes while the latter does the blur.

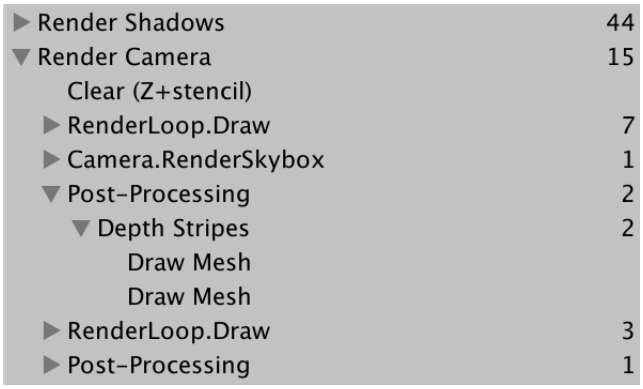
```
	public void RenderAfterOpaque (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		InitializeStatic();
		DepthStripes(cb, cameraColorId, cameraDepthId, width, height);
	}

	public void RenderAfterTransparent (
		CommandBuffer cb, int cameraColorId, int cameraDepthId,
		int width, int height
	) {
		//InitializeStatic();
		//DepthStripes(cb, cameraColorId, cameraDepthId, width, height);
		if (blurStrength > 0) {
			Blur(cb, cameraColorId, width, height);
		}
		else {
			Blit(cb, cameraColorId, BuiltinRenderTextureType.CameraTarget);
		}
	}
```

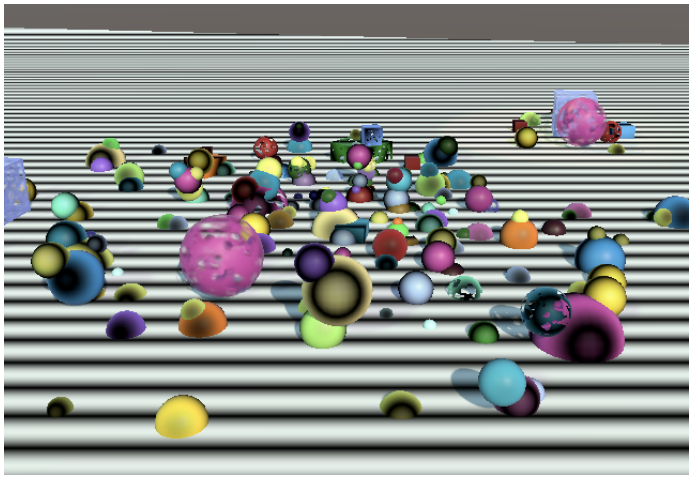
`MyPipeline.Render` now also has to invoke the stack directly after drawing the sky box, using the appropriate method.

```
		context.DrawSkybox(camera);

		if (defaultStack) {
			defaultStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				camera.pixelWidth, camera.pixelHeight
			);
			context.ExecuteCommandBuffer(postProcessingBuffer);
			postProcessingBuffer.Clear();
		}

		drawSettings.sorting.flags = SortFlags.CommonTransparent;
		filterSettings.renderQueueRange = RenderQueueRange.transparent;
		context.DrawRenderers(
			cull.visibleRenderers, ref drawSettings, filterSettings
		);

		DrawDefaultPipeline(context, camera);

		if (defaultStack) {
			defaultStack.RenderAfterTransparent(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				camera.pixelWidth, camera.pixelHeight
			);
			…
		}
```

We also have to make sure that the render target is set up correctly after rendering the opaque post-processing effects. We have to set the color and depth targets again, and this time we do care that they are loaded.

```
		if (activeStack) {
			activeStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				camera.pixelWidth, camera.pixelHeight
			);
			context.ExecuteCommandBuffer(postProcessingBuffer);
			postProcessingBuffer.Clear();
			cameraBuffer.SetRenderTarget(
				cameraColorTextureId,
				RenderBufferLoadAction.Load, RenderBufferStoreAction.Store,
				cameraDepthTextureId,
				RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
			);
			context.ExecuteCommandBuffer(cameraBuffer);
			cameraBuffer.Clear();
		}
```





Drawing depth stripes after opaque geometry.

5.5 Optional Stripes

Because the depth stripes are just a test, let's make them optional by adding a toggle to `MyPostProcessingStack`.

```
	[SerializeField]
	bool depthStripes;

	…

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

Depth stripes enabled.

6 Post-Processing Per Camera

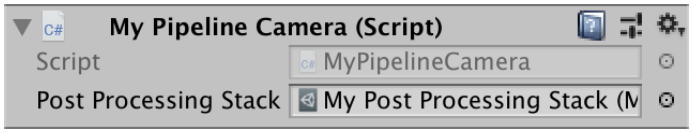
Currently the only way to enable post-processing is to configure a default stack, which get applied to all cameras. This includes not only the main camera and scene camera, but also cameras used to render reflection probes and any other cameras you might use. So the default stack is only appropriate for effects that should be applied to all those cameras. Typically most post-processing effects are applied to the main camera only. Also, there might be multiple cameras that each need different effects. So let's make it possible to select a stack per camera.

6.1 Camera Configuration

We cannot add configuration options to the existing `Camera` component. What we can do instead is create a new component type that contains the extra options. Name it `MyPipelineCamera`, have it require that it's attached to a game object that has a `Camera` component, and add a configurable post-processing stack field. Also add a public getter property to retrieve the stack.

```
using UnityEngine;

[RequireComponent(typeof(Camera))]
public class MyPipelineCamera : MonoBehaviour {

	[SerializeField]
	MyPostProcessingStack postProcessingStack = null;

	public MyPostProcessingStack PostProcessingStack {
		get {
			return postProcessingStack;
		}
	}
}
```

Attach this component to the main camera and assign our stack to it. The default stack of the pipeline asset can then be set to none.



Extra camera component with stack.

To make this work `MyPipeline.Render` now has to get the `MyPipelineCamera` component from the camera used for rendering. If the component exists, use its stack as the active stack instead of the default.

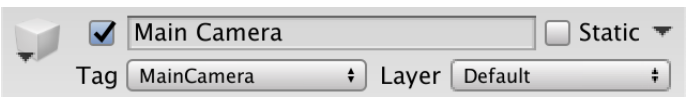
```
		var myPipelineCamera = camera.GetComponent<MyPipelineCamera>();
		MyPostProcessingStack activeStack = myPipelineCamera ?
			myPipelineCamera.PostProcessingStack : defaultStack;

		if (activeStack) {
			cameraBuffer.GetTemporaryRT(
				cameraColorTextureId, camera.pixelWidth, camera.pixelHeight, 0,
				FilterMode.Bilinear
			);
			…
		}

		…

		if (activeStack) {
			activeStack.RenderAfterOpaque(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				camera.pixelWidth, camera.pixelHeight
			);
			…
		}

		…

		if (activeStack) {
			activeStack.RenderAfterTransparent(
				postProcessingBuffer, cameraColorTextureId, cameraDepthTextureId,
				camera.pixelWidth, camera.pixelHeight
			);
			…
		}
```

6.2 Scene Camera

We can now select a post-processing stack for each camera in the scene, but we cannot directly control the camera used to render the scene window. What we can do instead is attach the `ImageEffectAllowedInSceneView` attribute to `MyPipelineCamera`.

```
[ImageEffectAllowedInSceneView, RequireComponent(typeof(Camera))]
public class MyPipelineCamera : MonoBehaviour { … }
```

Despite the attribute's name, it doesn't apply to image effects specifically. Unity will simply copy all components of the active main camera that have this attribute to the scene camera. So to make this work the camera must have the *MainCamera* tag.



Camera tagged as main.

The next tutorial is Image Quality.

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick