Custom Pipeline Taking Control of Rendering

Create a pipeline asset and instance.
*Cull, filter, sort, render.*
Keep memory clean.
Provide a good editing experience.

This is the first installment of a tutorial series covering Unity's scriptable render pipeline. This tutorial assumes you've gone through the Basics series first, and the Procedural Grid tutorial. The first few parts of the Rendering series are also useful.

This tutorial is made with Unity 2018.2.9f1.
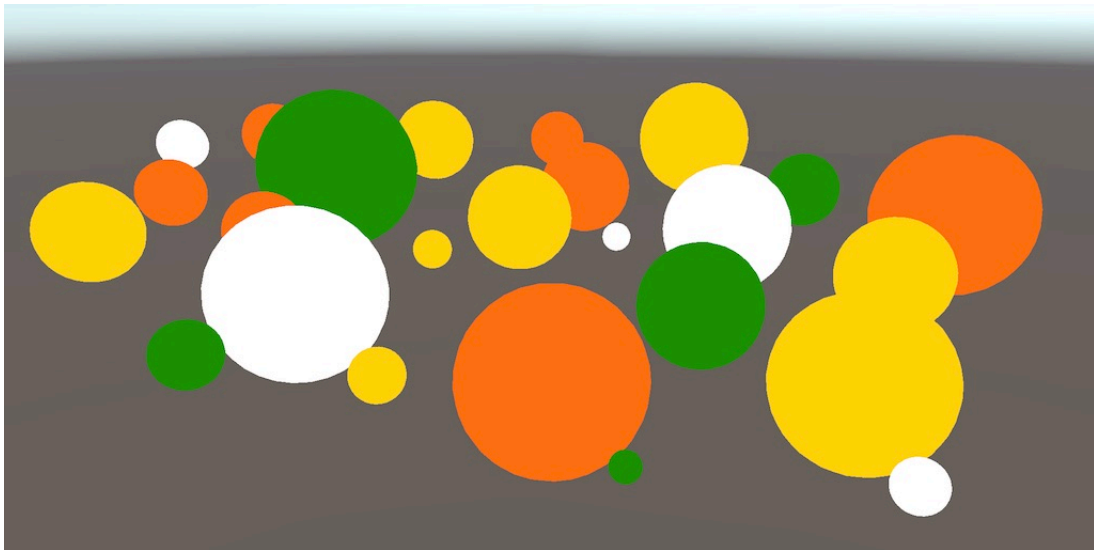
The birth of a new render pipeline.

1 Creating a Pipeline

To render anything, Unity has to determine what shapes have to be drawn, where, when, and with what settings. This can get very complex, depending on how many effects are involved. Lights, shadows, transparency, image effects, volumetric effects, and so on all have to be dealt with in the correct order to arrive at the final image. This process is known as a render pipeline.

Unity 2017 supports two predefined render pipelines, one for forward rendering and one for deferred rendering. It also still supports an older deferred rendering approach introduced in Unity 5. These pipelines are fixed. You are able to enable, disable, or override certain parts of the pipelines, but it's not possible to drastically deviate from their design.

Unity 2018 added support for scriptable render pipelines, making it possible to design pipelines from scratch, though you still have to rely on Unity for many individual steps, like culling. Unity 2018 introduced two new pipelines made with this new approach, the lightweight pipeline and the high-definition pipeline. Both pipelines are still in the preview stage and the scriptable render pipeline API is still marked as experimental technology. But at this point it is stable enough for us to go ahead and create our own pipeline.

In this tutorial we will setup a minimal render pipeline that draws unlit shapes. Once that's working, we can extend our pipeline in later tutorials, adding lighting, shadows, and more advanced features.

1.1 Project Setup

Open Unity 2018 and create a new project. I'm using Unity 2018.2.9f1, but any 2018.2 version or higher should also work. Create a standard 3D project, with analytics disabled. We'll create our own pipeline, so don't select one of the pipeline options.

Once the project is open, go the package manager via *Window / Package Manager* and remove all the packages that were included by default, as we won't need them. Only keep the *Package Manager UI*, which cannot be removed.
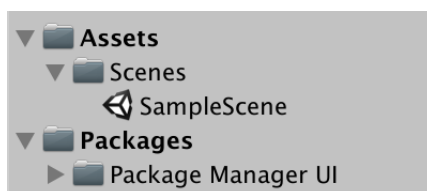

Starting project.

We're going to work in linear color space, but Unity 2018 still uses gamma space as the default. So go to the player settings via *Edit / Project Settings / Player* and switch *Color Space* in the *Other Settings* section to *Linear*.


Linear color space.

We'll need a few simple materials to test our pipeline. I've created four materials. First, a default standard opaque material with a red albedo. Second, the same material but with *Rendering Mode* set to *Transparent* and a blue albedo with decreased alpha. Third, a material using the *Unlit/Color* shader with its color set to yellow. And finally a material using the *Unlit/Transparent* shader without any changes, so it appears solid white.
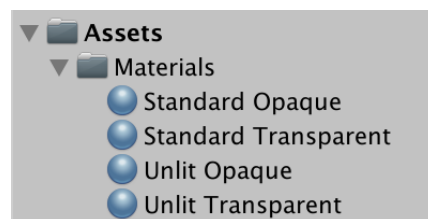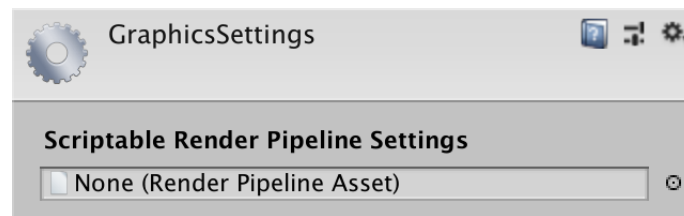

Test materials.

Fill the scene with a few objects, making use of all four materials.
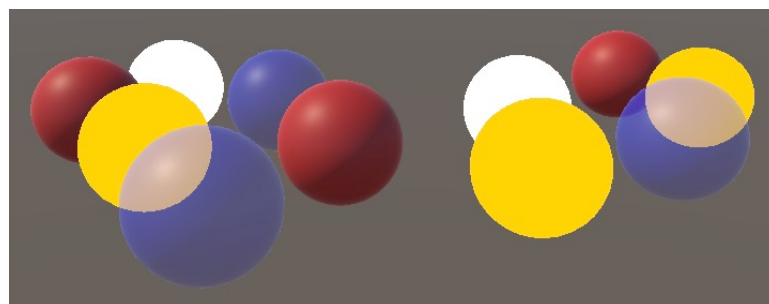
Scene displaying four materials.

1.2 Pipeline Asset

Currently, Unity uses the default forward rendering pipeline. To use a custom pipeline, we have to select one in the graphics settings, which can be found via *Edit / Project Settings / Graphics*.

Using the default pipeline.

To setup our own pipeline, we have to assign a pipeline asset to the *Scriptable Render Pipeline Settings* field. Such assets have to extend `RenderPipelineAsset`, which is a `ScriptableObject` type.

Create a new script for our custom pipeline asset. We'll simply name our pipeline *My Pipeline*. Its asset type will thus be `MyPipelineAsset` and it has to extend `RenderPipelineAsset`, which is defined in the `UnityEngine.Experimental.Rendering` namespace.

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

public class MyPipelineAsset : RenderPipelineAsset {}
```

> **Will it always be in the experimental namespace?**
>
> It will move out of the experimental namespace at some point, either to `UnityEngine.Rendering` or to another namespace. When that happens, it's just a matter of updating the `using` statement, unless the API also gets changed.

The main purpose of the pipeline asset is to give Unity a way to get a hold of a pipeline object instance that is responsible for rendering. The asset itself is just a handle and a place to store pipeline settings. We don't have any settings yet, so all we have to do is give Unity a way to get our pipeline object instance. This is done by overriding the `InternalCreatePipeline` method. But we haven't defined our pipeline object type yet, so at this point we'll just return `null`.

The return type of `InternalCreatePipeline` is `IRenderPipeline`. The *I* prefix of the type name indicates that it is an interface type.

```
public class MyPipelineAsset : RenderPipelineAsset {

    protected override IRenderPipeline InternalCreatePipeline () {
        return null;
    }
}
```

Now we need to add an asset of this type to our project. To make that possible, add a `CreateAssetMenu` attribute to `MyPipelineAsset`.

```
[CreateAssetMenu]
public class MyPipelineAsset : RenderPipelineAsset {}
```

That puts an entry in the *Asset / Create* menu. Let's be tidy and put it in a *Rendering* submenu. We do this by setting the `nemuName` property of the attribute to *Rendering/My Pipeline*. The property can be set directly after the attribute type, within round brackets.

```
[CreateAssetMenu(menuName = "Rendering/My Pipeline")]
public class MyPipelineAsset : RenderPipelineAsset {}
```

Use the new menu item to add the asset to the project, naming it *My Pipeline*.
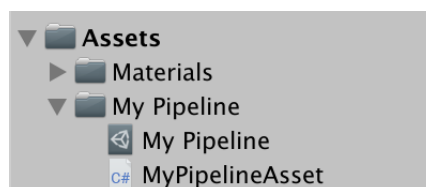
Pipeline asset and its script.

Then assign it to *Scriptable Render Pipeline Settings*.

Asset in use.

We have now replaced the default pipeline, which changes a few things. First, a lot of options have disappeared from the graphics settings, which Unity also mentions in an info panel. Second, as we've bypassed the default pipeline without providing a valid replacement, nothing gets rendered an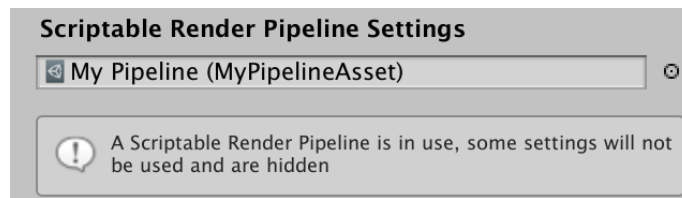ymore. The game window, scene window, and material previews are no longer functional, although the scene window still shows the skybox. If you open the frame debugger—via *Window / Analysis / Frame Debugger*—and enable it, you will see that indeed nothing gets drawn in the game window.

1.3 Pipeline Instance

To create a valid pipeline, we have to provide an object instance that implements `IRenderPipeline` and is responsible for the rendering process. So create a class for that, naming it `MyPipeline`.

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

public class MyPipeline : IRenderPipeline {}
```

Although we can implement `IRenderPipeline` ourselves, it is more convenient to extend the abstract `RenderPipeline` class instead. That type already provides a basic implementation of `IRenderPipeline` that we can build on.

```
public class MyPipeline : RenderPipeline {}
```

Now we can return a new instance of `MyPipeline` in `InternalCreatePipeline`. This means that we technically have a valid pipeline, although it still doesn't render anything.

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

2 Rendering

The pipeline object takes care of rendering each frame. All Unity does is invoke the pipeline's `Render` method with a context and the cameras that are active. This is done for the game window, but also for the scene window and material previews in the editor. It is up to us to configure things appropriately, figure out what needs to be rendered, and do everything in the correct order.

2.1 Context

`RenderPipeline` contains an implementation of the `Render` method defined in the `IRenderPipeline` interface. Its first argument is the render context, which is a `ScriptableRenderContext` struct, acting as a facade for native code. Its second argument is an array containing all cameras that need to be rendered.

`RenderPipeline`.`Render` doesn't draw anything, but checks whether the pipeline object is valid to use for rendering. If not, it will raise an exception. We will override this method and invoke the base implementation, to keep this check.

```
public class MyPipeline : RenderPipeline {

    public override void Render (
        ScriptableRenderContext renderContext, Camera[] cameras
    ) {
        base.Render(renderContext, cameras);
    }
}
```

It is through the render context that we issue commands to the Unity engine to render things and control render state. One of the simplest examples is drawing the skybox, which can be done by invoking the `DrawSkyBox` method.

```
        base.Render(renderContext, cameras);

        renderContext.DrawSkybox();
```
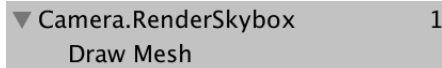
`DrawSkybox` requires a camera as a argument. We'll simply use the first element of `cameras`.

```
        renderContext.DrawSkybox(cameras[0]);
```

We still don't see the skybox appear in the game window. That's because the commands that we issue to the context are buffered. The actual works happens after we submit it for execution, via the `Submit` method.

```
        renderContext.DrawSkybox(cameras[0]);

        renderContext.Submit();
```

The skybox finally appears in the game window, and you can also see it appear in the frame debugger.

Frame debugger showing skybox gets drawn.

2.2 Cameras

We are supplied with an array of cameras, because there can exist multiple in the scene that all have to be rendered. Example uses for multiple-camera setups are split-screen multiplayer, mini maps, and rear-view mirrors. Each camera needs to be handled separately.

We won't worry about multi-camera support for our pipeline at this point. We'll simply create an alternative `Render` method that acts on a single camera. Have it draw the skybox and then submit. So we submit per camera.

```
    void Render (ScriptableRenderContext context, Camera camera) {
        context.DrawSkybox(camera);

        context.Submit();
    }
```

Invoke the new method for each element of the `cameras` array. I use a `foreach` loop in this case, as Unity's pipelines also use this approach to loop through the cameras.

```
    public override void Render (
        ScriptableRenderContext renderContext, Camera[] cameras
    ) {
        base.Render(renderContext, cameras);

        //renderContext.DrawSkybox(cameras[0]);

        //renderContext.Submit();

        foreach (var camera in cameras) {
            Render(renderContext, camera);
        }
    }
```

Note that the orientation of the camera currently doesn't affect how the skybox gets rendered. We pass the camera to `DrawSkybox`, but that's only used to determine whether the skybox should be drawn at all, which is controlled via the camera clear flags.

To correctly render the skybox—and the entire scene—we have to setup the view-projection matrix. This transformation matrix combines the camera's position and orientation—the view matrix— with the camera's perspective or orthographic projection—the projection matrix. You can see this matrix in the frame debugger. It is *unity_MatrixVP*, one of the shader properties used when something is drawn.

At the moment, the *unity_MatrixVP* matrix is always the same. We have to apply the camera's properties to the context, via the `SetupCameraProperties` method. That sets up the matrix as well as some other properties.

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

        context.DrawSkybox(camera);

        context.Submit();
    }
```

Now the skybox gets rendered correctly, taking the camera properties into account, both in the game window and in the scene window.

2.3 Command Buffers

The context delays the actual rendering until we submit it. Before that, we configure it and add commands to it for later execution. Some tasks—like drawing the skybox —can be issued via a dedicated method, but other commands have to be issued indirectly, via a separate command buffer.

A command buffer can be created by instantiating a new `CommandBuffer` object, which is defined in the `UnityEngine.Rendering` namespace. Command buffers already existed before the scriptable rendering pipeline was added, so they aren't experimental. Create such a buffer before we draw the skybox.

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

public class MyPipeline : RenderPipeline {

    …

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

        var buffer = new CommandBuffer();

        context.DrawSkybox(camera);

        context.Submit();
    }
}
```

We can instruct the context to execute the buffer via its `ExecuteCommandBuffer` method. Once again, this doesn't immediately execute the commands, but copies them to the internal buffer of the context.

```
        var buffer = new CommandBuffer();
        context.ExecuteCommandBuffer(buffer);
```

Command buffers claim resources to store their commands at the native level of the Unity engine. If we no longer need these resources, it is best to release them immediately. This can be done by invoking the buffer's `Release` method, directly after invoking `ExecuteCommandBuffer`.

```
        var buffer = new CommandBuffer();
        context.ExecuteCommandBuffer(buffer);
        buffer.Release();
```

Executing an empty command buffer does nothing. We added it so that we can clear the render target, to make sure that rendering isn't influenced by what was drawn earlier. This is possible via a command buffer, but not directly via the context.

A clear command can be added to the buffer by invoking `ClearRenderTarget`. It requires three arguments: two booleans and a color. The first argument controls whether the depth information is cleared, the second whether the color is cleared, and the third is the clear color, if used. For example, let's clear the depth data, ignore color data, and use `Color.clear` as the clear color.

```
var buffer = new CommandBuffer();
buffer.ClearRenderTarget(true, false, Color.clear);
context.ExecuteCommandBuffer(buffer);
buffer.Release();
```

The frame debugger will now show us that a command buffer get executed, which clears the render target. In this case, it indicated that Z and stencil get cleared. Z refers to the depth buffer, and the stencil buffer always gets cleared.

```
▼ Unnamed command buffer          1
      Clear (Z+stencil)
  ▶ Camera.RenderSkybox           1
```

Clearing the depth and stencil buffers.

What gets cleared is configured per camera, via its clear flags and background color. We can use those instead of hard-coding how we clear the render target.

```
CameraClearFlags clearFlags = camera.clearFlags;
buffer.ClearRenderTarget(
    (clearFlags & CameraClearFlags.Depth) != 0,
    (clearFlags & CameraClearFlags.Color) != 0,
    camera.backgroundColor
);
```

How do the clear flags work?

`CameraClearFlags` is an enumeration that can be used as a set of bit flags. Each bit of the value is used to indicate whether a certain feature is enabled or not.

To extract a bit flag from the entire value, combine the value with the desired flag using the bitwise AND operator `&`. If the result is not zero, then the flag is set.

Because we haven't given the command buffer a name, the debugger displays the default name, which is *Unnamed command buffer*. Let's use the camera's name instead, by assigning it to the buffer's `name` property. We'll use object initializer syntax to do this.

```
var buffer = new CommandBuffer {
    name = camera.name
};
```

```
▼ Main Camera                       1
      Clear (Z+stencil)
  ▶ Camera.RenderSkybox             1
```

Using camera name for the command buffer.

How does object initializer syntax work?

We could've also written `buffer.name = camera.name;` as a separate statement after invoking the constructor. But when creating a new object, you can append a code block to the constructor's invocation. Then you can set the object's fields and properties in the block, without having to reference the object instance explicitly. Also, it makes explicit that the instances should only be used after those fields and properties have been set. Besides that, it makes initialization possible where only a single statement is allowed, without requiring constructors with many parameter variants.

Note that we omitted the empty parameter list of the constructor invocation, which is allowed when object initializer syntax is used.

2.4 Culling

We're able to render the skybox, but not yet any of the objects that we put in the scene. Rather than rendering every object, we're only going to render those that the camera can see. We do that by starting with all renderers in the scene and then culling those that fall outside of the view frustum of the camera.

What are renderers?

It are components attached to game objects that turn them into something that can be rendered. Typically, a `MeshRenderer` component.

Figuring out what can be culled requires us to keep track of multiple camera settings and matrices, for which we can use the `ScriptableCullingParameters` struct. Instead of filling it ourselves, we can delegate that work to the static `CullResults`.GetCullingParameters method. It takes a camera as input and produces the culling parameters as output. However, it doesn't return the parameters struct. Instead, we have to supply it as a second output parameter, writing `out` in front of it.

```
void Render (ScriptableRenderContext context, Camera camera) {
    ScriptableCullingParameters cullingParameters;
    CullResults.GetCullingParameters(camera, out cullingParameters);

    …
}
```

Why do we have to write `out`?

Structs are value types, so they're treated like simple values. They aren't objects with an identity, with variables and fields only holding references to their location in memory. So passing the struct as an argument provides a method with a copy of that value. The method can change the copy, but that has no effect on the value that was copied.

When a struct parameter is defined as an output parameter, it acts like an object reference, but pointing to the place on the memory stack where the argument resides. When the method changes that argument, it affects that value, not a copy.

The `out` keyword tells us that the method is responsible for correctly setting the parameter, replacing the previous value.

Besides the output parameter, GetCullingParameters also returns whether it was able to create valid parameters. Not all camera settings are valid, resulting in degenerate results that cannot be used for culling. So if it fails, we have nothing to render and can exit from Render.

```
    if (!CullResults.GetCullingParameters(camera, out cullingParameters)) {
        return;
    }
```

Once we have the culling parameters, we can use them to cull. This is done by invoking the static `CullResults`.Cull method with both the culling parameters and the context as arguments. The result is a `CullResults` struct, which contains information about what is visible.

In this case, we have to supply the culling parameters as a reference parameter, by writing **ref** in front of it.

```
        if (!CullResults.GetCullingParameters(camera, out cullingParameters)) {
            return;
        }

        CullResults cull = CullResults.Cull(ref cullingParameters, context);
```

Why do we have to write ref?

It works just like **out**, except in this case the method is not required to assign something to the value. And whoever invokes the method is responsible for properly initializing the value first. So it can be used for input, and optionally for output.

Why is ScriptableCullingParameters a struct?

It's probably an optimization attempt, the idea being that you can create multiple parameter structs without having to worry about memory allocations. However, **ScriptableCullingParameters** is very large for a struct, which is why a reference parameter is used here, again for performance reasons. Maybe it started small but grew into a huge struct over time. Reusable object instances might be a better approach now, but we have to work with whatever Unity Technologies decides to use.

2.5 Drawing

Once we know what is visible, we can move on to rendering those shapes. This is done by invoking `DrawRenderers` on the context, with `cull.visibleRenderers` as an argument, telling it which renderers to use. Besides that, we have to supply draw settings and filter settings. Both are structs—**DrawRendererSettings** and **FilterRenderersSettings**—for which we'll initially use their default values. The draw settings have to be passed as a reference.

```
        buffer.Release();

        var drawSettings = new DrawRendererSettings();

        var filterSettings = new FilterRenderersSettings();

        context.DrawRenderers(
            cull.visibleRenderers, ref drawSettings, filterSettings
        );

        context.DrawSkybox(camera);
```

We're not seeing any objects yet, because the default filter settings include nothing. We can instead include everything by providing `true` as an argument for the `FilterRenderersSettings` constructor. This tells it to initialize itself so it includes everything.

```
var filterSettings = new FilterRenderersSettings(true);
```

Also, we have to configure the draw settings by providing its constructor with the camera and a shader pass as arguments. The camera is used to setup sorting and culling layers, while the pass controls which shader pass is used for rendering.

The shader pass is identified via a string, which has to be wrapped in a `ShaderPassName` struct. As we're only supporting unlit materials in our pipeline, we'll use Unity's default unlit pass, identified with *SRPDefaultUnlit*.

```
var drawSettings = new DrawRendererSettings(
    camera, new ShaderPassName("SRPDefaultUnlit")
);
```


Opaque spheres are visible.

We see the opaque unlit shapes appear, but not the transparent ones. However, the frame debugger indicates that the unlit shapes get drawn too.

All unlit renderers are drawn.

They do get drawn, but because the transparent shader pass does not write to the depth buffer they end up getting drawn over by the skybox. The solution is to delay drawing transparent renderers until after the skybox.

First, limit the draw before the skybox to only the opaque renderers. This is done by setting the `renderQueueRange` of the filter settings to `RenderQueueRange.opaque`, which covers the render queues from 0 up to and including 2500.

```
var filterSettings = new FilterRenderersSettings(true) {
    renderQueueRange = RenderQueueRange.opaque
};
```

```
▶ Main Camera                              1
▼ RenderLoop.Draw                          3
    Draw Mesh Sphere Unlit Opaque (1)
    Draw Mesh Sphere Unlit Opaque
    Draw Mesh Sphere Unlit Opaque (2)
▶ Camera.RenderSkybox                      1
```

Only opaque renderers are drawn.

Next, change the queue range to `RenderQueueRange.transparent`—from 2501 up to and including 5000—after rendering the skybox, and render again.

```
var filterSettings = new FilterRenderersSettings(true) {
    renderQueueRange = RenderQueueRange.opaque
};

context.DrawRenderers(
    cull.visibleRenderers, ref drawSettings, filterSettings
);

context.DrawSkybox(camera);

filterSettings.renderQueueRange = RenderQueueRange.transparent;
context.DrawRenderers(
    cull.visibleRenderers, ref drawSettings, filterSettings
);
```
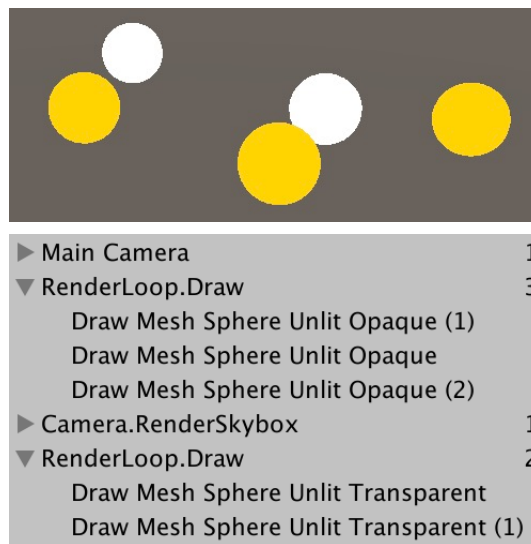
Opaque, skybox, then transparent.

We draw the opaque renderers before the skybox to prevent overdraw. As the shapes will always be in front of the skybox, we avoid work by drawing them first. That's because the opaque shader pass writes to the depth buffer, which is used to skip anything that's drawn later that ends up further away.

Besides covering parts of the sky, opaque renderers can also end up obscuring each other. Ideally, only the one closest to the camera is drawn for each fragment of the frame buffer. So to reduce overdraw as much as possible, we should draw the nearest shapes first. That can be done by sorting the renderers before drawing, which is controlled via sorting flags.

The draw settings contain a `sorting` struct of type **DrawRendererSortSettings**, which contains the sort flags. Set it to **SortFlags**.CommonOpaque before drawing the opaque shapes. This instructs Unity to sort the renderers by distance, from front to back, plus a few other criteria.

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

However, transparent rendering works differently. It combines the color of what's being drawn with what has been drawn before, so the result appears transparent. That requires the reverse draw order, from back to front. We can use **SortFlags**.CommonTransparent for that.

```
        context.DrawSkybox(camera);

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

Our pipeline is now able to render both opaque and transparent unlit objects correctly.

3 Polishing

Being able to render correctly is only part of having a functional pipeline. There are other things to consider, such as whether it is fast enough, doesn't allocate unneeded temporary objects, and integrates well with the Unity editor.

3.1 Memory Allocations

Let's check whether our pipeline behaves well in terms of memory management, or if it allocates memory every frame, which will trigger frequent memory garbage collection runs. This is done by opening the profiler via *Window / Analysis / Profiler* and inspecting the *CPU Usage* data in *Hierarchy* mode. While you can do this in play mode in the editor, it is also a good idea to profile a build, by making sure you create a development build and having it attach to the profiler automatically, although deep profiling isn't possible in that case.

Sort by *GC Alloc* and you'll see that memory indeed gets allocated each frame. Some of it is out of our control, but quite a few bytes are allocated inside our pipeline's `Render` method.

It turns out that culling allocates the most memory. The reason for this is that although `CullResults` is a struct, it contains three lists, which are objects. Each time we ask for a new cull result, we end up allocating memory for new lists. So there isn't much benefit to `CullResults` being a struct.

Fortunately, `CullResults` has an alternative `Cull` method that accepts a struct as a reference parameter, instead of returning a new one. That makes it possible to reuse the lists. All we have to do is turn `cull` into a field and provide it as an additional argument to `CullResults.Cull`, instead of assigning to it.

```
    CullResults cull;

    …

    void Render (ScriptableRenderContext context, Camera camera) {
        …

        //CullResults cull = CullResults.Cull(ref cullingParameters, context);
        CullResults.Cull(ref cullingParameters, context, ref cull);

        …
    }
```

Another source of continuous memory allocation is our use of the camera's name property. Each time we get its value, it fetches the name data from native code, which necessitates the creation of a new string, which is an object. So let's always name our command buffer *Render Camera* instead.

```
var buffer = new CommandBuffer() {
    name = "Render Camera"
};
```


Using a constant buffer name.

Finally, the command buffer itself is an object too. Fortunately, we can create a command buffer once and reuse it. Replace the local variable with a `cameraBuffer` field. Thanks to the object initialization syntax, we can create a named command buffer as its default value. The only other change is that we have to clear the command buffer instead of releasing it, for which we can use its `Clear` method.

```
CommandBuffer cameraBuffer = new CommandBuffer {
    name = "Render Camera"
};

…

void Render (ScriptableRenderContext context, Camera camera) {
    …

    //var buffer = new CommandBuffer() {
    //   name = "Render Camera"
    //};
    cameraBuffer.ClearRenderTarget(true, false, Color.clear);
    context.ExecuteCommandBuffer(cameraBuffer);
    //buffer.Release();
    cameraBuffer.Clear();

    …
}
```

After these changes our pipeline no longer creates temporary objects every frame.

3.2 Frame Debugger Sampling

Another thing that we can do is improve the data shown by the frame debugger. Unity's pipelines display a nested hierarchy of events, but ours are all at the root level. We can build a hierarchy by using our command buffer to begin and end profiler samples.

Let's start by invoking `BeginSample` right before `ClearRenderTarget`, immediately followed by an invocation of `EndSample`. Each sample must have both a begin and an end, and both must be provided with the exact same name. Besides that, I have found that it is best to use the same name as the command buffer that defines the sampling. The command buffer's name often gets used anyway.

```
cameraBuffer.BeginSample("Render Camera");
cameraBuffer.ClearRenderTarget(true, false, Color.clear);
cameraBuffer.EndSample("Render Camera");
context.ExecuteCommandBuffer(cameraBuffer);
cameraBuffer.Clear();
```

```
▼ Render Camera                          1
    ▼ Render Camera                      1
         Clear (Z+stencil)
  ▶ RenderLoop.Draw                      3
  ▶ Camera.RenderSkybox                  1
  ▶ RenderLoop.Draw                      2
```

Sampling creates a hierarchy.

We now see a *Render Camera* level nested inside the original *Render Camera* of the command buffer, which in turn contains the clear operation. But it is possible to go a step further, nesting all other actions related to the camera inside it as well. This requires us to delay the ending of the sample until right before we submit the context. So we have to insert an additional `ExecuteCommandBuffer` at that point, only containing the instruction to end the sample. Use the same command buffer for this, again clearing it after we are done.

```
cameraBuffer.BeginSample("Render Camera");
cameraBuffer.ClearRenderTarget(true, false, Color.clear);
//cameraBuffer.EndSample("Render Camera");
context.ExecuteCommandBuffer(cameraBuffer);
cameraBuffer.Clear();

…

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

context.Submit();
```

Nested sampling.

This looks good, except that the clear action gets nested inside a redundant *Render Camera* level, while all other actions are directly below the root level. I'm not sure why this happens, but it can be avoided by beginning the sample after clearing.

```
//cameraBuffer.BeginSample("Render Camera");
cameraBuffer.ClearRenderTarget(true, false, Color.clear);
cameraBuffer.BeginSample("Render Camera");
context.ExecuteCommandBuffer(cameraBuffer);
cameraBuffer.Clear();
```

No redundant nesting.

3.3 Rendering the Default Pipeline

Because our pipeline only supports unlit shaders, objects that use different shaders are not rendered, making them invisible. While this is correct, it hides the fact that some objects use the wrong shader. It would be nice if we visualized those objects with Unity's error shader, so they show up as obviously incorrect magenta shapes. Let's add a dedicated `DrawDefaultPipeline` method for this, with a context and a camera parameter. We'll invoke it at the end, after drawing the transparent shapes.

```
        void Render (ScriptableRenderContext context, Camera camera) {
            …

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

            DrawDefaultPipeline(context, camera);

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

            context.Submit();
        }

        void DrawDefaultPipeline(ScriptableRenderContext context, Camera camera) {}
```

Unity's default surface shader has a *ForwardBase* pass that is used as the first forward rendering pass. We can use this to identify objects that have a material that works with the default pipeline. Select that pass via new draw settings and use that for rendering, together with new default filter settings. We don't care about sorting or separating opaque and transparent renderers, because they're invalid anyway.

```
        void DrawDefaultPipeline(ScriptableRenderContext context, Camera camera) {
            var drawSettings = new DrawRendererSettings(
                camera, new ShaderPassName("ForwardBase")
            );

            var filterSettings = new FilterRenderersSettings(true);

            context.DrawRenderers(
                cull.visibleRenderers, ref drawSettings, filterSettings
            );
        }
```


Rendering forward base pass.

The objects that use the default shader now show up. They're also visible in the frame debugger.

```
▼ Render Camera                                    13
    Clear (Z+stencil)
  ▶ RenderLoop.Draw                                 3
  ▶ Camera.RenderSkybox                             1
  ▼ RenderLoop.Draw                                 8
      Draw Mesh Sphere Unlit Transparent (1)
      Draw Mesh Sphere Unlit Transparent
      Draw Mesh Sphere Standard Transparent
      Draw Mesh Sphere Standard Opaque (2)
      Draw Mesh Sphere Standard Opaque (1)
      Draw Mesh Sphere Standard Transparent (1)
      Draw Mesh Sphere Standard Opaque
      Draw Mesh Sphere Standard Transparent (2)
```

Everything gets drawn.

Because our pipeline does not support a forward base pass they do not get rendered correctly. The necessary data isn't setup, so everything that depends on lighting ends up as black. Instead, we should render them with an error shader. To do this, we need an error material. Add a field for that. Then, at the start of `DrawDefaultPipeline`, create the error material if it doesn't already exist. This is done by retrieving the *Hidden/InternalErrorShader* via `Shader`.Find, then creating a new material with that shader. Also, set the material's hide flags to `HideFlags`.HideAndDontSave so it doesn't show up in the project window and doesn't get saved along with all other assets.

```csharp
	Material errorMaterial;

	…

	void DrawDefaultPipeline(ScriptableRenderContext context, Camera camera) {
		if (errorMaterial == null) {
			Shader errorShader = Shader.Find("Hidden/InternalErrorShader");
			errorMaterial = new Material(errorShader) {
				hideFlags = HideFlags.HideAndDontSave
			};
		}

		…
	}
```

One option of the draw settings is to override the material used when rendering, by invoking `SetOverrideMaterial`. Its first parameter is the material to use. Its second parameter is the index of the pass of the material's shader to be used for rendering. As the error shader only has a single pass, use zero.

```csharp
		var drawSettings = new DrawRendererSettings(
			camera, new ShaderPassName("ForwardBase")
		);
		drawSettings.SetOverrideMaterial(errorMaterial, 0);
```
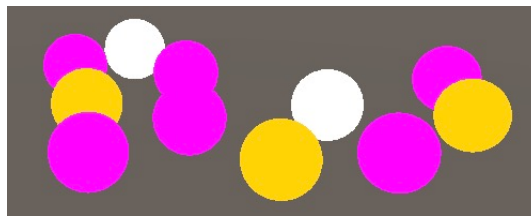
Using the error shader.

Objects that use an unsupported material now clearly show up as incorrect. But this is only true for materials of Unity's default pipeline, whose shaders have a `ForwardBase` pass. There are other built-in shaders that we can identify with different passes, specifically *PrepassBase*, *Always*, *Vertex*, *VertexLMRGBM*, and *VertexLM*.

Fortunately, it is possible to add multiple passes to the draw settings, by invoking `SetShaderPassName`. The name is the second parameter of this method. Its first parameter is an index that controls the order in which the passes are drawn. We don't care about that, so any order is fine. The pass provided via the constructor always has index zero, just increment the index for additional passes.

```
var drawSettings = new DrawRendererSettings(
    camera, new ShaderPassName("ForwardBase")
);
drawSettings.SetShaderPassName(1, new ShaderPassName("PrepassBase"));
drawSettings.SetShaderPassName(2, new ShaderPassName("Always"));
drawSettings.SetShaderPassName(3, new ShaderPassName("Vertex"));
drawSettings.SetShaderPassName(4, new ShaderPassName("VertexLMRGBM"));
drawSettings.SetShaderPassName(5, new ShaderPassName("VertexLM"));
drawSettings.SetOverrideMaterial(errorMaterial, 0);
```

That covers all shaders provided by Unity up to this point, which should be sufficient to help point out the use of incorrect materials when creating a scene. But we only need to bother doing this during development, not in a build. So let's only invoke `DrawDefaultPipeline` in the editor. One way to do this is by adding a `Conditional` attribute to the method.

3.4 Conditional Code Execution

The `Conditional` attribute is defined in the `System.Diagnostics` namespace. We can use that namespace, but it unfortunately also contains a `Debug` type, which conflicts with `UnityEngine.Debug`. As we only need the attribute, we can avoid the conflict by using an alias. Instead of using the entire namespace, we use a specific type and assign it to a valid type name. In this case, we'll define `Conditional` as an alias for `System.Diagnostics.ConditionalAttribute`.

```
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Experimental.Rendering;
using Conditional = System.Diagnostics.ConditionalAttribute;
```

Add the attribute to our method. It requires a string argument that specifies a symbol. If the symbol is defined during compilation, then the method invocation gets included as normal. But if the symbol is not defined, then the invocation of this method—including all its arguments—are omitted. It would be as if the `DrawDefaultPipeline(context, camera);` code didn't exist during compilation.

To only include the invocation when compiling for the Unity editor, we have to rely on the *UNITY_EDITOR* symbol.

```
	[Conditional("UNITY_EDITOR")]
	void DrawDefaultPipeline(ScriptableRenderContext context, Camera camera) {
		…
	}
```

We can go a step further and also include the invocation in development builds, only excluding it from release builds. To do so, add an additional conditional with the *DEVELOPMENT_BUILD* symbol.

```
	[Conditional("DEVELOPMENT_BUILD"), Conditional("UNITY_EDITOR")]
	void DrawDefaultPipeline(ScriptableRenderContext context, Camera camera) {
		…
	}
```

3.5 UI in Scene Window

One thing that we haven't considered up to this point is Unity's in-game UI. To test it, add a UI element to the scene, for example a single button, via *GameObject / UI / Button*. That creates a canvas with a button it it, plus an event system.

It turns out that the UI gets rendered in the game window, without us having to do anything. Unity takes care of it for us. The frame debugger shows that the UI gets rendered separately, as an overlay.

UI gets drawn in screen space.

At least, that's the case when the canvas is set to render in screen space. When set to render in world space, the UI gets rendered along with the other transparent objects.

UI in world space.

Although the UI works in the game window, it doesn't show up the scene window. The UI always exists in world space in the scene window, but we have to manually inject it into the scene. Adding the UI is done by invoking the static `ScriptableRenderContext`.`EmitWorldGeometryForSceneView` method, with the current camera as an argument. This must be done before culling.

```
        if (!CullResults.GetCullingParameters(camera, out cullingParameters)) {
            return;
        }

        ScriptableRenderContext.EmitWorldGeometryForSceneView(camera);

        CullResults.Cull(ref cullingParameters, context, ref cull);
```

But this also adds the UI a second time in the game window. To prevent that, we must only emit the UI geometry when rendering the scene window. This is the case when the `cameraType` of the camera is equal to **CameraType**.SceneView.

```
        if (camera.cameraType == CameraType.SceneView) {
            ScriptableRenderContext.EmitWorldGeometryForSceneView(camera);
        }
```

This works, but only in the editor. Conditional compilation makes sure that the `EmitWorldGeometryForSceneView` doesn't exist when compiling for a build, which means that we now get a compiler error when trying to build. To make it work again, we have to make the code that invokes `EmitWorldGeometryForSceneView` conditional as well. That is done by putting the code in between `#if` and `#endif` statements. The `#if` statement requires a symbol, just like the `Conditional` attribute. By using *UNITY_EDITOR*, the code to only get included when compiling for the editor.

```
    void Render (ScriptableRenderContext context, Camera camera) {
        ScriptableCullingParameters cullingParameters;
        if (!CullResults.GetCullingParameters(camera, out cullingParameters)) {
            return;
        }

#if UNITY_EDITOR
        if (camera.cameraType == CameraType.SceneView) {
            ScriptableRenderContext.EmitWorldGeometryForSceneView(camera);
        }
#endif

        CullResults.Cull(ref cullingParameters, context, ref cull);

        …
    }
```

Our pipeline is now properly configured and ready to become more complex. Want to know when the next tutorial is released? Keep tabs on my Patreon page!

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick