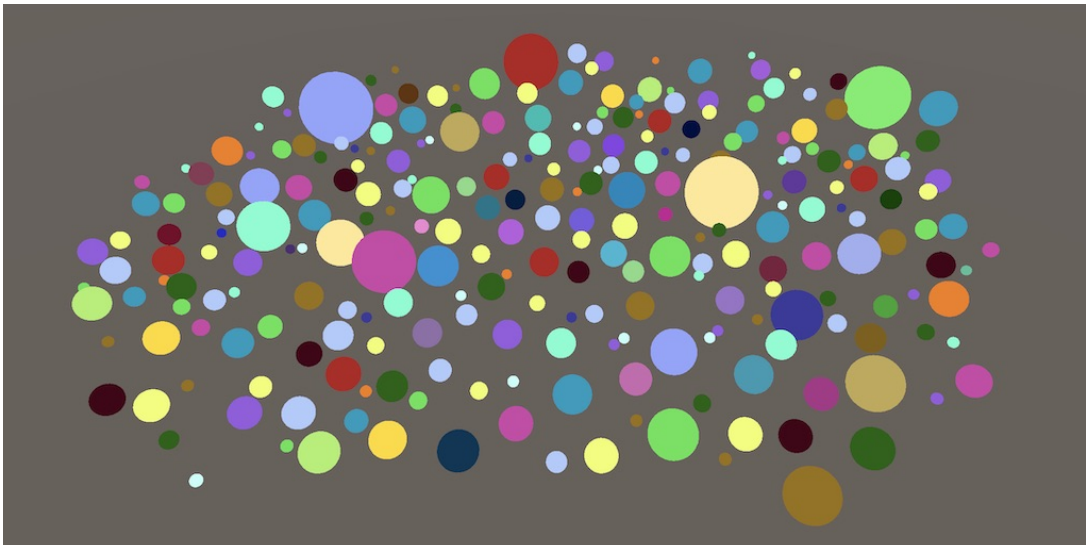
Custom Shaders HLSL and Core Library

Write an HLSL shader.
Define constant buffers.
Use the Render Pipeline Core Library.
Support dynamic batching and GPU instancing.

This is the second installment of a tutorial series covering Unity's scriptable render pipeline. It's about creating a shader using HLSL and efficiently rendering multiple objects by batching them in a single draw call.

This tutorial is made with Unity 2018.3.0f2.



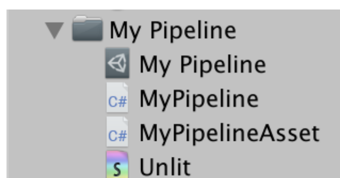
256 spheres, a single draw call.

1 Custom Unlit Shader

Although we've used the default unlit shader to test our pipeline, taking full advantage of a nontrivial custom pipeline requires the creation of custom shaders to work with it. So we're going to create a shader of our own, replacing Unity's default unlit shader.

1.1 Creating a Shader

A shader asset can be created via one of the options in the *Assets / Create / Shader* menu. The *Unlit Shader* is most appropriate, but we're going to start fresh, by deleting all the default code from the created shader file. Name the asset *Unlit*.

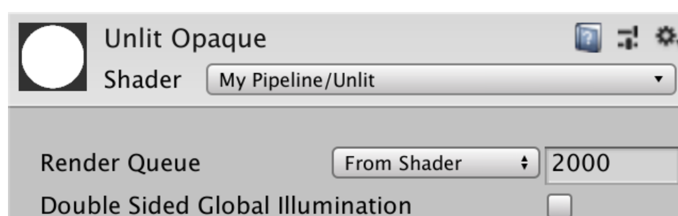


Unlit shader asset.

The fundamentals of shader files are explained in Rendering 2, Shader Fundamentals. Give it a read if you're unfamiliar with writing shaders so you know the basics. The minimum to get a working shader is to define a `Shader` block with a `Properties` block plus a `SubShader` block with a `Pass` block inside it. Unity will turn that into a default white unlit shader. After the `Shader` keyword comes a string that will be used in the shader dropdown menu for materials. We'll use *My Pipeline/Unlit* for it.

```
Shader "My Pipeline/Unlit" {

	Properties {}

	SubShader {

		Pass {}
	}
}
```

Adjust the *Unlit Opaque* material so it uses our new shader, which will turn it white, if it weren't already.



Unlit opaque material with custom shader.

1.2 HLSL

To write our own shader, we have to put a program inside its `Pass` block. Unity supports either GLSL or HLSL programs. While GLSL is used in the default shader and also in Rendering 2, Shader Fundamentals, Unity's new rendering pipeline shaders use HLSL, so we'll use that for our pipeline too. That means that we have to put all our code in between an `HLSLPROGRAM` and an `ENDHLSL` statement.

```
		Pass {
			HLSLPROGRAM
			
			ENDHLSL
		}
```

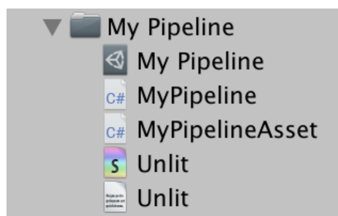
What's the difference between GLSL and HLSL programs?

In practice, Unity uses virtually the same syntax for both and takes care of converting to the appropriate shader code per build target. The biggest difference is that GLSL programs include some code by default. HLSL programs don't do anything implicitly, requiring us to include anything that we need explicitly. That's fine, because the old GLSL include files are weighed down by old and obsolete code. We'll rely on newer HLSL include files instead.

At minimum, a Unity shader requires a vertex program and a fragment program function, each defined with a pragma compiler directive. We'll use `UnlitPassVertex` for the vertex function and `UnlitPassFragment` for the other. But we won't put the code for these functions in the shader file directly. Instead, we'll put the HLSL code in a separate include file, which we'll also name *Unlit*, but with the *hlsl* extension. Put it in the same folder as *Unlit.shader* and then include it the HLSL program, after the pragma directives.

```
			HLSLPROGRAM
			
			#pragma vertex UnlitPassVertex
			#pragma fragment UnlitPassFragment
			
			#include "Unlit.hlsl"
			
			ENDHLSL
```

Unfortunately, Unity doesn't have a convenient menu item for the creation of an HLSL include file asset. You'll have to create it yourself, for example by duplicating the *Unlit.shader* file, changing its file extension to *hlsl* and removing the shader code from it.



Unlit HLSL include file asset.

Inside the include file, begin with an include guard to prevent duplicating code in case the files gets included more that once. While that should never happen, it's good practice to always do this for every include file.

```
#ifndef MYRP_UNLIT_INCLUDED
#define MYRP_UNLIT_INCLUDED

#endif // MYRP_UNLIT_INCLUDED
```

At minimum, we have to know the vertex position in the vertex program, which has to output a homogeneous clip-space position. So we'll define an input and an output structure for the vertex program, both with a single `float4` position.

```
#ifndef MYRP_UNLIT_INCLUDED
#define MYRP_UNLIT_INCLUDED

struct VertexInput {
	float4 pos : POSITION;
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
};

#endif // MYRP_UNLIT_INCLUDED
```

Next, we'll define the vertex program function, `UnlitPassVertex`. For now, we'll directly use the object-space vertex position as the clip-space position. That is incorrect, but is the quickest way to get a compiling shader. We'll add the correct space conversion later.

```
struct VertexOutput {
	float4 clipPos : SV_POSITION;
};

VertexOutput UnlitPassVertex (VertexInput input) {
	VertexOutput output;
	output.clipPos = input.pos;
	return output;
}

#endif // MYRP_UNLIT_INCLUDED
```

We keep the default white color for now, so our fragment program function can simply return 1 as a `float4`. It receives the interpolated vertex output as its input, so add that as a parameter, even though we don't use it yet.

```
VertexOutput UnlitPassVertex (VertexInput input) {
    VertexOutput output;
    output.clipPos = input.pos;
    return output;
}

float4 UnlitPassFragment (VertexOutput input) : SV_TARGET {
    return 1;
}

#endif // MYRP_UNLIT_INCLUDED
```

Should we use `half` or `float`?

Most mobile GPUs support both precision types, `half` being more efficient. So if you're optimizing for mobiles it makes sense to use `half` as much as possible. The rule is to use `float` for positions and texture coordinate only and `half` for everything else, provided that the results are acceptable.

When not targeting mobile platforms, precision isn't an issue because the GPU always uses `float`, even if we write `half`. I'll consistently use `float` in this tutorial series.

There's also the `fixed` type, but it's only really supported by old hardware that you wouldn't target for modern apps. It's usually equivalent to `half`.

1.3 Transformation Matrices

At this point we have a compiling shader, although it doesn't produce sensible results yet. The next step is to convert the vertex position to the correct space. If we had a model-view-projection matrix then we could convert directly from object space to clip space, but Unity doesn't create such a matrix for us. It does make the model matrix available, which we can use to convert from object space to world space. Unity expects our shader to have a `float4x4 unity_ObjectToWorld` variable to store the matrix. As we're working with HLSL, we have to define that variable ourselves. Then use it to convert to world space in the vertex function and use that for its output.

```
float4x4 unity_ObjectToWorld;

struct VertexInput {
	float4 pos : POSITION;
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
};

VertexOutput UnlitPassVertex (VertexInput input) {
	VertexOutput output;
	float4 worldPos = mul(unity_ObjectToWorld, input.pos);
	output.clipPos = worldPos;
	return output;
}
```

Next, we need to convert from world space to clip space. That's done with a view-projection matrix, which Unity makes available via a `float4x4 unity_MatrixVP` variable. Add it and then complete the conversion.

```
float4x4 unity_MatrixVP;
float4x4 unity_ObjectToWorld;

…

VertexOutput UnlitPassVertex (VertexInput input) {
	VertexOutput output;
	float4 worldPos = mul(unity_ObjectToWorld, input.pos);
	output.clipPos = mul(unity_MatrixVP, worldPos);
	return output;
}
```

I changed the code, but it's still not working?

When editing include files, Unity doesn't always respond to a change and fails to refresh the shaders. When that happens, try again by saving the file once more, if necessary with a small change that you can later undo.

Our shader now works correctly. All objects that use the unlit material are once again visible, fully white. But our conversion isn't as efficient as it could be, because it's performing a full matrix multiplication with a 4D position vector. The fourth component of the position is always 1. By making that explicit we make it possible for the compiler to optimize the computation.

```
	float4 worldPos = mul(unity_ObjectToWorld, float4(input.pos.xyz, 1.0));
```

Is the optimization meaningful?

It's an optimization that Unity does itself and was pretty keen on upgrading all shaders of everyone to use it. It's the difference between a mad and an add instruction. Whether that is a noticeable difference depends on the platform. In any case, it can only be faster, never slower. Here's the generated D3D11 code for the space conversion without the optimization:

```
0: mul r0.xyzw, v0.yyyy, cb1[1].xyzw
1: mad r0.xyzw, cb1[0].xyzw, v0.xxxx, r0.xyzw
2: mad r0.xyzw, cb1[2].xyzw, v0.zzzz, r0.xyzw
3: mad r0.xyzw, cb1[3].xyzw, v0.wwww, r0.xyzw
4: mul r1.xyzw, r0.yyyy, cb0[1].xyzw
5: mad r1.xyzw, cb0[0].xyzw, r0.xxxx, r1.xyzw
6: mad r1.xyzw, cb0[2].xyzw, r0.zzzz, r1.xyzw
7: mad o0.xyzw, cb0[3].xyzw, r0.wwww, r1.xyzw
```

And here is the conversion with the optimization:

```
0: mul r0.xyzw, v0.yyyy, cb1[1].xyzw
1: mad r0.xyzw, cb1[0].xyzw, v0.xxxx, r0.xyzw
2: mad r0.xyzw, cb1[2].xyzw, v0.zzzz, r0.xyzw
3: add r0.xyzw, r0.xyzw, cb1[3].xyzw
4: mul r1.xyzw, r0.yyyy, cb0[1].xyzw
5: mad r1.xyzw, cb0[0].xyzw, r0.xxxx, r1.xyzw
6: mad r1.xyzw, cb0[2].xyzw, r0.zzzz, r1.xyzw
7: mad o0.xyzw, cb0[3].xyzw, r0.wwww, r1.xyzw
```

1.4 Constant Buffers

Unity doesn't provide us with a model-view-projection matrix, because that way a matrix multiplication of the M and VP matrices can be avoided. Besides that, the VP matrix can be reused for everything that gets drawn with the same camera during a frame. Unity's shaders takes advantage of that fact and put the matrices in different constant buffers. Although we define them as variables, their data remains constant during the drawing of a single shape, and often longer than that. The VP matrix gets put in a per-frame buffer, while the M matrix gets put in a per-draw buffer.

While it is not strictly required to put shader variables in constant buffers, doing so makes it possible for all data in the same buffer to be changed more efficiently. At least, that's the case when it is supported by the graphics API. OpenGL doesn't.

To be as efficient as possible, we'll also make use of constant buffers. Unity puts the VP matrix in a `UnityPerFrame` buffer and the M matrix in a `UnityPerDraw` buffer. There's more data that gets put in these buffers, but we don't need it yet so there is no need to include it. A constant buffer is defined like a struct, except with the `cbuffer` keyword and the variables remain accessible as before.

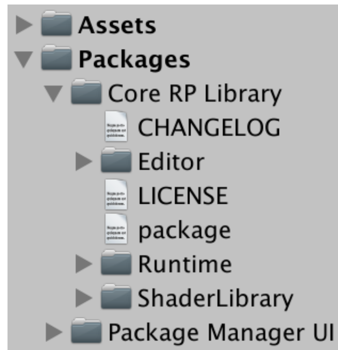
```
cbuffer UnityPerFrame {
	float4x4 unity_MatrixVP;
};

cbuffer UnityPerDraw {
	float4x4 unity_ObjectToWorld;
}
```

1.5 Core Library

Because constant buffers don't benefit all platforms, Unity's shaders rely on macros to only use them when needed. The `CBUFFER_START` macro with a name parameter is used instead of directly writing `cbuffer` and an accompanying `CBUFFER_END` macro replaces the end of the buffer. Let's use that approach as well.

```
CBUFFER_START(UnityPerFrame)
	float4x4 unity_MatrixVP;
CBUFFER_END

CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
CBUFFER_END
```

That results in a compiler error, because those two macros are not defined. Rather than figure out when it is appropriate to use constant buffers and define the macros ourselves, we'll make use of Unity's core library for render pipelines. It can be added to our project via the package manager window. Switch to the *All Packages* list and enable *Show preview packages* under *Advanced*, then select *Render-pipelines.core*, and install it. I'm using version 4.6.0-preview, the highest version that works in Unity 2018.3.



Render Pipeline Core Library installed.

Now we can include the common library functionality, which we can access via *Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl*. It defines multiple useful functions and macros, along with the constant buffer macros, so include it before using them.

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

CBUFFER_START(UnityPerFrame)
float4x4 unity_MatrixVP;
CBUFFER_END
```

How do those macros work, exactly?

You can see that by opening the *Common.hlsl* file in the core library package. It ends up including an API-specific include file from its *API* subfolder, which defines the macros.

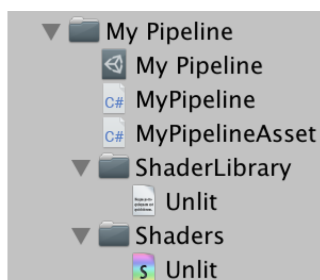
1.6 Compilation Target Level

Our shader works again, at least for most platforms. After including the library, our shader fails to compile for OpenGL ES 2. That happens because by default Unity uses a shader compiler for OpenGL ES 2 that doesn't work with the core library. We can fix that by adding `#pragma prefer_hlslcc gles` to our shader, which is what Unity does for its shaders in the Lightweight render pipeline. However, instead of doing that we simply won't support OpenGL ES 2 at all, as it's only relevant when targeting old mobile devices. We do that by using the `#pragma target` directive to target shader level 3.5 instead of the default level, which is 2.5.

```
			#pragma target 3.5

			#pragma vertex UnlitPassVertex
			#pragma fragment UnlitPassFragment
```

1.7 Folder Structure

Note that all the HLSL include files of the core library are located *ShaderLibrary* folder. Let's do that too, so put *Unlit.hlsl* in a new *ShaderLibrary* folder inside *My Pipeline*. Put the shader in a separate *Shader* folder too.



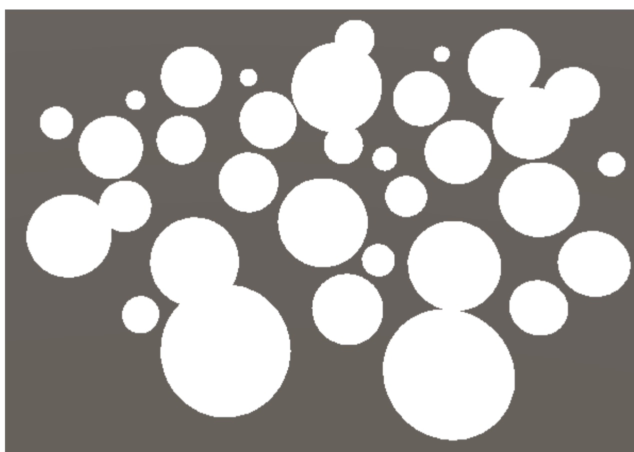
My Pipeline folder structure.

To keep our shader intact while still relying on relative include paths, we'll have to change our include statement from *Unlit.hlsl* to *../ShaderLibrary/Unlit.hlsl*.

```
			#include "../ShaderLibrary/Unlit.hlsl"
```

2 Dynamic Batching

Now that we have a minimal custom shader we can use it to further investigate how our pipeline renders things. A big question is how efficient it can render. We'll test that by filling the scene with a bunch of spheres that use our unlit material. You could use thousands, but a few dozen also gets the message across. They can have different transformations, but keep their scales uniform, meaning that each scale's X, Y, and Z components are always equal.



A bunch of white spheres.

When investigating how the scene is drawn via the frame debugger, you'll notice that every sphere requires its own separate draw call. That isn't very efficient, as each draw call introduces overhead as the CPU and GPU need to communicate. Ideally, multiple spheres get drawn together with a single call. While that's possible, it currently doesn't happen. The frame debugger gives us a hint about it when you select one of the draw calls.

Why this draw call can't be batched with the previous one

Dynamic Batching is turned off in the Player Settings or is disabled temporarily in the current context to avoid z-fighting.

No dynamic batching.

2.1 Enabling Batching

The frame debugger tells us that dynamic batching isn't used, because it's either turned off or because depth sorting interferes with it. If you check the player settings, then you'll see that indeed the *Dynamic Batching* option is disabled. However, enabling it has no effect. That's because the player setting applies to Unity's default pipeline, not our custom one.

To enable dynamic batching for our pipeline, we have to indicate that it is allowed when drawing in `MyPipeline.Render`. The draw settings contain a `flags` field that we have to set to `DrawRendererFlags.EnableDynamicBatching`.

```
		var drawSettings = new DrawRendererSettings(
			camera, new ShaderPassName("SRPDefaultUnlit")
		);
		drawSettings.flags = DrawRendererFlags.EnableDynamicBatching;
		drawSettings.sorting.flags = SortFlags.CommonOpaque;
```

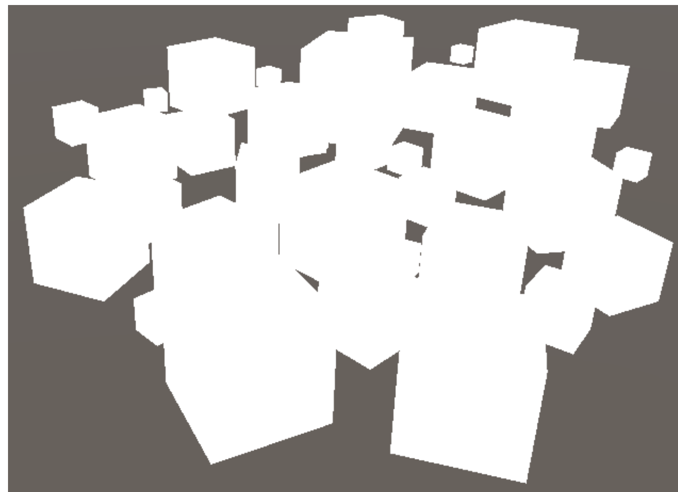
After that change we still don't get dynamic batching, but the reason has changed. Dynamic batching means that Unity merges objects together in a single mesh before they are drawn. That requires CPU time each frame and to keep that in check it's limited to small meshes only.

Why this draw call can't be batched with the previous one

A submesh we are trying to dynamic-batch has more than 300 vertices.

Too many vertices to batch.

The sphere mesh is too big, but cubes are small and will work. So adjust all objects to use the cube mesh instead. You can select them all and adjust their mesh filter in one go.



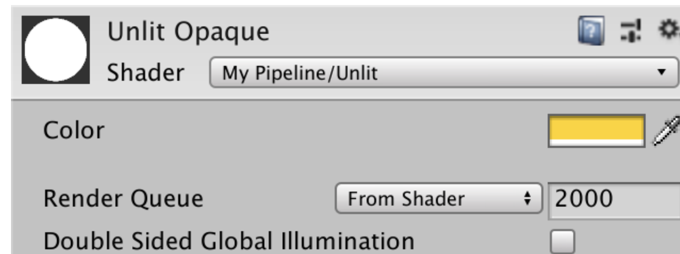
▼ Render Camera	3
Clear (Z+stencil)	
▼ RenderLoop.Draw	1
Dynamic Batch	
▼ Camera.RenderSkybox	1
Draw Mesh	

Cubes drawn in a single dynamic batch.

2.2 Colors

Dynamic batching works for small meshes that are all drawn with the same material. But when multiple materials are involved things get more complicated. To illustrate this, we'll make it possible to change the color of our unlit shader. Add a color property to its `Properties` block named `_Color`, with *Color* as its label, using white as the default.

```
	Properties {
		_Color ("Color", Color) = (1, 1, 1, 1)
	}
```

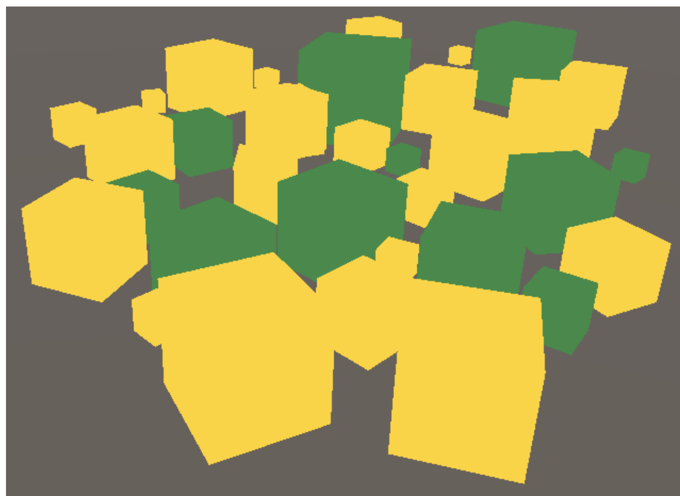


Material with adjusted color.

Now we can adjust the color of our material, but it doesn't affect what gets drawn yet. Add a `float4 _Color` variable to our include file and return that instead of the fixed value in `UnlitPassFragment`. The color is defined per material, so can be put in a constant buffer that only needs to change when materials are switched. We'll name the buffer `UnityPerMaterial`, just like Unity does.

```
CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
CBUFFER_END

CBUFFER_START(UnityPerMaterial)
	float4 _Color;
CBUFFER_END

struct VertexInput {
	float4 pos : POSITION;
};

…

float4 UnlitPassFragment (VertexOutput input) : SV_TARGET {
	return _Color;
}
```

Duplicate our material and set both to use different colors, so we can distinguish them. Select a few objects and have them use the new material, so you end up with a mix.



Render Camera 6
Clear (Z+stencil)
RenderLoop.Draw 4
Dynamic Batch
Dynamic Batch
Dynamic Batch
Dynamic Batch
Camera.RenderSkybox 1

Two materials, four batches.

Dynamic batching still happens, but we end up with multiple batches. There will be at least one batch per material, because each requires different per-material data. But there'll often be more batches because Unity prefers to group objects spatially to reduce overdraw.

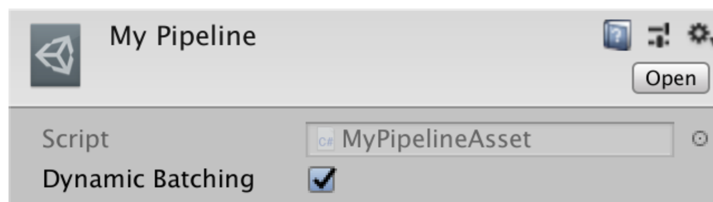
Why this draw call can't be batched with the previous one
Objects have different materials.

No batching because of different materials.

2.3 Optional Batching

Dynamic batching can be a benefit, but it can also end up not making much of a difference, or even slow things down. If your scene doesn't contain lots of small meshes that share the same material, it might make sense to disable dynamic batching so Unity doesn't have to figure out whether to use it or not each frame. So we'll add an option to enable dynamic batching to our pipeline. We cannot rely on the player settings. Instead, we add a toggle configuration option to `MyPipelineAsset`, so we can configure it via our pipeline asset in the editor.

```
	[SerializeField]
	bool dynamicBatching;
```



Dynamic batching enabled.

When the `MyPipeline` instance is created, we have to tell it whether to use dynamic batching or not. We'll provide this information as an argument when invoking its constructor.

```
	protected override IRenderPipeline InternalCreatePipeline () {
		return new MyPipeline(dynamicBatching);
	}
```

To make that work, we can no longer rely on the default constructor of `MyPipeline`. Give it a public constructor method, with a boolean parameter to control dynamic batching. We'll setup the drawn flags once in the constructor and keep track of them in a field.

```
	DrawRendererFlags drawFlags;

	public MyPipeline (bool dynamicBatching) {
		if (dynamicBatching) {
			drawFlags = DrawRendererFlags.EnableDynamicBatching;
		}
	}
```

Copy the flags to the draw settings in `Render`.

```
		drawSettings.flags = drawFlags;
```

Note that when we toggle the *Dynamic Batching* option of our asset in the editor, the batching behavior of Unity immediately changes. Each time we adjust the asset a new pipeline instance gets created.

3 GPU Instancing

Dynamic batching is not the only way in which we can reduce the number of draw calls per frame. Another approach is to use GPU instancing. In the case of instancing, the CPU tells the GPU to draw a specific mesh-material combination more than once via a single draw call. That makes it possible to group objects that use the same mesh and material without having to construct a new mesh. That also removes the limit on the mesh size.

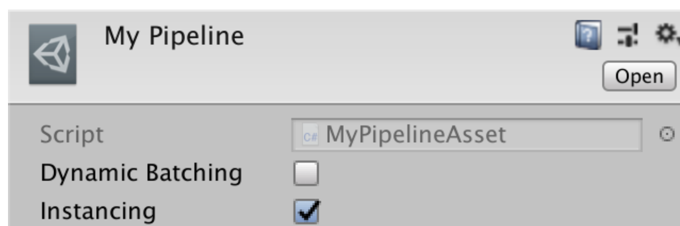
3.1 Optional Instancing

GPU instancing is enabled by default, but we overrode that with our custom draw flags. Let's make GPU instancing optional too, which makes it easy to compare the results with and without it. Add another toggle to `MyPipelineAsset` and pass it to the constructor invocation.

```
	[SerializeField]
	bool instancing;
	
	protected override IRenderPipeline InternalCreatePipeline () {
		return new MyPipeline(dynamicBatching, instancing);
	}
```

In the `MyPipeline` constructor method, also set the flags for instancing after doing so for dynamic batching. In this case the flags value is `DrawRendererFlags.EnableInstancing` and we boolean-OR it into the flags, so both dynamic batching and instancing can be enabled at the same time. When they're both enabled Unity prefers instancing over batching.

```
	public MyPipeline (bool dynamicBatching, bool instancing) {
		if (dynamicBatching) {
			drawFlags = DrawRendererFlags.EnableDynamicBatching;
		}
		if (instancing) {
			drawFlags |= DrawRendererFlags.EnableInstancing;
		}
	}
```



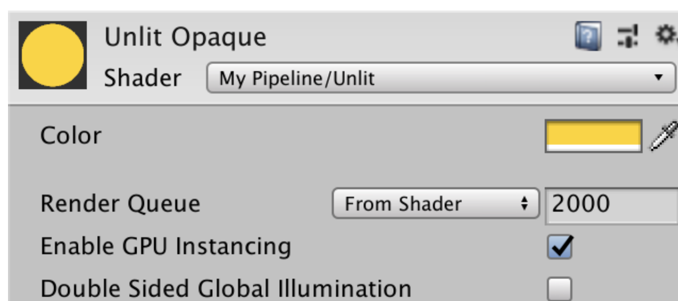
Instancing enabled, dynamic batching disabled.

3.2 Material Support

That GPU instancing is enabled for our pipeline doesn't mean that objects are automatically instanced. It has to be supported by the material that they're using. Because instancing isn't always needed, it is optional, which requires two shader variants: one that supports instancing and one that doesn't. We can create all required variants by adding the `#pragma multi_compile_instancing` directive to our shader. In our case, that produces two shader variants, one with and one without the `INSTANCING_ON` keyword defined.

```
			#pragma target 3.5

			#pragma multi_compile_instancing

			#pragma vertex UnlitPassVertex
			#pragma fragment UnlitPassFragment
```

That change also makes a new material configuration option appear for our material: *Enable GPU Instancing*.



Material with instancing enabled.

3.3 Shader Support

When instancing is enabled, the GPU is told to draw the same mesh multiple times with the same constant data. But the M matrix is part of that data. That means that we end up with the same mesh rendered multiple times the exact same way. To get around that problem, an array containing the M matrices of all objects has to be put in a constant buffer. Each instance gets drawn with its own index, which can be used to retrieve the correct M matrix from the array.

We must now either use `unity_ObjectToWorld` when not instancing, or a matrix array when we are instancing. To keep the code in `UnlitPassVertex` the same for both cases, we'll define a macro for the matrix, specifically `UNITY_MATRIX_M`. We use that macro name, because the core library has an include file that defines macros to support instancing for us, and it also redefines `UNITY_MATRIX_M` to use the matrix array when needed.

```
CBUFFER_START(UnityPerDraw)
	float4x4 unity_ObjectToWorld;
CBUFFER_END

#define UNITY_MATRIX_M unity_ObjectToWorld

…

VertexOutput UnlitPassVertex (VertexInput input) {
	VertexOutput output;
	float4 worldPos = mul(UNITY_MATRIX_M, float4(input.pos.xyz, 1.0));
	output.clipPos = mul(unity_MatrixVP, worldPos);
	return output;
}
```

The include file is *UnityInstancing.hlsl*, and because it might redefine `UNITY_MATRIX_M` we have to include it after defining that macro ourselves.

```
#define UNITY_MATRIX_M unity_ObjectToWorld

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

When using instancing, the index of the object that's currently being drawn is added it its vertex data by the GPU. The `UNITY_MATRIX_M` relies on the index, so we have to add it to the `VertexInput` structure. We can use the `UNITY_VERTEX_INPUT_INSTANCE_ID` macro for that.

```
struct VertexInput {
	float4 pos : POSITION;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Finally, we have to make the index available before using `UNITY_MATRIX_M` in `UnlitPassVertex`, via the `UNITY_SETUP_INSTANCE_ID` macro, providing the input as an argument.

```
VertexOutput UnlitPassVertex (VertexInput input) {
	VertexOutput output;
	UNITY_SETUP_INSTANCE_ID(input);
	float4 worldPos = mul(UNITY_MATRIX_M, float4(input.pos.xyz, 1.0));
	output.clipPos = mul(unity_MatrixVP, worldPos);
	return output;
}
```

Our cubes now get instanced. Just like with dynamic batching, we end up with multiple batches because we're using different materials. Make sure that all materials used have GPU instancing enabled.

Render Camera 6
Clear (Z+stencil)
RenderLoop.Draw 4
Draw Mesh (instanced) Cube
Draw Mesh (instanced) Cube
Draw Mesh (instanced) Cube
Draw Mesh (instanced) Cube
Camera.RenderSkybox 1

Four instanced draw calls.

Besides the object-to-world matrices, by default world-to-object matrices are put in the instancing buffer too. Those are the inverse of the M matrices, which are needed for normal vectors when using non-uniform scales. But we're only using uniform scales, so we don't need those additional matrices. We can inform Unity about that, by adding the `#pragma instancing_options assumeuniformscaling` directive to our shader.

```
			#pragma multi_compile_instancing
			#pragma instancing_options assumeuniformscaling
```

If you do need to support non-uniform scaling, then you'll have to use a shader that doesn't have this option enabled.

3.4 Many Colors

If we wanted to include more colors in our scene we would need to make more materials, which means we end up with more batches. But if the matrices can be put in arrays, is should be possible to do the same for colors. Then we could combine objects with different colors in a single batch. With a little work, that can indeed be done.

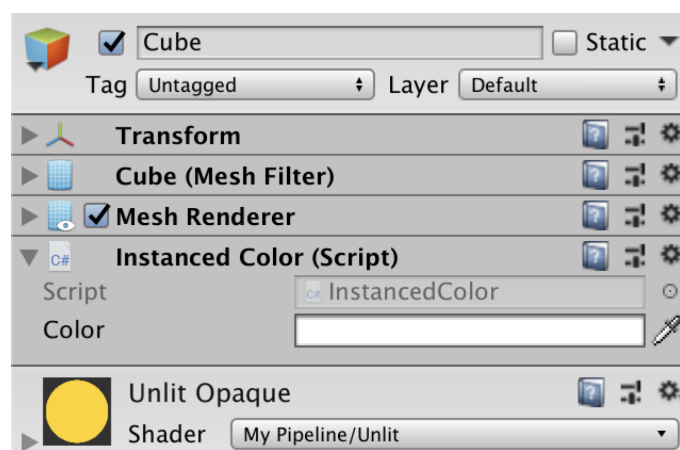
The first step of support a unique color per object is to make it possible to set the color for each individually. We cannot do that via the material, because that's an asset that the objects all share. Let's create a component for it, naming it `InstancedColor`, giving it a single configurable color field. As it's not specific to our pipeline, keep its script file outside the *My Pipeline* folder.

```
using UnityEngine;

public class InstancedColor : MonoBehaviour {

	[SerializeField]
	Color color = Color.white;
}
```

To override the material's color, we have to provide the object's renderer component with a material property block. Do that by creating a new `MaterialPropertyBlock` object instance, give it a *_Color* property via its `SetColor` method, then pass it to the object's `MeshRenderer` component, by invoking its `SetPropertyBlock` method. We assume that the colors remain constant while in play mode, so do this in the `Awake` method of our class.

```
	void Awake () {
		var propertyBlock = new MaterialPropertyBlock();
		propertyBlock.SetColor("_Color", color);
		GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);
	}
```

Add our component to one of the objects in the scene. You'll see that its color changes, but only after we enter play mode.



Cube with color component.

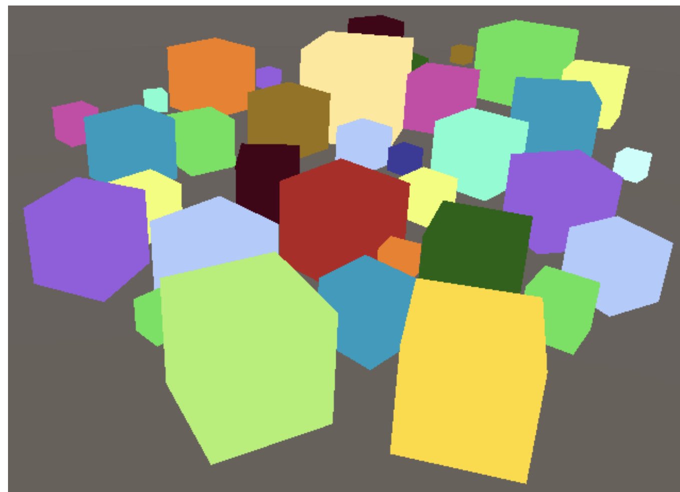
To immediately see the color changes in the scene while in edit mode, move the code that sets the color to an `OnValidate` method. The `Awake` method can then simply invoke `OnValidate` so we don't need to duplicate code.

```
	void Awake () {
		OnValidate();
	}

	void OnValidate () {
		var propertyBlock = new MaterialPropertyBlock();
		propertyBlock.SetColor("_Color", color);
		GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);
	}
```

When is `OnValidate` invoked?

`OnValidate` is a special Unity message method. It gets invoked while in edit mode, when the component is loaded or changed. So each time the scene is loaded and when we edit the component. Thus, the individual colors appear immediately.

Add the component to all shapes, by selecting them all and adding it once, but make sure not to add it a second time to the object that already has one. Also have them all use the same material. Alternative materials can be removed, because we configure the color per object.



Many colors, one material.

Note that we're creating a new `MaterialPropertyBlock` instance each time we set a color override. That isn't necessary, because each mesh renderer internally keeps track of the overridden properties, copying them from the property block. That means that we can reuse it, so keep track of a single static block, creating it only when needed.

```
	static MaterialPropertyBlock propertyBlock;

	…

	void OnValidate () {
		if (propertyBlock == null) {
			propertyBlock = new MaterialPropertyBlock();
		}
		propertyBlock.SetColor("_Color", color);
		GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);
	}
```

Also, we can slightly speed up the matching of the color property by prefetching its property ID via the `Shader.PropertyToID` method. Each shader property name gets a global identifier integer. These identifiers are subject to change, but always remain constant during a single session, meaning between plays and compilation. So we fetch it once, which can be done as the default value for a static field.

```
	static int colorID = Shader.PropertyToID("_Color");

	…

	void OnValidate () {
		if (propertyBlock == null) {
			propertyBlock = new MaterialPropertyBlock();
		}
		propertyBlock.SetColor(colorID, color);
		GetComponent<MeshRenderer>().SetPropertyBlock(propertyBlock);
	}
```

3.5 Per-Instance Colors

Overriding the color per object caused GPU instancing to break. Although we're using a single material, what matters is the data used for rendering. As we've overridden the color per object, we have forced them to be drawn separately.

Why this draw call can't be batched with the previous one
Non-instanced properties set for instanced shader.

Not instanced because of color differences.

The idea was to put the color data in an array, which will make instancing work again. Our `_Color` property has to be given the same treatment as the M matrix. In this case we have to be explicit, as the core library doesn't redefine a macro for arbitrary properties. Instead, we manually create a constant buffer for the purpose of instancing, via the `UNITY_INSTANCING_BUFFER_START` and accompanying ending macro, naming it `PerInstance` to keep our naming scheme consistent. Inside the buffer, we define the color as `UNITY_DEFINE_INSTANCED_PROP(float4, _Color)`. When instancing isn't used that ends up equal to `float4 _Color`, but otherwise we end up with an array of instance data.

```
//CBUFFER_START(UnityPerMaterial)
	//float4 _Color;
//CBUFFER_END

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

To deal with the two possible ways in which the color can now be defined, we have to access it via the `UNITY_ACCESS_INSTANCED_PROP` macro, passing it our buffer and the name of the property.

```
float4 UnlitPassFragment (VertexOutput input) : SV_TARGET {
	return UNITY_ACCESS_INSTANCED_PROP(PerInstance, _Color);
}
```

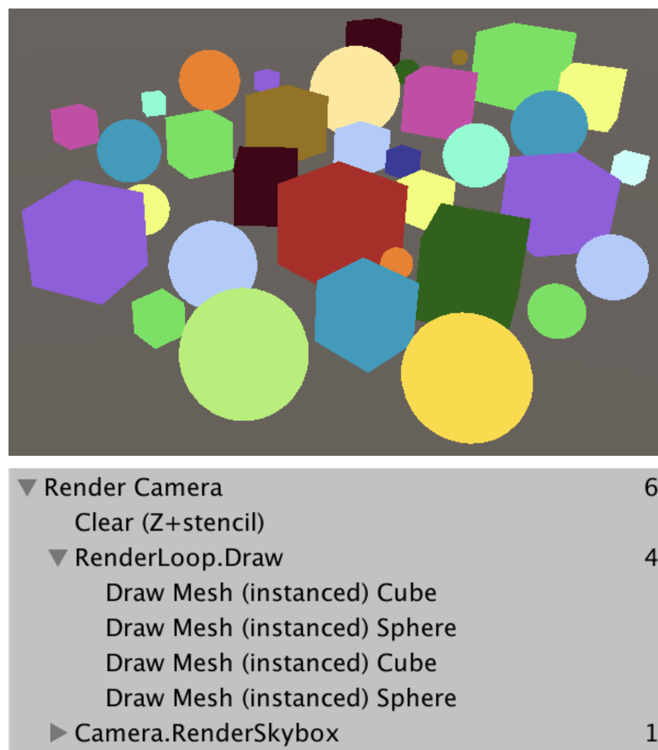
Now the instance index must also be made available in `UnlitPassFragment`. So add `UNITY_VERTEX_INPUT_INSTANCE_ID` to `VertexOutput`, then use `UNITY_SETUP_INSTANCE_ID` in `UnlitPassFragment` like we did in `UnlitPassVertex`. To make that work, we have to copy the index from the vertex input to the vertex output, for which we can use the `UNITY_TRANSFER_INSTANCE_ID` macro.

```
struct VertexInput {
	float4 pos : POSITION;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct VertexOutput {
	float4 clipPos : SV_POSITION;
	UNITY_VERTEX_INPUT_INSTANCE_ID
};

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

float4 UnlitPassFragment (VertexOutput input) : SV_TARGET {
	UNITY_SETUP_INSTANCE_ID(input);
	return UNITY_ACCESS_INSTANCED_PROP(PerInstance, _Color);
}
```

Render Camera 3
Clear (Z+stencil)
RenderLoop.Draw 1
Draw Mesh (instanced) Cube
Camera.RenderSkybox 1

Many colors, one draw.

All object now end up combined in a single draw call, even if they all use a different color. However, there is a limit to how much data can be put in the constant buffers. The maximum instance batch size depends on how much data we vary per instance. Besides that, the buffer maximum varies per platform. And we're still limited to using the same mesh and material. For example, mixing cubes and spheres will split up the batches.



Both cubes and spheres.

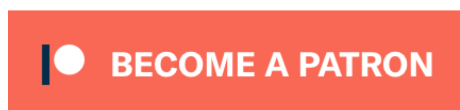
At this point we have a minimal shader that can be used to drawn many objects as efficiently as possible. In the future, we'll build on this foundation to create more advanced shaders.

The next tutorial is Lights.

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!



Or make a direct donation!

made by Jasper Flick