updated 2023-08-03 published 2020-11-27

Particles Color and Depth Textures

Support flipbook, near fade, soft, and distortion particles.
Determine fragment depth, for orthographic and perspective projections.
Copy and sample the color and depth buffers.

This is the 15th part of a tutorial series about creating a custom scriptable render pipeline. We'll create depth-based fading and distorting particles, relying on a color and depth texture.

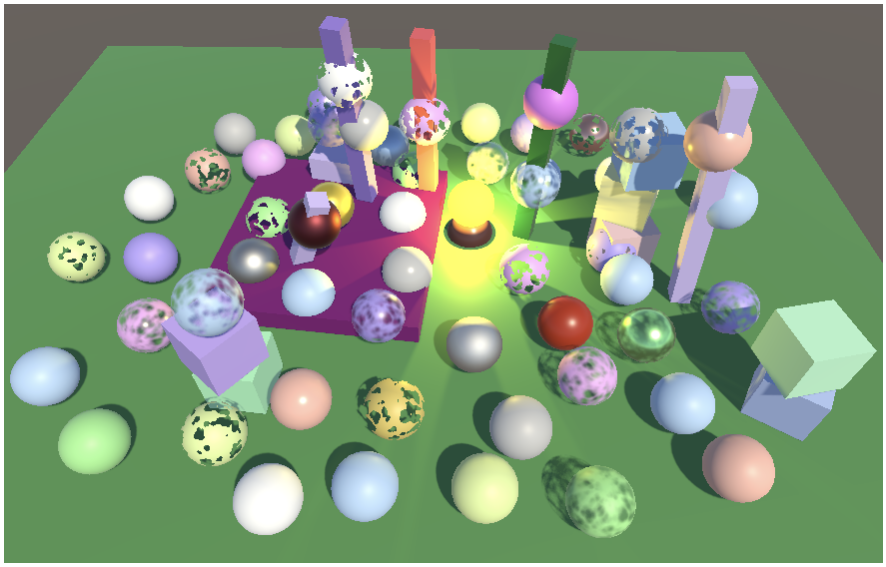
This tutorial is made with Unity 2019.4.14f1 and upgraded to 2022.3.5f1.

Using particles to create a messy atmosphere.

1 Unlit Particles

A particle system can use any material, so our RP can already render them, with limitations. In this tutorial we'll only consider unlit particles. Lit particles work the same way, just with more shader properties and lighting calculations.

I set up a new scene for the particles that is a variant of the already existing test scene. It has a few long vertical cubes and a bright yellow lightbulb to serve as a background for the particles systems.



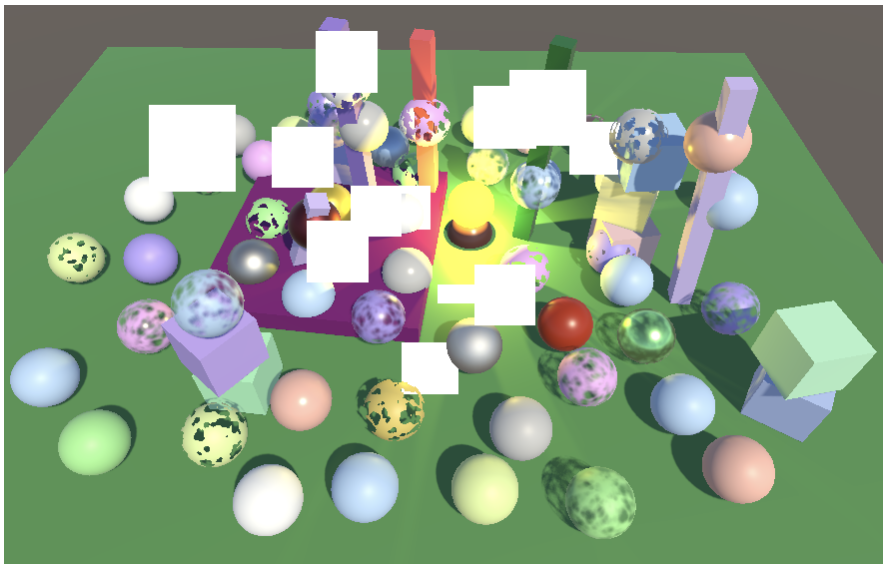
Scene without particles and without post FX.

1.1 Particle System

Create a particle system via *GameObject / Effects / Particle System* and position it a bit below the ground plane. I assume that you already know how to configure particle systems and won't go into details about that. If not check Unity's documentation to learn about the specific modules and their settings.

> **What about the Visual Effects Graph?**
>
> The VFX graph is compute-shader based and currently tightly coupled with URP and HDRP. It cannot be easily used with a custom SRP.
>
> Note that the regular particle system is not superseded by the VFX graph. It's better for many small systems—up to about a thousand particles each—while VFX graph is better for massive systems.

The default system makes particles move upward and fill a cone-shaped region. If we assign our unlit material to it the particles will show up as solid white squares aligned with the camera plane. They pop in and out of existence but because they start below plane they appear to rise out of the ground.



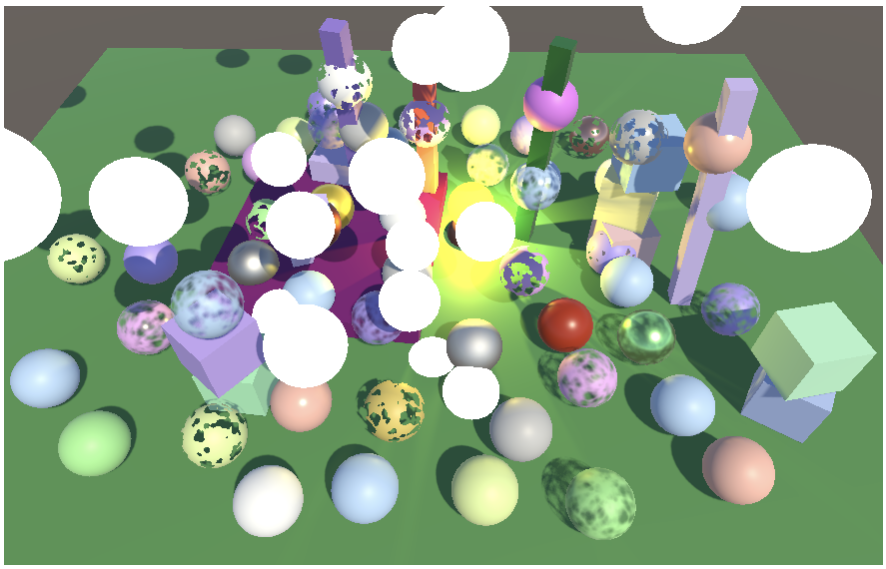
Default particle system with unlit material, positioned below ground.

1.2 Unlit Particles Shader

We could use our unlit shader for particles, but let's make a dedicated one for them. It starts as a copy of the unlit shader with its menu item changed to *Custom RP/Particles/Unlit*. Also, as particles are always dynamic it doesn't need a meta pass.

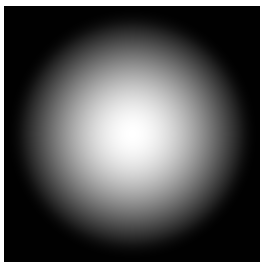
```
Shader "Custom RP/Particles/Unlit" {

	…

	SubShader {
		…

		//Pass {
			//Tags {
				//"LightMode" = "Meta"
			//}

			//…
		//}
	}

	CustomEditor "CustomShaderGUI"
}
```

Create a dedicated material for unlit particles with this shader, then make the particle system use it. Currently it is equivalent to the earlier unlit material. It's also possible to set the particle system to render meshes, even with shadows if that's enabled for both the material and the particle system. However, GPU instancing doesn't work because particles systems use procedural drawing for that, which we won't cover in this tutorial. Instead all particle meshes get merged into a single mesh, just like billboard particles.



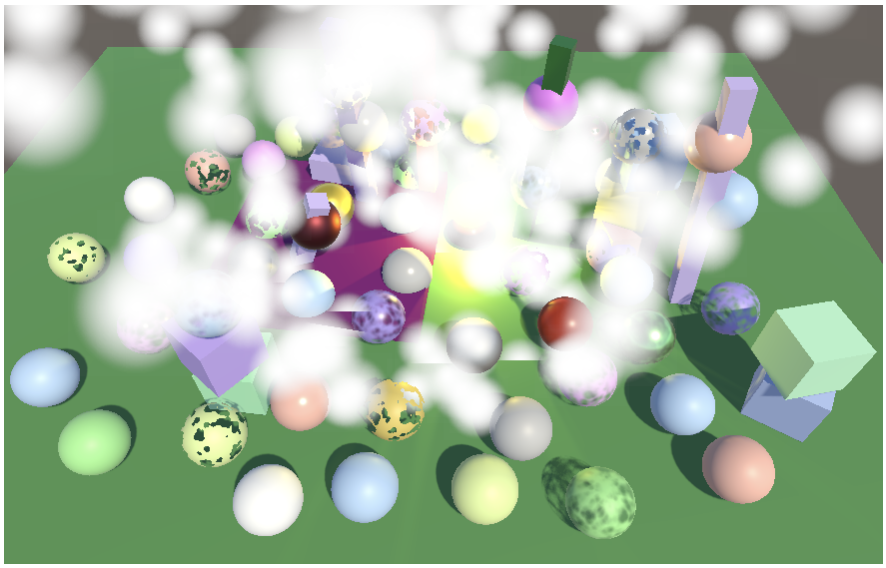
Sphere mesh particles, with shadows.

From now on we'll only concern ourselves with billboard particles, without shadows. Here is a base map for a single particle, containing a simple smoothly fading white disc.



Base map for single particle, on black background.

When using that texture for our fade particles we get a simplistic effect that looks somewhat like white smoke is coming out of the ground. To make it more convincing increase the emission rate to something like 100.



Textured billboard particles, emission rate set to 100.

1.3 Vertex Colors

It's possible to use a different color per particle. The simplest way to demonstrate this is to set the starting color to randomly pick between black and white. However, doing so doesn't currently change the appearance of the particles. To make this work we have to add support for vertex colors to our shader. Rather than create new HLSL files for particles we'll add support for it to *UnlitPass*.

The first step is to add a `float4` vertex attribute with the `COLOR` semantic.

```
struct Attributes {
	float3 positionOS : POSITION;
	float4 color : COLOR;
	float2 baseUV : TEXCOORD0;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Add it to `Varyings` as well and pass it through `UnlitPassVertex`, but only if *_VERTEX_COLORS* is defined. That way we can enable and disable vertex color support as desired.

```
struct Varyings {
	float4 positionCS : SV_POSITION;
	#if defined(_VERTEX_COLORS)
		float4 color : VAR_COLOR;
	#endif
	float2 baseUV : VAR_BASE_UV;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

Varyings UnlitPassVertex (Attributes input) {
	…
	#if defined(_VERTEX_COLORS)
		output.color = input.color;
	#endif
	output.baseUV = TransformBaseUV(input.baseUV);
	return output;
}
```

Next, add a color to `InputConfig` in *UnlitInput*, set it to opaque white by default, and factor it into the result of `GetBase`.

```
struct InputConfig {
	float4 color;
	float2 baseUV;
};

InputConfig GetInputConfig (float2 baseUV) {
	InputConfig c;
	c.color = 1.0;
	c.baseUV = baseUV;
	return c;
}

…

float4 GetBase (InputConfig c) {
	float4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, c.baseUV);
	float4 baseColor = INPUT_PROP(_BaseColor);
	return baseMap * baseColor * c.color;
}
```

Back to *UnlitPass*, copy the interpolated vertex color to `config` if it exists in `UnlitPassFragment`.

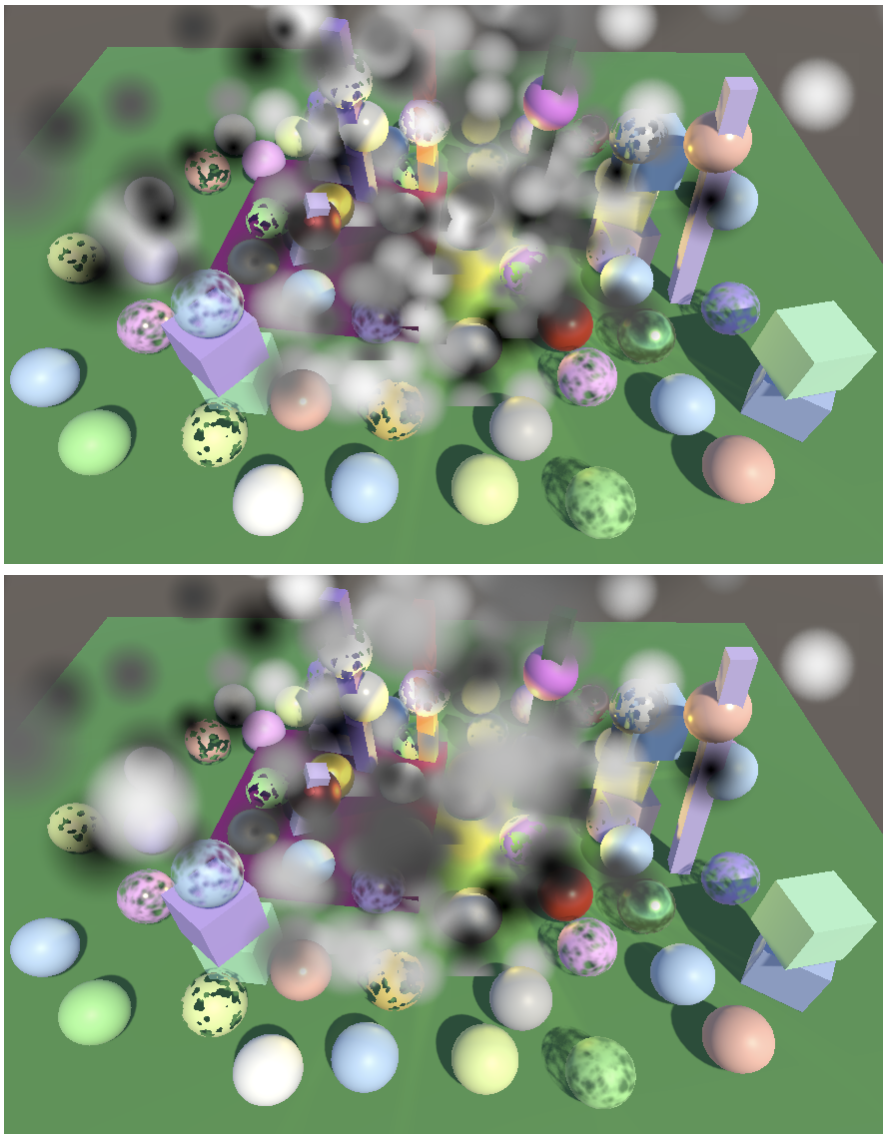
```
	InputConfig config = GetInputConfig(input.baseUV);
	#if defined(_VERTEX_COLORS)
		config.color = input.color;
	#endif
```

To finally add support for vertex colors to *UnlitParticles* add a toggle shader property to it.

```
		[HDR] _BaseColor("Color", Color) = (1.0, 1.0, 1.0, 1.0)
		[Toggle(_VERTEX_COLORS)] _VertexColors ("Vertex Colors", Float) = 0
```

Along with the corresponding shader feature that defined the keyword. You can do this for the regular *Unlit* shader too if you want it to support vertex colors as well.

```
		#pragma shader_feature _VERTEX_COLORS
```



Using vertex colors, without and with sorting by distance.

We now get colored particles. At this point particle sorting becomes an issue. If all particles have the same color their draw order doesn't matter, but if they're different we need to sort them by distance to get the correct result. Note that when sorting based on distance particles might suddenly swap draw order due to position of view changes, like any transparent object.

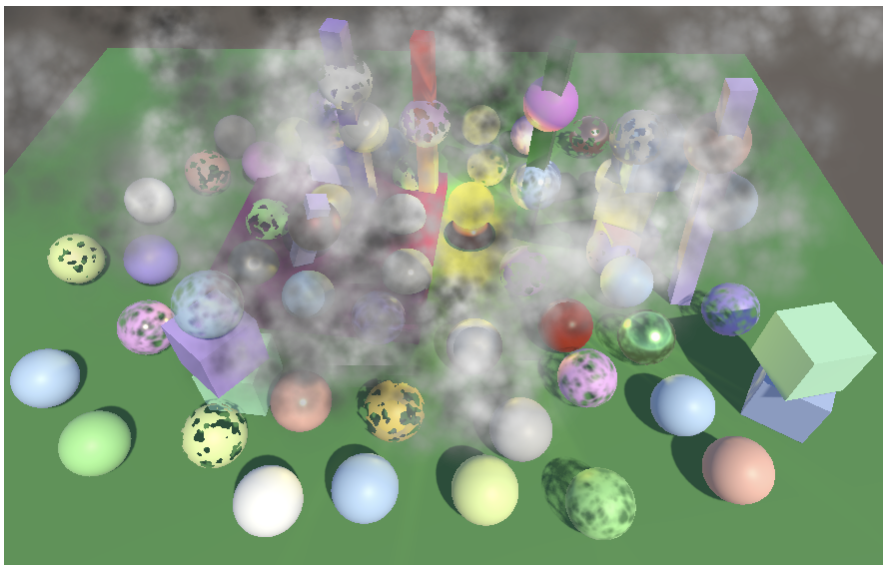
1.4 Flipbooks

Billboard particles can be animated, by cycling through different base maps. Unity refers to these as flipbook particles. This is done by using a texture atlas laid out in a regular grid, like this texture containing a 4×4 grid of a looping noise pattern.

Base map for particle flipbook, on black background.

Create a new unlit particle material that uses the flipbook map, then duplicate our particle system and have it use that flipbook material. Deactivate the singular-particle version so we only see the flipbook system. As each particle now represents a little cloud increase their size to something like 2. Enable the *Texture Sheet Animation* module of the particle system, configure it for a 4×4 flipbook, make it start at a random frame, and go through one cycle during a particle's lifetime.

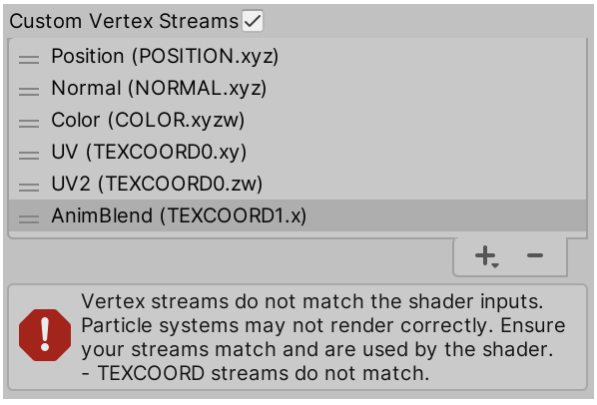
Extra variety can be added by randomly flipping particles along X and Y 50% of the time, starting with an arbitrary rotation, and making particles rotate with a random velocity.



Flipbook particle system.

1.5 Flipbook Blending

When the system is in motion it is obvious that the particles cycle through a few frames, because the flipbook frame rate is very low. For particles with a lifetime of five seconds it's 3.2 frames per second. This can be smoothed out by blending between successive frames. This requires us to send a second pair of UV coordinates and an animation blend factor to the shader. We do that by enabling custom vertex stream in the *Renderer* module. Add *UV2* and *AnimBlend*. You could also remove the normal stream, as we don't need it.



Custom vertex streams.

After adding the streams an error will be displayed indicating a mismatch between the particle system and the shader that it currently uses. This error will go away after we consume these streams in our shader. Add a shader keyword toggle property to *UnlitParticle* to control whether we support flipbook blending or not.

```
		[Toggle(_VERTEX_COLORS)] _VertexColors ("Vertex Colors", Float) = 0
		[Toggle(_FLIPBOOK_BLENDING)] _FlipbookBlending ("Flipbook Blending", Float) = 0
```

Along with the accompanying shader feature.

```
			#pragma shader_feature _FLIPBOOK_BLENDING
```

If flipbook blending is active both UV pairs are provided via `TEXCOORD0`, so it has to be a `float4` instead of a `float2`. The blend factor is provided as a single `float` via `TEXCOORD1`.

```
struct Attributes {
	float3 positionOS : POSITION;
	float4 color : COLOR;
	#if defined(_FLIPBOOK_BLENDING)
		float4 baseUV : TEXCOORD0;
		float flipbookBlend : TEXCOORD1;
	#else
		float2 baseUV : TEXCOORD0;
	#endif
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

We'll add the new data as a single `float3 flipbookUVB` field to `Varyings`, if needed.

```
struct Varyings {
	…
	float2 baseUV : VAR_BASE_UV;
	#if defined(_FLIPBOOK_BLENDING)
		float3 flipbookUVB : VAR_FLIPBOOK;
	#endif
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Adjust `UnlitPassVertex` so it copies all relevant data to it, when appropriate.

```
Varyings UnlitPassVertex (Attributes input) {
	…
	output.baseUV.xy = TransformBaseUV(input.baseUV.xy);
	#if defined(_FLIPBOOK_BLENDING)
		output.flipbookUVB.xy = TransformBaseUV(input.baseUV.zw);
		output.flipbookUVB.z = input.flipbookBlend;
	#endif
	return output;
}
```

Add `flipbookUVB` to `InputConfig` as well, along with a boolean to indicate whether flipbook blending is enabled, which isn't the case by default.

```
struct InputConfig {
	float4 color;
	float2 baseUV;
	float3 flipbookUVB;
	bool flipbookBlending;
};

InputConfig GetInputConfig (float2 baseUV) {
	…
	c.flipbookUVB = 0.0;
	c.flipbookBlending = false;
	return c;
}
```

If flipbook blending is enabled we have to sample the base map a second time in `GetBase`, with the flipbook UV, then interpolate from the first to second sample based on the blend factor.

```
float4 GetBase (InputConfig c) {
	float4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, c.baseUV);
	if (c.flipbookBlending) {
		baseMap = lerp(
			baseMap, SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, c.flipbookUVB.xy),
			c.flipbookUVB.z
		);
	}
	float4 baseColor = INPUT_PROP(_BaseColor);
	return baseMap * baseColor * c.color;
}
```

To finally activate flipbook blending override the default configuration in `UnlitPassFragment` when appropriate.

```
	#if defined(_VERTEX_COLORS)
		config.color = input.color;
	#endif
	#if defined(_FLIPBOOK_BLENDING)
		config.flipbookUVB = input.flipbookUVB;
		config.flipbookBlending = true;
	#endif
```

Flipbook blending.

2 Fading Near Camera

When the camera is inside a particle system particles will end up very close to the camera's near place and also pass from one side to the other. The particle system has a *Renderer / Max Particle Size* property that prevents individual billboard particles from covering too much of the window. Once they reach their maximum visible size they'll appear to slide out of the way instead of growing larger as they approach the near plane.

Another way to deal with particles close to the near plane is to fade them out based on their fragment depth. This can look better when moving through a particle system that represents atmospherical effects.

2.1 Fragment Data

We already have the fragment depth available in out fragment functions. It's provided via the `float4` with the `SV_POSITION` semantic. We've already used the XY components of it for dithering, but let's now make it formal that we're using fragment data.

In the vertex function `SV_POSITION` represents the clip-space position of the vertex, as 4D homogeneous coordinates. But in the fragment function `SV_POSITION` represents the screen-space—also known as window-space—position of the fragment. The space conversion is performed by the GPU. To make this explicit let's rename `postionCS` to `positionCS_SS` in all our `Varyings` structs.

```
    float4 positionCS_SS : SV_POSITION;
```

Make the adjustment in the accompanying vertex functions as well.

```
    output.positionCS_SS = TransformWorldToHClip(positionWS);
```

Next, we'll introduce a new *Fragment* HLSL include file containing a `Fragment` struct and a `GetFragment` function that returns the fragment, given a `float4` screen-space position vector. Initially the fragment only has a 2D position, which comes from the screen-space position's XY components. These are texel coordinates with a 0.5 offset. It's (0.5, 0.5) for the texel in the bottom left corner of the screen, (1.5, 0.5) for the texel to the right of it, and so on.

```
#ifndef FRAGMENT_INCLUDED
#define FRAGMENT_INCLUDED

struct Fragment {
	float2 positionSS;
};

Fragment GetFragment (float4 positionSS) {
	Fragment f;
	f.positionSS = positionSS.xy;
	return f;
}

#endif
```

Include this file in *Common* after all other include statements, then adjust `ClipLOD` so its first argument is a `Fragment` instead of a `float4`.

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

#include "Fragment.hlsl"

…

void ClipLOD (Fragment fragment, float fade) {
	#if defined(LOD_FADE_CROSSFADE)
		float dither = InterleavedGradientNoise(fragment.positionSS, 0);
		clip(fade + (fade < 0.0 ? dither : -dither));
	#endif
}
```

Let's also define common linear and point clamp sampler states in *Common* as well at this point, because we'll be using those in multiple places later. Do so before including *Fragment*.

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

SAMPLER(sampler_linear_clamp);
SAMPLER(sampler_point_clamp);

#include "Fragment.hlsl"
```

Then remove the generic sampler definition from *PostFXStackPasses*, as that's now a duplicate that would cause a compiler error.

```
TEXTURE2D(_PostFXSource2);
//SAMPLER(sampler_linear_clamp);
```

Next, add a fragment to the `InputConfig` structs of both *LitInput* and *UnlitInput*. Then add the screen-space position vector as a first parameter to the `GetInputConfig` functions, so they can invoke `GetFragment` with it.

```
struct InputConfig {
	Fragment fragment;
	…
};

InputConfig GetInputConfig (float4 positionSS, …) {
	InputConfig c;
	c.fragment = GetFragment(positionSS);
	…
}
```

Add the argument in all places that we invoke `GetInputConfig`.

```
	InputConfig config = GetInputConfig(input.positionCS_SS, …);
```

Then adjust `LitPassFragment` so it invokes `ClipLOD` after getting the config, so it can pass a fragment to it. Also pass the fragment's position to `InterleavedGradientNoise` instead of using `input.positionCS_SS` directly.

```
float4 LitPassFragment (Varyings input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);
	//ClipLOD(input.positionSS.xy, unity_LODFade.x);
	InputConfig config = GetInputConfig(input.positionCS_SS, input.baseUV);
	ClipLOD(config.fragment, unity_LODFade.x);

	…
	surface.dither = InterleavedGradientNoise(config.fragment.positionSS, 0);
	…
}
```

`ShadowCasterPassFragment` must also be changed so it clips after getting the config.

```
void ShadowCasterPassFragment (Varyings input) {
	UNITY_SETUP_INSTANCE_ID(input);
	//ClipLOD(input.positionCS.xy, unity_LODFade.x);
	InputConfig config = GetInputConfig(input.positionCS_SS, input.baseUV);
	ClipLOD(config.fragment, unity_LODFade.x);

	float4 base = GetBase(config);
	#if defined(_SHADOWS_CLIP)
		clip(base.a - GetCutoff(config));
	#elif defined(_SHADOWS_DITHER)
		float dither = InterleavedGradientNoise(input.positionSS.xy, 0);
		clip(base.a - dither);
	#endif
}
```

2.2 Fragment Depth

To fade particles near the camera we need to know the fragment's depth. So add a depth field to `Fragment`.

```
struct Fragment {
	float2 positionSS;
	float depth;
};
```

The fragment depth is stored in the last component of the screen-space position vector. It's the value that was used to perform the perspective division to project 3D positions onto the screen. This is the view-space depth, so it's the distance from the camera XY plane, not its near plane.

```
Fragment GetFragment (float4 positionCS_SS) {
	Fragment f;
	f.positionSS = positionSS.xy;
	f.depth = positionSS.w;
	return f;
}
```

What is view space?

It is world space rotated and translated so the camera ends up without rotation at the origin.

We can verify that this is correct by directly returning the fragment depth in `LitPassFragment` and `UnlitPassFragment`, scaled down so we can see it as a grayscale gradient.

```
	InputConfig config = GetInputConfig(input.positionCS_SS, input.baseUV);
	return float4(config.fragment.depth.xxx / 20.0, 1.0);
```

Fragment depth, divided by 20.

2.3 Orthographic Depth

The above approach only works when using a perspective camera. When using an orthographic camera there is no perspective division, thus the last component of the screen-space position vector is always 1.

We can determine whether we're dealing with an orthographic camera by adding a `float4 unity_OrthoParams` field to *UnityInput*, via which Unity communicated information about the orthographic camera to the GPU.

```
float4 unity_OrthoParams;
float4 _ProjectionParams;
```

In case of an orthographic camera its last component will be 1, otherwise it will be zero. Add an `IsOrthographicCamera` function to *Common* that uses this fact, defined before including *Fragment* so we can use it there. If you'll never use orthographic cameras you could hard-code it to return `false`, or control this via a shader keyword.

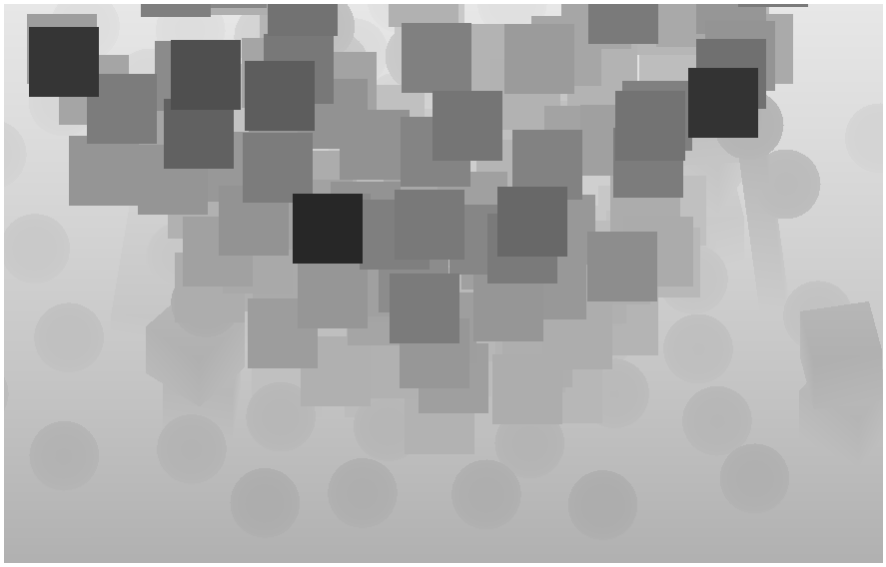
```
bool IsOrthographicCamera () {
	return unity_OrthoParams.w;
}

#include "Fragment.hlsl"
```

For an orthographic camera the best we can do is rely on the Z component of the screen-space position vector, which contains the converted clip-space depth of the fragment. This is the raw value that is used for depth comparisons and is written to the depth buffer if depth writing is enabled. It's a value in the 0–1 range and is linear for orthographic projections. To convert it to view-space depth we have to scale it by the camera's near-far range and then add the near plane distance. The near and far distances are stored in the Y and Z components of `_ProjectionParams`. We also need to reverse the raw depth if a reversed depth buffer is used. Do this in a new `OrthographicDepthBufferToLinear` function, also defined in *Common* before including *Fragment*.

```
float OrthographicDepthBufferToLinear (float rawDepth) {
	#if UNITY_REVERSED_Z
		rawDepth = 1.0 - rawDepth;
	#endif
	return (_ProjectionParams.z - _ProjectionParams.y) * rawDepth + _ProjectionParams.y;
}

#include "Fragment.hlsl"
```

Now `GetFragment` can check whether an orthographic camera is used and if so rely on `OrthographicDepthBufferToLinear` to determine the fragment depth.

```
	f.depth = IsOrthographicCamera() ?
		OrthographicDepthBufferToLinear(positionSS.z) : positionSS.w;
```



Fragment depth for orthographic camera.

After verifying that the fragment depth is correct for both camera types remove the debug visualization from `LitPassFragment` and `UnlitPassFragment`.

```
	//return float4(config.fragment.depth.xxx / 20.0, 1.0);
```

2.4 Distance-Based Fading

Back to the *UnlitParticles* shader, add a *Near Fade* keyword toggle property, along with properties to make its distance and range configurable. The distance determines how close to the camera plane the particles should disappear completely. This is the camera plane, not its near place. So a value of at least the near plane should be used. 1 is a reasonable default. The range controls the length of the transition region, inside which the particles will linearly fade out. Again 1 is a reasonable default, and at minimum it must be a small positive value.

```
		[Toggle(_NEAR_FADE)] _NearFade ("Near Fade", Float) = 0
		_NearFadeDistance ("Near Fade Distance", Range(0.0, 10.0)) = 1
		_NearFadeRange ("Near Fade Range", Range(0.01, 10.0)) = 1
```

Add a shader feature to enable the near fading.

```
			#pragma shader_feature _NEAR_FADE
```

Then include the distance and range in the `UnityPerMaterial` buffer in *UnlitInput*.

```
UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
	UNITY_DEFINE_INSTANCED_PROP(float4, _BaseMap_ST)
	UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
	UNITY_DEFINE_INSTANCED_PROP(float, _NearFadeDistance)
	UNITY_DEFINE_INSTANCED_PROP(float, _NearFadeRange)
	UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
	UNITY_DEFINE_INSTANCED_PROP(float, _ZWrite)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

Next, add a boolean `nearFade` field to `InputConfig` to control whether near fading is active, which it isn't by default.

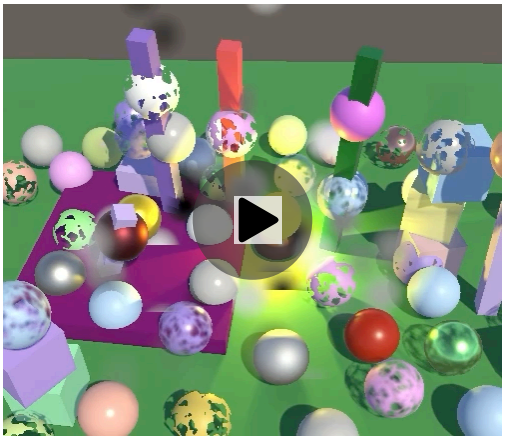
```
struct InputConfig {
	…
	bool nearFade;
};

InputConfig GetInputConfig (float4 positionCC_SS, float2 baseUV) {
	…
	c.nearFade = false;
	return c;
}
```

Fading near the camera is done by simply decreasing the fragment's base alpha. The attenuation factor is equal to the fragment depth minus the fade distance, then divided by the fade range. As the result can be negative saturate it before factoring it into the alpha of the base map. Do this in `GetBase`, when appropriate.

```
	if (c.flipbookBlending) { … }
	if (c.nearFade) {
		float nearAttenuation = (c.fragment.depth - INPUT_PROP(_NearFadeDistance)) /
			INPUT_PROP(_NearFadeRange);
		baseMap.a *= saturate(nearAttenuation);
	}
```

Finally, to activate the feature set the fragment's `nearFade` field to `true` in `UnlitPassFragment` is the *_NEAR_FADE* keyword if defined.

```
	#if defined(_FLIPBOOK_BLENDING)
		config.flipbookUVB = input.flipbookUVB;
		config.flipbookBlending = true;
	#endif
	#if defined(_NEAR_FADE)
		config.nearFade = true;
	#endif
```



Adjusting near fade distance.

3 Soft Particles

When billboard particles intersect geometry the sharp transition is both visually jarring and makes their flat nature obvious. The solution for this is to use soft particles, which fade out when there's opaque geometry close behind them. To make this work the particle's fragment depth has to be compared to the depth of whatever has been drawn earlier to the same position in the camera's buffer. This means that we'll have to sample the depth buffer.

3.1 Separate Depth Buffer

Up to this point we've always used a single frame buffer for the camera, which contained both color and depth information. This is the typical frame buffer configuration, but the color and depth data are always stored in separate buffers, known as frame buffer attachments. To access the depth buffer we'll need to define these attachments separately.

The first step is to replace the *_CameraFrameBuffer* identifier in `CameraRenderer` with two identifiers, which we'll name *_CameraColorAttachment* and *_CameraDepthAttachment*.

```
	//static int frameBufferId = Shader.PropertyToID("_CameraFrameBuffer");
	static int
		colorAttachmentId = Shader.PropertyToID("_CameraColorAttachment"),
		depthAttachmentId = Shader.PropertyToID("_CameraDepthAttachment");
```

In `Render` we now have to pass the color attachment to `PostFXStack.Render`, which is functionally equivalent to what we did before.

```
		if (postFXStack.IsActive) {
			postFXStack.Render(colorAttachmentId);
		}
```

In `Setup` we now have to get two buffers instead of one composite buffer. The color buffer has no depth, while the depth buffer's format is `RenderTextureFormat.Depth` and its filter mode is `FilterMode.Point`, because blending depth data makes no sense. Both attachments can be set with a single invocation of `SetRenderTarget`, using the same load and store actions for each.

```
		if (postFXStack.IsActive) {
			if (flags > CameraClearFlags.Color) {
				flags = CameraClearFlags.Color;
			}
			buffer.GetTemporaryRT(
				colorAttachmentId, camera.pixelWidth, camera.pixelHeight,
				0, FilterMode.Bilinear, useHDR ?
					RenderTextureFormat.DefaultHDR : RenderTextureFormat.Default
			);
			buffer.GetTemporaryRT(
				depthAttachmentId, camera.pixelWidth, camera.pixelHeight,
				32, FilterMode.Point, RenderTextureFormat.Depth
			);
			buffer.SetRenderTarget(
				colorAttachmentId,
				RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store,
				depthAttachmentId,
				RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
			);
		}
```

Both buffers have to be released as well. Once that's done our RP still works the same way as before, but now with frame buffer attachments that we can accessed separately.

```
	void Cleanup () {
		lighting.Cleanup();
		if (postFXStack.IsActive) {
			buffer.ReleaseTemporaryRT(colorAttachmentId);
			buffer.ReleaseTemporaryRT(depthAttachmentId);
		}
	}
```

3.2 Copying Depth

We cannot sample the depth buffer at the same time that it's used for rendering. We have to make a copy of it. So introduce a *_CameraDepthTexture* identifier and add a boolean field to indicate whether we're using a depth texture. We should only bother with copying depth when needed, which we'll determine in `Render` after getting the camera settings. But we'll initially simply always enable it.

```
	static int
		colorAttachmentId = Shader.PropertyToID("_CameraColorAttachment"),
		depthAttachmentId = Shader.PropertyToID("_CameraDepthAttachment"),
		depthTextureId = Shader.PropertyToID("_CameraDepthTexture");

	…

	bool useDepthTexture;

	public void Render (…) {
		…
		CameraSettings cameraSettings =
			crpCamera ? crpCamera.Settings : defaultCameraSettings;

		useDepthTexture = true;

		…
	}
```

Create a new `CopyAttachments` method that gets a temporary duplicate depth texture if needed and copies the depth attachment data to it. This can be done by invoking `CopyTexture` on the command buffer with a source and destination texture. This is much more efficient than doing it via a full-screen draw call. Also make sure to release the extra depth texture in `Cleanup`.

```
	void Cleanup () {
		…
		if (useDepthTexture) {
			buffer.ReleaseTemporaryRT(depthTextureId);
		}
	}

	…

	void CopyAttachments () {
		if (useDepthTexture) {
			buffer.GetTemporaryRT(
				depthTextureId, camera.pixelWidth, camera.pixelHeight,
				32, FilterMode.Point, RenderTextureFormat.Depth
			);
			buffer.CopyTexture(depthAttachmentId, depthTextureId);
			ExecuteBuffer();
		}
	}
```

We'll copy the attachments only once, after all opaque geometry has been drawn, so after the skybox in `Render`. This means that the depth texture is only available when rendering transparent objects.

```
		context.DrawSkybox(camera);
		CopyAttachments();
```

3.3 Copying Depth Without Post FX

Copying depth only works if we have a depth attachment to copy from, which is currently only the case when post FX are enabled. To make it possible without post FX we'll also need to use an intermediate frame buffer when a depth texture is used. Introduce a `useIntermediateBuffer` boolean field to keep track of this, initialized in `Setup` before potentially getting the attachments. This should now be done when either a depth texture is used or post FX are active. `Cleanup` is affected in the same way.

```
	bool useDepthTexture, useIntermediateBuffer;

	…

	void Setup () {
		context.SetupCameraProperties(camera);
		CameraClearFlags flags = camera.clearFlags;

		useIntermediateBuffer = useDepthTexture || postFXStack.IsActive;
		if (useIntermediateBuffer) {
			if (flags > CameraClearFlags.Color) {
				flags = CameraClearFlags.Color;
			}
			…
		}

		…
	}

	void Cleanup () {
		lighting.Cleanup();
		if (useIntermediateBuffer) {
			buffer.ReleaseTemporaryRT(colorAttachmentId);
			buffer.ReleaseTemporaryRT(depthAttachmentId);
		//}
			if (useDepthTexture) {
				buffer.ReleaseTemporaryRT(depthTextureId);
			}
		}
	}
```

But now rendering fails when no post FX are active, because we're only rendering to the intermediate buffer. We have to perform a final copy to the camera's target. Unfortunately we can only use `CopyTexture` to copy to a render texture, not to the final frame buffer. We could use the post FX copy pass to do it, but this step is specific to the camera renderer so we'll create a dedicated *CameraRenderer* shader for it. It starts the same as the *PostFX* shader but with only a copy pass and it includes its own HLSL file.

```
Shader "Hidden/Custom RP/Camera Renderer" {

	SubShader {
		Cull Off
		ZTest Always
		ZWrite Off

		HLSLINCLUDE
		#include "../ShaderLibrary/Common.hlsl"
		#include "CameraRendererPasses.hlsl"
		ENDHLSL

		Pass {
			Name "Copy"

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

The new *CameraRendererPasses* HLSL file has the same `Varyings` struct and `DefaultPassVertex` function as *PostFXStackPasses*. It also has a *_SourceTexture* texture and a `CopyPassFragment` function that simply returns the sampled source texture.

```
#ifndef CUSTOM_CAMERA_RENDERER_PASSES_INCLUDED
#define CUSTOM_CAMERA_RENDERER_PASSES_INCLUDED

TEXTURE2D(_SourceTexture);

struct Varyings { … };

Varyings DefaultPassVertex (uint vertexID : SV_VertexID) { … }

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

#endif
```

Next, add a material field to `CameraRenderer`. To initialize it create a public constructor method with a shader parameter and have it invoke `CoreUtils.CreateEngineMaterial` with the shader as an argument. That method creates a new material and sets it to be hidden in the editor makes sure that it doesn't get saved as an asset, so we don't have to do this explictly ourselves. It also logs an error if the shader is missing.

```
	Material material;

	public CameraRenderer (Shader shader) {
		material = CoreUtils.CreateEngineMaterial(shader);
	}
```

Also add a public `Dispose` method that gets rid of the material by passing it to `CoreUtils.Destroy`. That method either regularly or immediately destroys the material, depending on whether Unity is in play mode or not. We need to do this because new RP instances and thus renderers get created whenever the RP asset is modified, which could result in many materials getting created in the editor.

```
	public void Dispose () {
		CoreUtils.Destroy(material);
	}
```

Now `CustomRenderPipeline` must provide a shader when it constructs its renderer. So we'll do it in its own constructor method, also adding a parameter for the camera renderer shader to it.

```
	CameraRenderer renderer; // = new CameraRenderer();

	…

	public CustomRenderPipeline (
		bool allowHDR,
		bool useDynamicBatching, bool useGPUInstancing, bool useSRPBatcher,
		bool useLightsPerObject, ShadowSettings shadowSettings,
		PostFXSettings postFXSettings, int colorLUTResolution, Shader cameraRendererShader
	) {
		…
		renderer = new CameraRenderer(cameraRendererShader);
	}
```

And it must also invoke `Dispose` on the renderer from now on when it is disposed itself. We've already created a `Dispose` method for it, but for editor code only. Rename that version to `DisposeForEditor` and only have it reset the light mapping delegate.

```
	partial void DisposeForEditor ();

#if UNITY_EDITOR

	…

	partial void DisposeForEditor () {
		//base.Dispose(disposing);
		Lightmapping.ResetDelegate();
	}
```

Then add a new `Dispose` method that isn't editor-only, which invokes its base implementation, the version for the editor, and finally disposes the renderer.

```
	protected override void Dispose (bool disposing) {
		base.Dispose(disposing);
		DisposeForEditor();
		renderer.Dispose();
	}
```

And at the top level `CustomRenderPipelineAsset` must get a shader configuration property and pass it to the pipeline constructor. Then we can finally hook up the shader.

```
	[SerializeField]
	Shader cameraRendererShader = default;

	protected override RenderPipeline CreatePipeline () {
		return new CustomRenderPipeline(
			allowHDR, useDynamicBatching, useGPUInstancing, useSRPBatcher,
			useLightsPerObject, shadows, postFXSettings, (int)colorLUTResolution,
			cameraRendererShader
		);
	}
```

Camera renderer shader assigned.

At this point `CameraRenderer` has a functional material. Also add the *_SourceTexture* identifier to it and the give it a `Draw` method similar to the one in `PostFXStack`, except without a parameter for a pass, as we only have a single pass at this point.

```
	static int
		colorAttachmentId = Shader.PropertyToID("_CameraColorAttachment"),
		depthAttachmentId = Shader.PropertyToID("_CameraDepthAttachment"),
		depthTextureId = Shader.PropertyToID("_CameraDepthTexture"),
		sourceTextureId = Shader.PropertyToID("_SourceTexture");

	…

	void Draw (RenderTargetIdentifier from, RenderTargetIdentifier to) {
		buffer.SetGlobalTexture(sourceTextureId, from);
		buffer.SetRenderTarget(
			to, RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
		);
		buffer.DrawProcedural(
			Matrix4x4.identity, material, 0, MeshTopology.Triangles, 3
		);
	}
```

To finally fix our renderer copy the color attachment to the camera target in `Render` by invoking `Draw`, if post FX aren't active but we do use an intermediate buffer.

```
		if (postFXStack.IsActive) {
			postFXStack.Render(colorAttachmentId);
		}
		else if (useIntermediateBuffer) {
			Draw(colorAttachmentId, BuiltinRenderTextureType.CameraTarget);
			ExecuteBuffer();
		}
```

3.4 Reconstructing View-Space Depth

To sample the depth texture we need the UV coordinates of the fragment, which are in screen space. We can find those by dividing its position by the screen pixel dimensions, which Unity makes available via the XY components of `float4 _ScreenParams`, so add it to *UnityInput*.

```
float4 unity_OrthoParams;
float4 _ProjectionParams;
float4 _ScreenParams;
```

Then we can add the fragment UV and buffer depth to `Fragment`. Retrieve the buffer depth by sampling the camera depth texture with a point clamp sampler via the `SAMPLE_DEPTH_TEXTURE_LOD` macro. This macro does the same as `SAMPLE_TEXTURE2D_LOD` but only returns the R channel.

```
TEXTURE2D(_CameraDepthTexture);

struct Fragment {
	float2 positionSS;
	float2 screenUV;
	float depth;
	float bufferDepth;
};

Fragment GetFragment (float4 positionCS_SS) {
	Fragment f;
	f.positionSS = positionSS.xy;
	f.screenUV = f.positionSS / _ScreenParams.xy;
	f.depth = IsOrthographicCamera() ?
		OrthographicDepthBufferToLinear(positionSS.z) : positionSS.w;
	f.bufferDepth =
		SAMPLE_DEPTH_TEXTURE_LOD(_CameraDepthTexture, sampler_point_clamp, f.screenUV, 0);
	return f;
}
```

This gives us the raw depth buffer value. To convert it to view-space depth we can again invoke `OrthographicDepthBufferToLinear` in case of an orthographic camera, like for the current fragment's depth. The perspective depth also has to be converted, for which we can use `LinearEyeDepth`. It needs `_ZBufferParams` as a second argument.

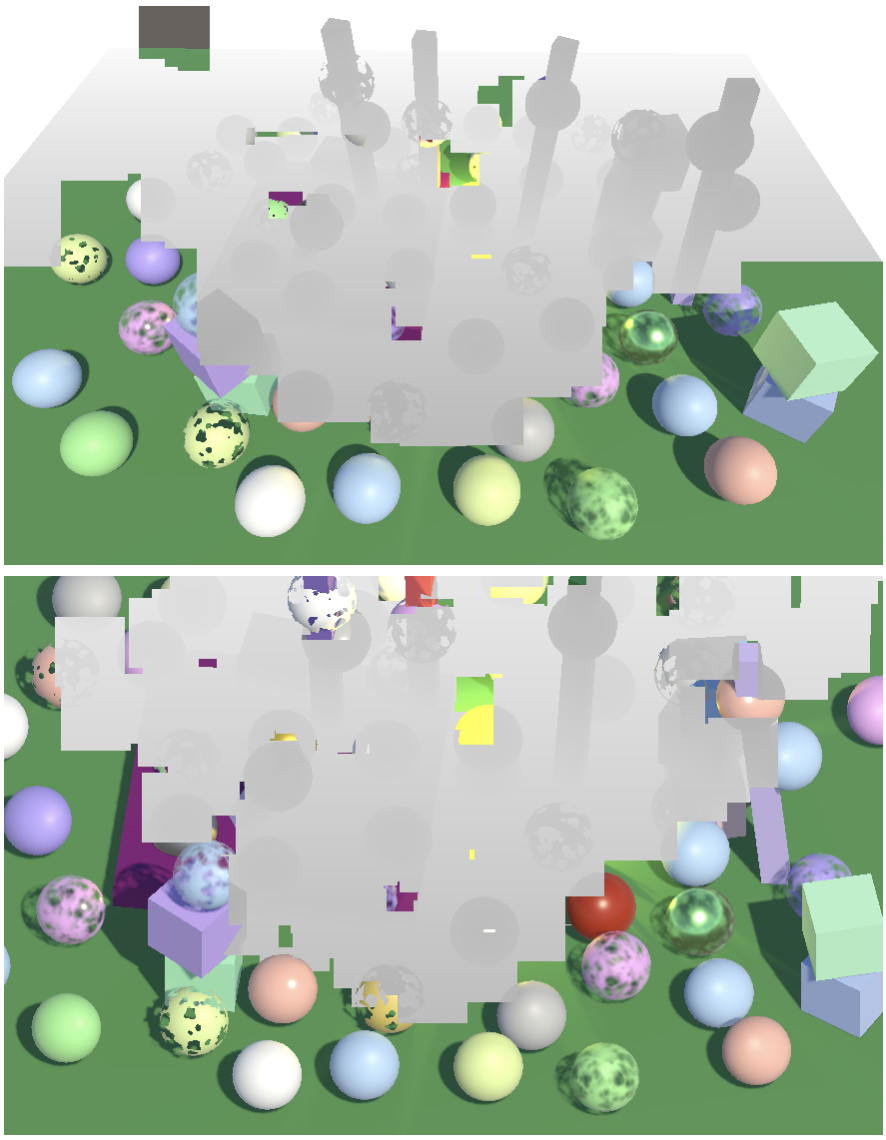
```
	f.bufferDepth = LOAD_TEXTURE2D(_CameraDepthTexture, f.positionSS).r;
	f.bufferDepth = IsOrthographicCamera() ?
		OrthographicDepthBufferToLinear(f.bufferDepth) :
		LinearEyeDepth(f.bufferDepth, _ZBufferParams);
```

`_ZBufferParams` is another `float4` made available by Unity that contains conversion factors from raw to linear depth. Add it to *UnityInput*.

```
float4 unity_OrthoParams;
float4 _ProjectionParams;
float4 _ScreenParams;
float4 _ZBufferParams;
```

To check whether we sample the buffer depth correctly return it scaled in `UnlitPassFragment`, like we tested the fragment depth earlier.

```
	InputConfig config = GetInputConfig(input.positionCS_SS, input.baseUV);
	return float4(config.fragment.bufferDepth.xxx / 20.0, 1.0);
```



Buffer depth, perspective and orthographic projections.

Remove the debug visualization once it's clear that the sampled depth is correct.

```
	//return float4(config.fragment.bufferDepth.xxx / 20.0, 1.0);
```

3.5 Optional Depth Texture

Copying depth requires extra work, especially when no post FX are used as that also requires intermediate buffers and an extra copy to the camera target. So let's make it configurable whether our RP supports copying depth. We'll create a new `CameraBufferSettings` struct for this, put in its own file, used to group all settings related to the camera buffers. Besides a toggle for copying depth also put the toggle to allow HDR in it as well. And also introduce a separate toggle to control whether depth is copied when rendering reflections. This is useful because reflections are rendered without post FX and particle systems don't show up in reflections either, so copying depth for reflections is expensive and likely useless. We do make it possible because depth could be used for other effects as well, which might be visible in reflections. Even then, keep in mind that the depth buffer is distinct per cube map reflection face, so there will be depth seams along the cube map edges.

```
[System.Serializable]
public struct CameraBufferSettings {

	public bool allowHDR;

	public bool copyDepth, copyDepthReflections;
}
```

Replace the current HDR toggle of `CustomRenderPipelineAsset` with these camera buffer settings.

```
	//[SerializeField]
	//bool allowHDR = true;

	[SerializeField]
	CameraBufferSettings cameraBuffer = new CameraBufferSettings {
		allowHDR = true
	};

	protected override RenderPipeline CreatePipeline () {
		return new CustomRenderPipeline(
			cameraBuffer, useDynamicBatching, useGPUInstancing, useSRPBatcher,
			useLightsPerObject, shadows, postFXSettings, (int)colorLUTResolution,
			cameraRendererShader
		);
	}
```

Apply the change to `CustomRenderPipeline` as well.

```
	//bool allowHDR;
	CameraBufferSettings cameraBufferSettings;

	…

	public CustomRenderPipeline (
		CameraBufferSettings cameraBufferSettings,
		bool useDynamicBatching, bool useGPUInstancing, bool useSRPBatcher,
		bool useLightsPerObject, ShadowSettings shadowSettings,
		PostFXSettings postFXSettings, int colorLUTResolution, Shader cameraRendererShader
	) {
		this.colorLUTResolution = colorLUTResolution;
		//this.allowHDR = allowHDR;
		this.cameraBufferSettings = cameraBufferSettings;
		…
	}

	protected override void Render (ScriptableRenderContext context, Camera[] cameras) {
		foreach (Camera camera in cameras) {
			renderer.Render(
				context, camera, cameraBufferSettings,
				useDynamicBatching, useGPUInstancing, useLightsPerObject,
				shadowSettings, postFXSettings, colorLUTResolution
			);
		}
	}
```

`CameraRenderer.Render` now has to use the appropriate settings depending on whether it's rendering reflections or not.

```
	public void Render (
		ScriptableRenderContext context, Camera camera,
		CameraBufferSettings bufferSettings,
		bool useDynamicBatching, bool useGPUInstancing, bool useLightsPerObject,
		ShadowSettings shadowSettings, PostFXSettings postFXSettings,
		int colorLUTResolution
	) {
		…

		//useDepthTexture = true;
		if (camera.cameraType == CameraType.Reflection) {
			useDepthTexture = bufferSettings.copyDepthReflection;
		}
		else {
			useDepthTexture = bufferSettings.copyDepth;
		}

		…
		useHDR = bufferSettings.allowHDR && camera.allowHDR;

		…
	}
```

Camera buffer settings, with HDR and non-reflection copy depth enabled.

Besides the settings for the entire RP we can also add a copy-depth toggle to `CameraSettings`, enabled by default.

```
	public bool copyDepth = true;
```

Camera copy depth toggle.

Then for regular cameras a depth texture is only used if both the RP and the camera have it enabled, similar to how HDR is controlled.

```
		if (camera.cameraType == CameraType.Reflection) {
			useDepthTexture = bufferSettings.copyDepthReflection;
		}
		else {
			useDepthTexture = bufferSettings.copyDepth && cameraSettings.copyDepth;
		}
```

3.6 Missing Texture

As the depth texture is optional it might not exist. When a shader samples it anyway the result will be random. It could be either an empty texture or an old copy, potentially of another camera. It's also possible that a shader samples the depth texture too early, during the opaque rendering phase. The least we can do is make sure that invalid samples will produce consistent results. We do this by creating a default missing texture in the constructor method of `CameraRender`. There is no `CoreUtils` method for textures, so we'll set its hide flags to `HideFlags.HideAndDontSave` ourselves. Name it *Missing* so it's obvious that a wrong texture is used when inspecting shader properties via the frame debugger. Make it a simple 1×1 texture with all channels set to 0.5. Also destroy it appropriately when the renderer is disposed.

```
	Texture2D missingTexture;

	public CameraRenderer (Shader shader) {
		material = CoreUtils.CreateEngineMaterial(shader);
		missingTexture = new Texture2D(1, 1) {
			hideFlags = HideFlags.HideAndDontSave,
			name = "Missing"
		};
		missingTexture.SetPixel(0, 0, Color.white * 0.5f);
		missingTexture.Apply(true, true);
	}

	public void Dispose () {
		CoreUtils.Destroy(material);
		CoreUtils.Destroy(missingTexture);
	}
```

Use the missing texture for the depth texture at the end of `Setup`.

```
	void Setup () {
		…
		buffer.BeginSample(SampleName);
		buffer.SetGlobalTexture(depthTextureId, missingTexture);
		ExecuteBuffer();
	}
```

3.7 Fading Particles Nearby Background

Now that we have a functional depth texture we can move on to finally support soft particles. The first step is to add shader properties for a soft particles keyword toggle, a distance, and a range to *UnlitParticles*, similar to the near fade properties. In this case the distance is measured from whatever is behind the particles, so we set it to zero by default.

```
		[Toggle(_SOFT_PARTICLES)] _SoftParticles ("Soft Particles", Float) = 0
		_SoftParticlesDistance ("Soft Particles Distance", Range(0.0, 10.0)) = 0
		_SoftParticlesRange ("Soft Particles Range", Range(0.01, 10.0)) = 1
```

Add the shader feature for it as well.

```
			#pragma shader_feature _SOFT_PARTICLES
```

Like for near fading, set an appropriate config field to `true` in `UnlitPassFragment` if the keyword is defined.

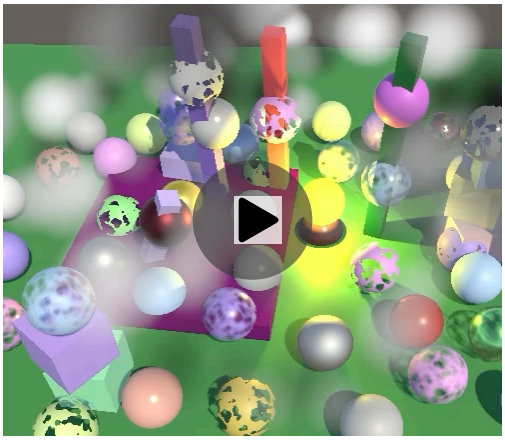
```
	#if defined(_NEAR_FADE)
		config.nearFade = true;
	#endif
	#if defined(_SOFT_PARTICLES)
		config.softParticles = true;
	#endif
```

In *UnlitInput*, add the new shader properties to `UnityPerMaterial` and the field to `InputConfig`.

```
UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
	…
	UNITY_DEFINE_INSTANCED_PROP(float, _SoftParticlesDistance)
	UNITY_DEFINE_INSTANCED_PROP(float, _SoftParticlesRange)
	UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
	UNITY_DEFINE_INSTANCED_PROP(float, _ZWrite)
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)

#define INPUT_PROP(name) UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, name)

struct InputConfig {
	…
	bool softParticles;
};

InputConfig GetInputConfig (float4 positionCC_SS, float2 baseUV) {
	…
	c.softParticles = false;
	return c;
}
```

Then apply another near attenuation in `GetBase`, this time based on the fragment's buffer depth minus its own depth.

```
	if (c.nearFade) {
		float nearAttenuation = (c.fragment.depth - INPUT_PROP(_NearFadeDistance)) /
			INPUT_PROP(_NearFadeRange);
		baseMap.a *= saturate(nearAttenuation);
	}
	if (c.softParticles) {
		float depthDelta = c.fragment.bufferDepth - c.fragment.depth;
		float nearAttenuation = (depthDelta - INPUT_PROP(_SoftParticlesDistance)) /
			INPUT_PROP(_SoftParticlesRange);
		baseMap.a *= saturate(nearAttenuation);
	}
```



Soft particles, adjusting fade range.

3.8 No Copy Texture Support

This all works fine, but only as long as direct copying of textures via `CopyTexture` is supported, at least at a basic level. This is mostly the case, but not for WebGL 2.0. So if we also want to support WebGL 2.0 we have fall back to copying via our shader instead, which is less efficient but at least works.

Keep track of whether `CopyTexture` is supported via a static boolean field in `CameraRenderer`. Initially set it to `false` so we can test the fallback approach even though our development machines all support it.

```
	static bool copyTextureSupported = false;
```

In `CopyAttachments` copy the depth depth via `CopyTexture` if supported, otherwise fall back to using our `Draw` method.

```
	void CopyAttachments () {
		if (useDepthTexture) {
			buffer.GetTemporaryRT(
				depthTextureId, camera.pixelWidth, camera.pixelHeight,
				32, FilterMode.Point, RenderTextureFormat.Depth
			);
			if (copyTextureSupported) {
				buffer.CopyTexture(depthAttachmentId, depthTextureId);
			}
			else {
				Draw(depthAttachmentId, depthTextureId);
			}
			ExecuteBuffer();
		}
	}
```

This initially fails to produce correct results because `Draw` changes the render target, so further drawing goes wrong. We have to set the render target back to the camera buffer afterwards, loading our attachments again.

```
		if (copyTextureSupported) {
			buffer.CopyTexture(depthAttachmentId, depthTextureId);
		}
		else {
			Draw(depthAttachmentId, depthTextureId);
			buffer.SetRenderTarget(
				colorAttachmentId,
				RenderBufferLoadAction.Load, RenderBufferStoreAction.Store,
				depthAttachmentId,
				RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
			);
		}
```

The second thing that goes wrong is that the depth doesn't get copied at all, because our copy pass only writes to the default shader target, which is for color data, not depth. To copy depth instead we need to add a second copy-depth pass to the *CameraRenderer* shader that writes depth instead of color. We do that by settings its `ColorMask` to zero and turning `ZWrite` on. It also needs a special fragment function, which we'll name `CopyDepthPassFragment`.

```
		Pass {
			Name "Copy Depth"

			ColorMask 0
			ZWrite On

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

The new fragment function has to sample depth and return it as a single `float` with the `SV_DEPTH` semantic, instead of a `float4` with the `SV_TARGET` semantic. This way we sample the raw depth buffer value and directly use it for the new depth of the fragment.

```
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);
}
```

Next, go back to `CameraRenderer` and add a boolean parameter to `Draw` to indicate whether we're drawing from and to depth, set to `false` by default. If so, use the second pass instead of the first pass.

```
	public void Draw (
		RenderTargetIdentifier from, RenderTargetIdentifier to, bool isDepth = false
	) {
		…
		buffer.DrawProcedural(
			Matrix4x4.identity, material, isDepth ? 1 : 0, MeshTopology.Triangles, 3
		);
	}
```

Then indicate that we're working with depth when copying the depth buffer.

```
			Draw(depthAttachmentId, depthTextureId, true);
```

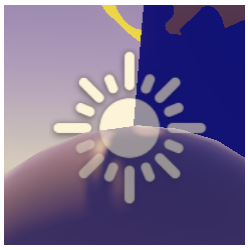
After veryfying that this approach also works, determine whether `CopyTexture` is supported by checking `SystemInfo.copyTextureSupport`. Any level of support greater than none is sufficient.

```
	static bool copyTextureSupported =
		SystemInfo.copyTextureSupport > CopyTextureSupport.None;
```

3.9 Gizmos and Depth

Now that we have a way to draw depth, we can use it to make our gizmos depth-aware again in combination with post FX or when using a depth texture. In `DrawGizmosBeforeFX`, before drawing the first gizmos, copy depth to the camera target if we use an intermediate buffer.

```
	partial void DrawGizmosBeforeFX () {
		if (Handles.ShouldRenderGizmos()) {
			if (useIntermediateBuffer) {
				Draw(depthAttachmentId, BuiltinRenderTextureType.CameraTarget, true);
				ExecuteBuffer();
			}
			context.DrawGizmos(camera, GizmoSubset.PreImageEffects);
		}
	}
```



Gizmos recognizing depth.

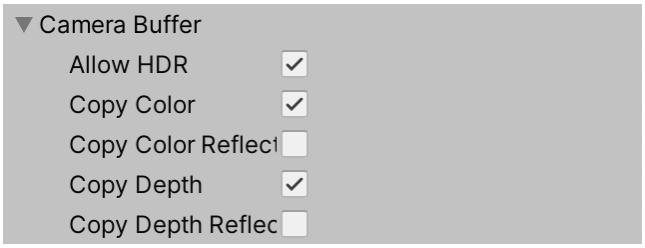
4 Distortion

Another feature of Unity's particles that we'll also support is distortion, which can be used to create effects like atmospheric refraction caused by heat. This requires sampling of the color buffer, like we're already sampling the depth buffer, but with the addition of a UV offset.

4.1 Color Copy Texture

We begin by adding toggles for copying color to `CameraBufferSettings`, again a separate one for regular and for reflection cameras.

```
	public bool copyColor, copyColorReflection, copyDepth, copyDepthReflection;
```



Copying color and depth.

Make copying color configurable per camera as well.

```
	public bool copyColor = true, copyDepth = true;
```

Also enabled for camera.

`CameraRendering` now also has to keep track of the identifier for a color texture and whether a color texture is used.

```
		colorTextureId = Shader.PropertyToID("_CameraColorTexture"),
		depthTextureId = Shader.PropertyToID("_CameraDepthTexture"),
		sourceTextureId = Shader.PropertyToID("_SourceTexture");

	…

	bool useColorTexture, useDepthTexture, useIntermediateBuffer;

	…

	public void Render (…) {
		…

		if (camera.cameraType == CameraType.Reflection) {
			useColorTexture = bufferSettings.copyColorReflection;
			useDepthTexture = bufferSettings.copyDepthReflection;
		}
		else {
			useColorTexture = bufferSettings.copyColor && cameraSettings.copyColor;
			useDepthTexture = bufferSettings.copyDepth && cameraSettings.copyDepth;
		}

		…
	}
```

Whether we use an intermediate buffer now also depends on whether a color texture is used. And we should also initially set the color texture to the missing texture. Release it when cleaning up as well.

```
	void Setup () {
		…

		useIntermediateBuffer =
			useColorTexture || useDepthTexture || postFXStack.IsActive;
		…
		buffer.BeginSample(SampleName);
		buffer.SetGlobalTexture(colorTextureId, missingTexture);
		buffer.SetGlobalTexture(depthTextureId, missingTexture);
		ExecuteBuffer();

	}

	void Cleanup () {
		lighting.Cleanup();
		if (useIntermediateBuffer) {
			buffer.ReleaseTemporaryRT(colorAttachmentId);
			buffer.ReleaseTemporaryRT(depthAttachmentId);
			if (useColorTexture) {
				buffer.ReleaseTemporaryRT(colorTextureId);
			}
			if (useDepthTexture) {
				buffer.ReleaseTemporaryRT(depthTextureId);
			}
		}
	}
```

We now need to copy the camera attachments when either a color or a depth texture is used, or both. Let's make the invocation of `CopyAttachments` dependent on that.

```
		context.DrawSkybox(camera);
		if (useColorTexture || useDepthTexture) {
			CopyAttachments();
		}
```

Then we can have it copy both textures separately and afterwards reset the render target and execute the buffer once.

```
	void CopyAttachments () {
		if (useColorTexture) {
			buffer.GetTemporaryRT(
				colorTextureId, camera.pixelWidth, camera.pixelHeight,
				0, FilterMode.Bilinear, useHDR ?
					RenderTextureFormat.DefaultHDR : RenderTextureFormat.Default
			);
			if (copyTextureSupported) {
				buffer.CopyTexture(colorAttachmentId, colorTextureId);
			}
			else {
				Draw(colorAttachmentId, colorTextureId);
			}
		}
		if (useDepthTexture) {
			buffer.GetTemporaryRT(
				depthTextureId, camera.pixelWidth, camera.pixelHeight,
				32, FilterMode.Point, RenderTextureFormat.Depth
			);
			if (copyTextureSupported) {
				buffer.CopyTexture(depthAttachmentId, depthTextureId);
			}
			else {
				Draw(depthAttachmentId, depthTextureId, true);
				//buffer.SetRenderTarget(…);
			}
			//ExecuteBuffer();
		}
		if (!copyTextureSupported) {
			buffer.SetRenderTarget(
				colorAttachmentId,
				RenderBufferLoadAction.Load, RenderBufferStoreAction.Store,
				depthAttachmentId,
				RenderBufferLoadAction.Load, RenderBufferStoreAction.Store
			);
		}
		ExecuteBuffer();
	}
```

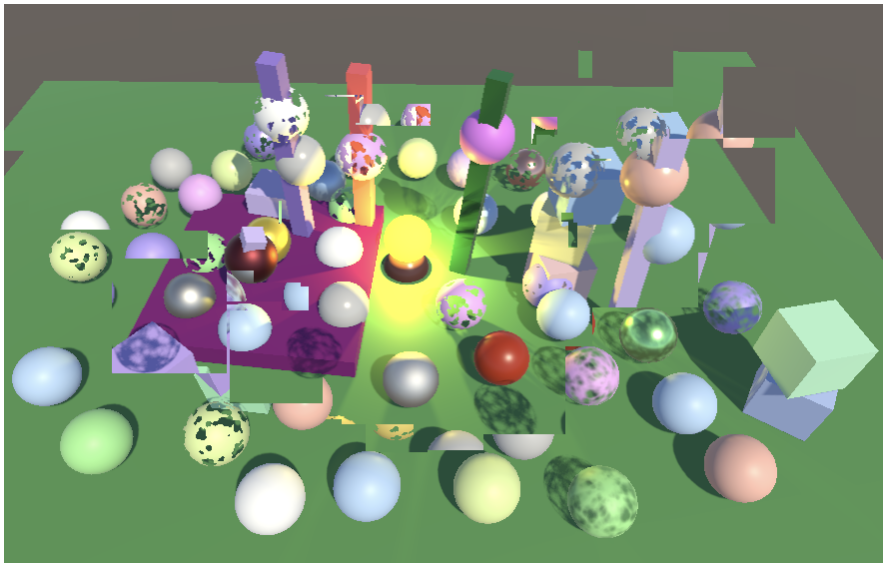
4.2 Sampling the Buffer Color

To sample the camera color texture add it to *Fragment*. We won't add a buffer color property to `Fragment`, as we're not interested in the color at its exact location. Instead we introduce a `GetBufferColor` function that takes a fragment and UV offset as parameters, retuning the sampled color.

```
TEXTURE2D(_CameraColorTexture);
TEXTURE2D(_CameraDepthTexture);

struct Fragment { … };

Fragment GetFragment (float4 positionCS_SS) { … }

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);
}
```

To test this return the buffer color with a small offset like 5% in both dimensions in `UnlitPassFragment`.

```
	InputConfig config = GetInputConfig(input.positionCS_SS, input.baseUV);
	return GetBufferColor(config.fragment, 0.05);
```



Sampling camera color buffer with offset.

Note that because the colors are copied after the opaque phase transparent objects are missing from it. Thus the particles erase all transparent objects that were drawn before them, including each other. At the same time depth plays no role in this case, so colors of fragments that are closer to the camera plane than the fragment itself get copied as well. Remove the debug visualization when it's clear that it works.

```
	//return GetBufferColor(config.fragment, 0.05);
```

Could sampling in front of a fragment be avoided?

Yes, up to a point. For an example, see the Flow / Looking Through Water tutorial.

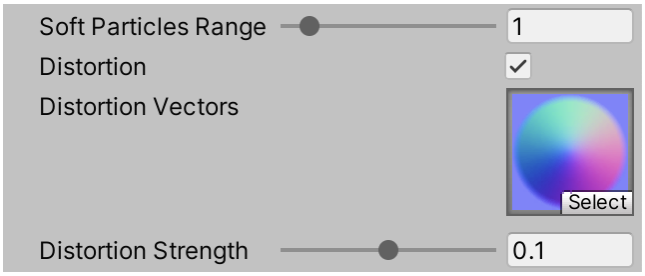
4.3 Distortion Vectors

To create a useful distortion effect we need a map of smoothly transitioning distortion vectors. Here is a simple map for a single round particle. It's a normal map, so import it as such.

Particle distortion map.

Add a keyword toggle shader property to *UnlitParticles*, along with distortion map and strength properties. The distortion will be applied as a screen-space UV offset, so small values are required. Let's use a strength range of 0–0.2, with 0.1 as the default.

```
		[Toggle(_DISTORTION)] _Distortion ("Distortion", Float) = 0
		[NoScaleOffset] _DistortionMap("Distortion Vectors", 2D) = "bumb" {}
		_DistortionStrength("Distortion Strength", Range(0.0, 0.2)) = 0.1
```



Distortion enabled.

Add the required shader feature.

```
			#pragma shader_feature _DISTORTION
```

Then add the distortion map and strength properties to *UnlitInput*.

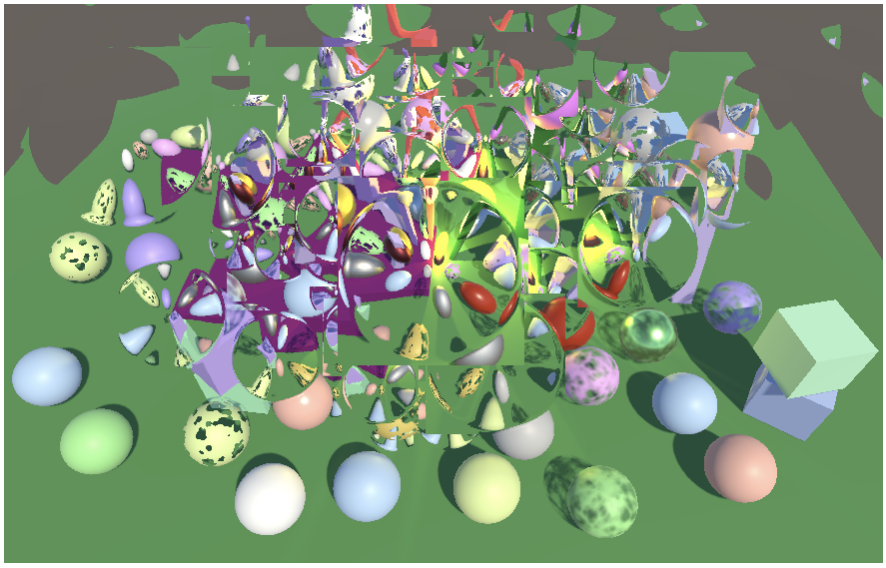
```
TEXTURE2D(_BaseMap);
TEXTURE2D(_DistortionMap);
SAMPLER(sampler_BaseMap);

UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
	…
	UNITY_DEFINE_INSTANCED_PROP(float, _SoftParticlesRange)
	UNITY_DEFINE_INSTANCED_PROP(float, _DistortionStrength)
	…
UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
```

Introduce a new `GetDistortion` function that returns a `float2` vector. Have it sample the distortion map and apply flipbook blending as for the base map, then decode the normal scaled by the distortion strength. We only need the XY components of the vector, so discard Z.

```
float2 GetDistortion (InputConfig c) {
	float4 rawMap = SAMPLE_TEXTURE2D(_DistortionMap, sampler_BaseMap, c.baseUV);
	if (c.flipbookBlending) {
		rawMap = lerp(
			rawMap, SAMPLE_TEXTURE2D(_DistortionMap, sampler_BaseMap, c.flipbookUVB.xy),
			c.flipbookUVB.z
		);
	}
	return DecodeNormal(rawMap, INPUT_PROP(_DistortionStrength)).xy;
}
```

In `UnlitPassFragment`, if distortion is enabled retrieve it and use it as an offset to get the buffer color, overriding the base color. Do this after clipping.

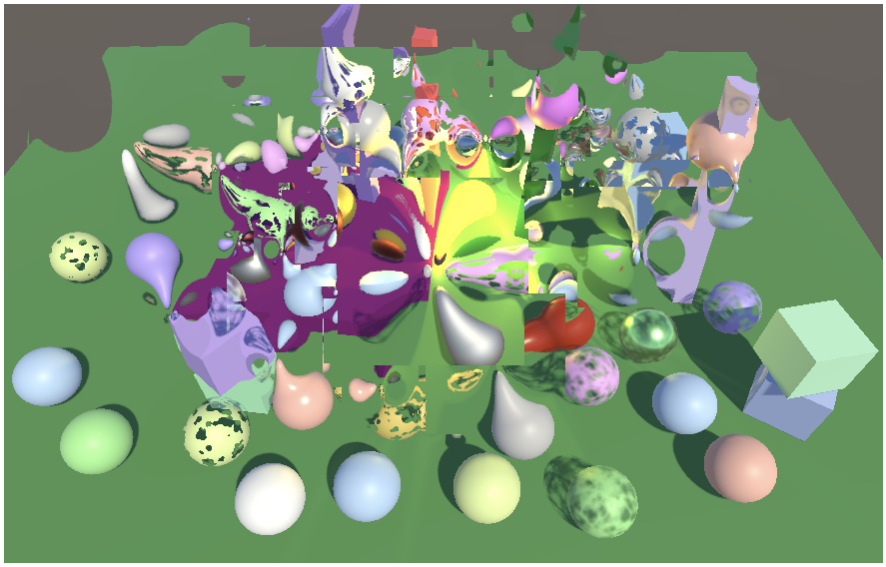
```
	float4 base = GetBase(config);
	#if defined(_CLIPPING)
		clip(base.a - GetCutoff(config));
	#endif
	#if defined(_DISTORTION)
		float2 distortion = GetDistortion(config);
		base = GetBufferColor(config.fragment, distortion);
	#endif
```



Distorted color buffer.

The result is that particles radially distort the color texture, except in their corners because the distortion vectors are zero there. But the distortion effect should depend on the visual strength of the particles, which is controlled by the original base alpha. So modulate the distortion offset vector with the base alpha.

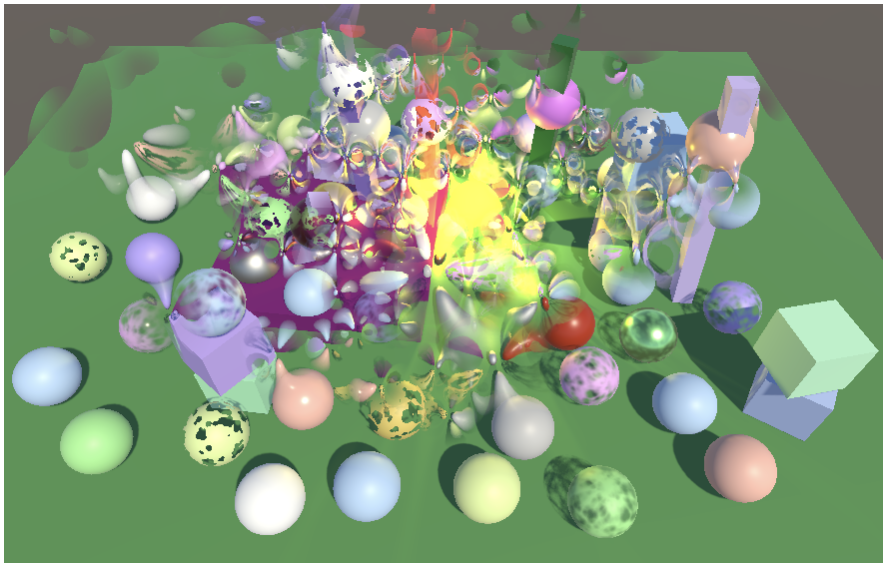
```
		float2 distortion = GetDistortion(config) * base.a;
```



Modulated distortion.

At this point we still get hard edges betraying that particles completely overlap each other and are rectangular. We hide that by keeping the original alpha of the particles.

```
		base.rgb = GetBufferColor(config.fragment, distortion).rgb;
```



Faded distortion.

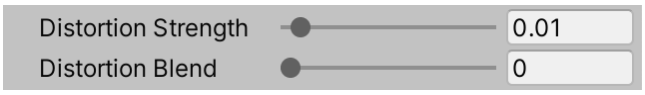
Now the distorted color texture samples fade as well, which makes the undistorted background and other particles partially visible again. The result is a smooth mess that doesn't make physical sense but is enough to provide the illusion of atmospheric refraction. This can be improved further by tweaking the distortion strength along with smoothly fading particles in and out by adjusting their color during their lifetime. Also, the offset vectors are aligned with the screen and aren't affected by the orientation of the particle. So if the particles are set to rotate during their lifetime their individual distortion patterns will appear to twist.

Distortion effect.

4.4 Distortion Blend

Currently when distortion is enabled we completely replace the original color of the particles, only keeping their alpha. The particle color can be combined with the distorted color buffer in various ways. We'll add a simple distortion blend shader property, to interpolate between the particle's own color and the distortion that it causes, using the same approach as Unity's particle shaders.

```
		_DistortionStrength("Distortion Strength", Range(0.0, 0.2)) = 0.1
		_DistortionBlend("Distortion Blend", Range(0.0, 1.0)) = 1
```



Distortion blend slider.

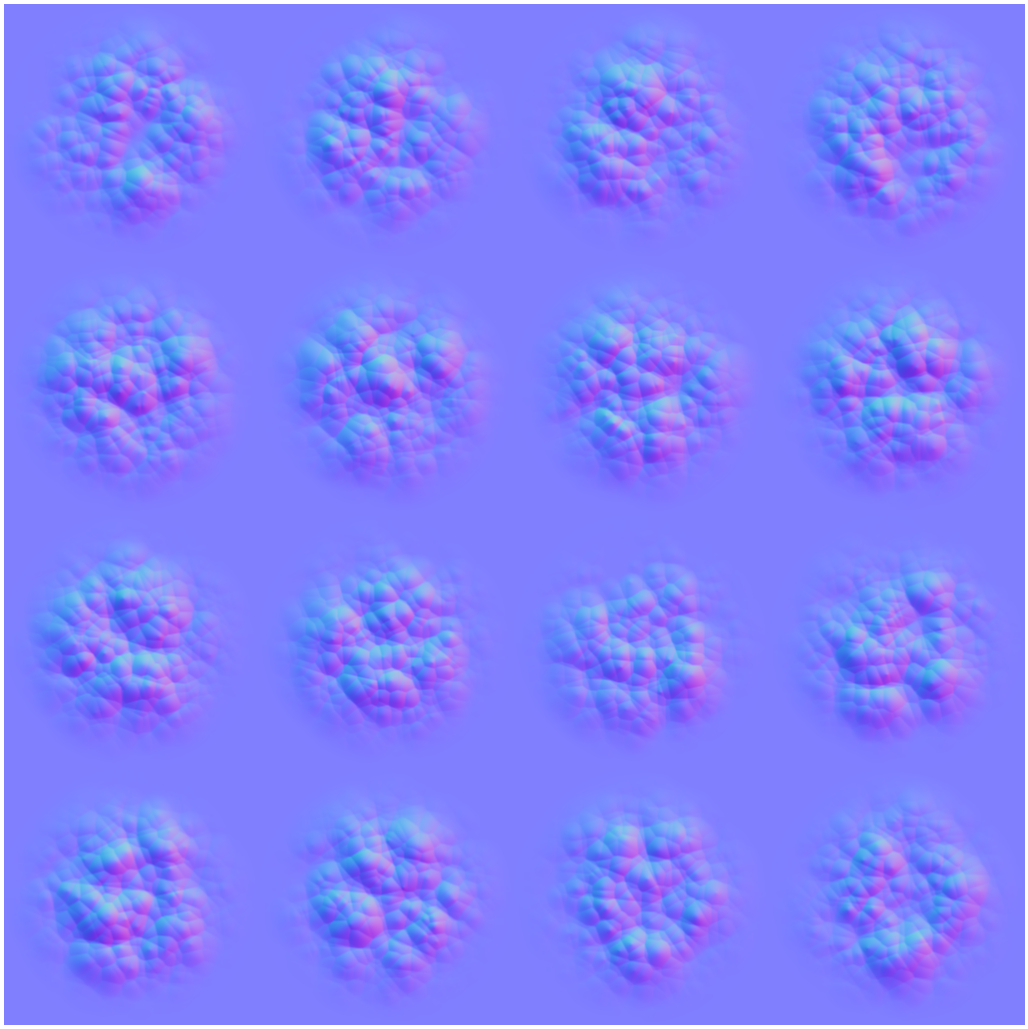
Add the property to *UnlitInput* along with a function to get it.

```
	UNITY_DEFINE_INSTANCED_PROP(float, _DistortionStrength)
	UNITY_DEFINE_INSTANCED_PROP(float, _DistortionBlend)

…

float GetDistortionBlend (InputConfig c) {
	return INPUT_PROP(_DistortionBlend);
}
```

The idea is that when the blend slider is at 1 we only see the distortion. Lowering it makes the particle color appear, but it won't completely hide the distortion. Instead we interpolate from distortion to particle color based on its alpha minus the blend slider, saturated. Thus when distortion is enabled the particle's own color will always be weaker and appear smaller compared to when distortion is disabled, unless where it's fully opaque. Perform the interpolation in `UnlitPassFragment`.

```
	#if defined(_DISTORTION)
		float2 distortion = GetDistortion(config) * base.a;
		base.rgb = lerp(
			GetBufferColor(config.fragment, distortion).rgb, base.rgb,
			saturate(base.a - GetDistortionBlend(config))
		);
	#endif
```

This looks better for more complex particles, like our flipbook example. So here is a distortion texture for the flipbook.



Distortion map for particle flipbook.

This can be used to create interesting distortion effects. Realistic effects would be subtle, as a little distortion is enough when the system is in motion. But for demonstration purposes I made the effects strong so they're visually obvious, even in screenshots.

Distortion with flipbook and post FX.

4.5 Fixing Nonstandard Cameras

Our current approach works when we only use a single camera, but it fails when rendering to an intermediate texture without post FX. This happens because we're performing a regular copy to the camera target, which ignores the viewport and final blend modes. So `CameraRenderer` also needs a `FinalPass` method. It's a copy of `PostFXStack.FinalPass`, except that we'll use the regular copy pass, so we should set the blend mode back to one-zero afterwards to not affect other copy actions. The source texture is always the color attachment and the final blend mode becomes a parameter.

Again, for Unity 2022 we do care about loading the buffers if we're not rendering to the full viewport.

```
	static Rect fullViewRect = new Rect(0f, 0f, 1f, 1f);
	…

	void DrawFinal (CameraSettings.FinalBlendMode finalBlendMode) {
		buffer.SetGlobalFloat(srcBlendId, (float)finalBlendMode.source);
		buffer.SetGlobalFloat(dstBlendId, (float)finalBlendMode.destination);
		buffer.SetGlobalTexture(sourceTextureId, colorAttachmentId);
		buffer.SetRenderTarget(
			BuiltinRenderTextureType.CameraTarget,
			finalBlendMode.destination == BlendMode.Zero && camera.rect == fullViewRect?
				RenderBufferLoadAction.DontCare : RenderBufferLoadAction.Load,
			RenderBufferStoreAction.Store
		);
		buffer.SetViewport(camera.pixelRect);
		buffer.DrawProcedural(
			Matrix4x4.identity, material, 0, MeshTopology.Triangles, 3
		);
		buffer.SetGlobalFloat(srcBlendId, 1f);
		buffer.SetGlobalFloat(dstBlendId, 0f);
	}
```

In this case we'll name the blend mode shader properties *_CameraSrcBlend* and *_CameraSrcBlend*.

```
		sourceTextureId = Shader.PropertyToID("_SourceTexture"),
		srcBlendId = Shader.PropertyToID("_CameraSrcBlend"),
		dstBlendId = Shader.PropertyToID("_CameraDstBlend");
```

Adjust the copy pass of *CameraRenderer* to rely on these properties.

```
		Pass {
			Name "Copy"

			Blend [_CameraSrcBlend] [_CameraDstBlend]

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

Finally, invoke `DrawFinal` instead of `Draw` in `Render`.

```
		if (postFXStack.IsActive) {
			postFXStack.Render(colorAttachmentId);
		}
		else if (useIntermediateBuffer) {
			DrawFinal(cameraSettings.finalBlendMode);
			ExecuteBuffer();
		}
```

Note that the color and depth textures only contain what the current camera rendered. Distortion particles and similar effects won't pick up data from other cameras.

The next tutorial is Render Scale.

made by Jasper Flick