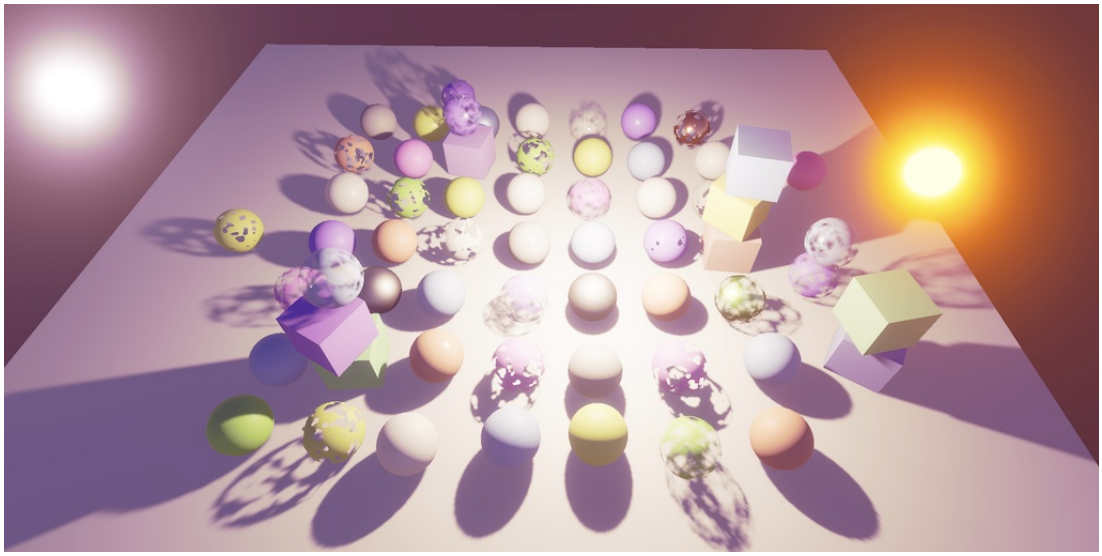
published 2023-11-21

Custom SRP 2.3.0 Shadow Textures

Let the render graph manage shadow textures.
Merge lighting code into lighting pass.



Showing off the top right camera of the Multiple Cameras scene.

This tutorial is made with Unity 2022.3.12f1 and follows Custom SRP 2.2.0.

1 Managed Shadow Textures

In the previous tutorial we let the render graph manage the camera textures. This time we also let it manage the shadow textures.

1.1 Shadows

Create a `ShadowTextures` ref struct type, like `CameraRendererTextures` but with only two texture handles, for the directional and other shadow atlases.

```
using UnityEngine.Experimental.Rendering.RenderGraphModule;

public readonly ref struct ShadowTextures
{
	public readonly TextureHandle directionalAtlas, otherAtlas;

	public ShadowTextures(
		TextureHandle directionalAtlas,
		TextureHandle otherAtlas)
	{
		this.directionalAtlas = directionalAtlas;
		this.otherAtlas = otherAtlas;
	}
}
```

`Shadows` will use these handles instead of its current render textures, so give it fields for them.

```
	TextureHandle directionalAtlas, otherAtlas;
```

We'll also let `Shadows` create and register the handles, via a new `GetRenderTextures` method that returns the shadow textures. It needs the render graph and a builder as parameters. To keep things simple we'll always provide valid handles. If an atlas is not needed we use the `defaultShadowTexture` from the graph's `defaultResources`. That refers to a 1×1 shadow texture that always exists.

```
	public ShadowTextures GetRenderTextures(
		RenderGraph renderGraph,
		RenderGraphBuilder builder)
	{
		int atlasSize = (int)settings.directional.atlasSize;
		var desc = new TextureDesc(atlasSize, atlasSize)
		{
			depthBufferBits = DepthBits.Depth32,
			name = "Directional Shadow Atlas"
		};
		directionalAtlas = shadowedDirLightCount > 0 ?
			builder.WriteTexture(renderGraph.CreateTexture(desc)) :
			renderGraph.defaultResources.defaultShadowTexture;

		atlasSize = (int)settings.other.atlasSize;
		desc.width = desc.height = atlasSize;
		desc.name = "Other Shadow Atlas";
		otherAtlas = shadowedOtherLightCount > 0 ?
				builder.WriteTexture(renderGraph.CreateTexture(desc)) :
				renderGraph.defaultResources.defaultShadowTexture;
		return new ShadowTextures(directionalAtlas, otherAtlas);
	}
```

We can indicate that our depth textures are specifically for shadow maps, by setting the `isShadowMap` field of `TextureDesc` to `true`. That avoids the allocation of a stencil buffer.

```
		var desc = new TextureDesc(atlasSize, atlasSize)
		{
			depthBufferBits = DepthBits.Depth32,
			isShadowMap = true,
			name = "Directional Shadow Atlas"
		};
```

To make this work we have to set up lighting and shadows while recording the graph, not while executing it. So we remove the `RenderGraphContext` parameter from `Setup`.

```
	public void Setup(
		//RenderGraphContext context,
		CullingResults cullingResults, ShadowSettings settings)
	{
		//buffer = context.cmd;
		//this.context = context.renderContext;
		…
	}
```

And add it to `Render` instead.

```
	public void Render(RenderGraphContext context)
	{
		buffer = context.cmd;
		this.context = context.renderContext;
		…
	}
```

Also, we no longer have to deal with missing textures in `Render` and instead always set the global textures using the handles.

```
		if (shadowedDirLightCount > 0)
		{
			RenderDirectionalShadows();
		}
		//else { … }
		if (shadowedOtherLightCount > 0)
		{
			RenderOtherShadows();
		}
		//else { … }

		buffer.SetGlobalTexture(dirShadowAtlasId, directionalAtlas);
		buffer.SetGlobalTexture(otherShadowAtlasId, otherAtlas);
```

`RenderDirectionalShadows` and `RenderOtherShadows` must use the handles are as well, instead of getting temporary textures.

```
	void RenderDirectionalShadows()
	{
		…
		// buffer.GetTemporaryRT(…);
		buffer.SetRenderTarget(
			directionalAtlas,
			RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store);
		…
	}

	…

	void RenderOtherShadows()
	{
		…
		// buffer.GetTemporaryRT(…);
		buffer.SetRenderTarget(
			otherAtlas,
			RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store);
		…
	}
```

Now that there are no more textures to release up we can remove `Cleanup`.

```
	//public void Cleanup() { … }
```

1.2 Lighting

The next step is to adapt `Lighting`. We simplify its `Setup` method so it does not immediately render shadows. As rendering is delayed we have to keep track of `dirLightCount`, `otherLightCount`, and `useLightsPerObject` in fields.

```
	int dirLightCount, otherLightCount;

	bool useLightsPerObject;

	public void Setup(
		//RenderGraphContext context,
		CullingResults cullingResults, ShadowSettings shadowSettings,
		bool useLightsPerObject, int renderingLayerMask)
	{
		//buffer = context.cmd;
		this.cullingResults = cullingResults;
		this.useLightsPerObject = useLightsPerObject;
		shadows.Setup(//context,
			cullingResults, shadowSettings);
		SetupLights(//useLightsPerObject,
			renderingLayerMask);
		//shadows.Render();
		//context.renderContext.ExecuteCommandBuffer(buffer);
		//buffer.Clear();
	}
```

Both `dirLightCount` and `otherLightCount` are set to zero in `SetupLight` and the code that uses the buffer gets split into a new `Render` method. The removed shadow-rendering code gets reinserted at its end.

```
	void SetupLights(//bool useLightsPerObject,
		int renderingLayerMask)
	{
		…
		//int dirLightCount = 0, otherLightCount = 0;
		dirLightCount = otherLightCount = 0;
		…
	}

	public void Render(RenderGraphContext context)
	{
		CommandBuffer buffer = context.cmd;
		buffer.SetGlobalInt(dirLightCountId, dirLightCount);
		if (dirLightCount > 0) { … }
		…

		shadows.Render(context);
		context.renderContext.ExecuteCommandBuffer(buffer);
		buffer.Clear();
	}
```

We currently set the lights-per-object shader keyword in `SetupLights` directly, but it's better to do this via a command buffer, so remove that code.

```
		if (useLightsPerObject)
		{
			…
			//Shader.EnableKeyword(lightsPerObjectKeyword);
		}
		//else { … }
```

Instead we do it in `Render`, by invoking the buffer's `SetKeyword` method.

```
		CommandBuffer buffer = context.cmd;
		buffer.SetKeyword(lightsPerObjectKeyword, useLightsPerObject);
		buffer.SetGlobalInt(dirLightCountId, dirLightCount);
```

However, that method requires a `GlobalKeyword` parameter instead of `string`. So change our keyword to that, by invoking `GlobalKeyword.Create` with the string as an argument when initializing the static field. An added benefit of using `GlobalKeyword` is that the keyword name has to be used only once to look up its ID.

```
	static readonly GlobalKeyword lightsPerObjectKeyword =
		GlobalKeyword.Create("_LIGHTS_PER_OBJECT");
```

Can't we use the `GlobalKeyword` constructor method?

The constructor only looks up the ID and thus only works if the global keyword already exists. The `Create` method looks up the ID and also registers it if needed.

As `Lighting` encapsulates `Shadows` add a method to get the shadow textures, passing through the required arguments.

```
	public ShadowTextures GetShadowTextures(
		RenderGraph renderGraph, RenderGraphBuilder builder) =>
		shadows.GetRenderTextures(renderGraph, builder);
```

`Cleanup` only forwarded its invocation to `Shadows`, so this method can be removed.

```
	//public void Cleanup() { … }
```

1.3 Lighting Pass

Moving on to the passes, `LightingPass` can now construct its lighting object by itself and no longer needs to keep track of the other lighting data.

```
	readonly Lighting lighting = new();

	//CullingResults cullingResults;

	//ShadowSettings shadowSettings;

	//bool useLightsPerObject;

	//int renderingLayerMask;
```

`Render` now forwards to the lighting's `Render` method.

```
	void Render(RenderGraphContext context) => lighting.Render(context);
```

And `Record` no longer needs a lighting parameter, invokes `Setup` on the lighting data of the pass, and returns the shadow textures, using its builder.

```
	public ShadowTextures Record(
		RenderGraph renderGraph, //Lighting lighting,
		CullingResults cullingResults, ShadowSettings shadowSettings,
		bool useLightsPerObject, int renderingLayerMask)
	{
		using RenderGraphBuilder builder = renderGraph.AddRenderPass(
			sampler.name, out LightingPass pass, sampler);
		//pass.lighting = lighting;
		//pass.cullingResults = cullingResults;
		//pass.shadowSettings = shadowSettings;
		//pass.useLightsPerObject = useLightsPerObject;
		//pass.renderingLayerMask = renderingLayerMask;
		pass.lighting.Setup(cullingResults, shadowSettings,
			useLightsPerObject, renderingLayerMask);
		builder.SetRenderFunc<LightingPass>((pass, context) => pass.Render(context));
		return pass.lighting.GetShadowTextures(renderGraph, builder);
	}
```

This pass should not be culled if the shadow textures aren't used anywhere else, because it also sets up all GPU lighting data. So disable pass culling for it.

```
		builder.AllowPassCulling(false);
		return pass.lighting.GetShadowTextures(renderGraph, builder);
```

1.4 Geometry Pass

The shadow textures are read by `GeometryPass`, so add a `ShadowTextures` parameter to its `Record` method and indicate that both atlases are read from.

```
    public static void Record(
        …
        in CameraRendererTextures textures,
        in ShadowTextures shadowTextures)
    {
        …
        builder.ReadTexture(shadowTextures.directionalAtlas);
        builder.ReadTexture(shadowTextures.otherAtlas);

        builder.SetRenderFunc<GeometryPass>(
            (pass, context) => pass.Render(context));
    }
```

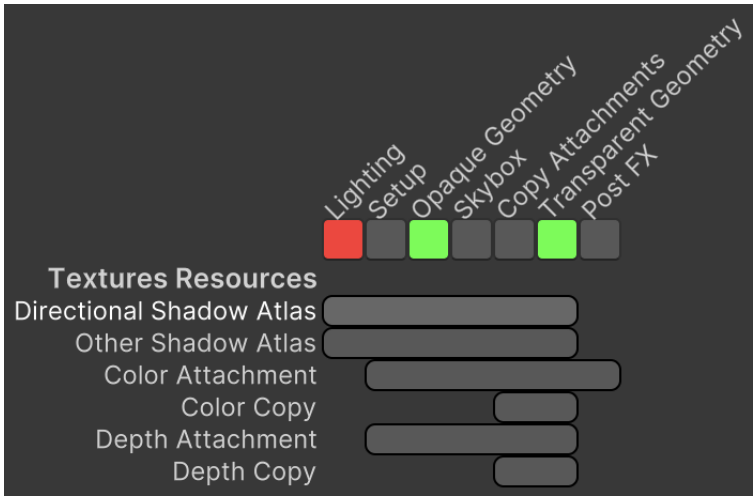
1.5 Camera Renderer

To tie it all together, adapt `CameraRenderer.Render` to work with the new approach. It gets the shadow textures from `LightingPass.Record` and no longer passes it the lighting object. It provides the shadows textures when recording both geometry passes. It also no longer has to clean up lighting after executing the graph.

```
		ShadowTextures shadowTextures = LightingPass.Record(
			renderGraph, //lighting,
			…);

		…

		GeometryPass.Record(…, shadowTextures);

		…

		GeometryPass.Record(…, shadowTextures);

		…

		//lighting.Cleanup();
```

Finally, we can get rid of the lighting data here.

```
	//readonly Lighting lighting = new();
```



Shadow texture usage.

2 Cleanup

We've transitioned to using graph-managed shadow textures, but let's also do a bit more code cleanup. I reformatted all C# code to enforce a maximum line width of 80 characters, but I won't show those changes. Besides that, two other things are worth mentioning.

2.1 Shadow Keywords

First, `Shadows` uses three arrays for setting global shader keywords. Let's change these to also work with `GlobalKeyword` instead of `string`.

```
	static readonly GlobalKeyword[] directionalFilterKeywords = {
		GlobalKeyword.Create("_DIRECTIONAL_PCF3"),
		GlobalKeyword.Create("_DIRECTIONAL_PCF5"),
		GlobalKeyword.Create("_DIRECTIONAL_PCF7"),
	};

	static readonly GlobalKeyword[] otherFilterKeywords = {
		GlobalKeyword.Create("_OTHER_PCF3"),
		GlobalKeyword.Create("_OTHER_PCF5"),
		GlobalKeyword.Create("_OTHER_PCF7"),
	};

	static readonly GlobalKeyword[] cascadeBlendKeywords = {
		GlobalKeyword.Create("_CASCADE_BLEND_SOFT"),
		GlobalKeyword.Create("_CASCADE_BLEND_DITHER"),
	};
```

The only other change needed is to adjust `SetKeywords` to match, also making use of `CommandBuffer.SetKeyword` to simplifies the code.

```
	void SetKeywords(GlobalKeyword[] keywords, int enabledIndex)
	{
		for (int i = 0; i < keywords.Length; i++)
		{
			buffer.SetKeyword(keywords[i], i == enabledIndex);
		}
	}
```

2.2 Merging Lighting Code

Second, as only `LightingPass` uses `Lighting` let's merge the latter into the former, because that's where the code belongs now. This requires `LightingPass` to also use the `Unity.Collections` and `UnityEngine` namespaces.

```
using Unity.Collections;
using UnityEngine;
using UnityEngine.Experimental.Rendering.RenderGraphModule;
using UnityEngine.Rendering;
```

Delete its `lighting` object field and its `Render` method.

```
	//readonly Lighting lighting = new();

	…

	//void Render(RenderGraphContext context) => lighting.Render(context);
```

Then copy everything except `GetShadowTextures` from `Lighting` to `LightingPass` and delete the `Lighting` script asset. The only change needed for the copied code is making `Render` private.

```
	//public
	void Render(RenderGraphContext context) { … }
```

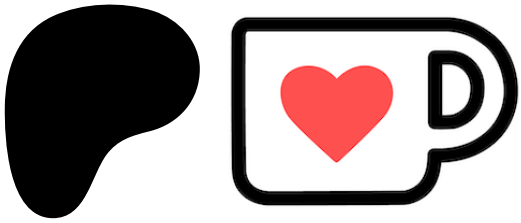
The last step is to remove the indirection via `lighting` from `Record`.

```
		//pass.lighting.Setup
		pass.Setup(cullingResults, shadowSettings,
			useLightsPerObject, renderingLayerMask);
		builder.SetRenderFunc<LightingPass>(
			(pass, context) => pass.Render(context));
		builder.AllowPassCulling(false);
		return pass.shadows.GetRenderTextures(renderGraph, builder);
```

The next tutorial is Custom SRP 2.4.0.

Enjoying the tutorials? Are they useful?

Please support me on Patreon or Ko-fi!



Or make a direct donation!

made by Jasper Flick