Scriptable Render Pipeline

SRP的核心是一堆API集合，使得整个渲染过程及相关配置暴露给用户，使得用户可以精确地控制项目的渲染流程。
SRP API为原有的Unity构件提供了一些新的接口(interface):

- Lights
- Materials
- Cameras
- Command Buffers

但是和Unity交互的方式变了。由于性能原因，当自定义SRP时，通常是在使用一组renderers(这里的render应该是指MeshRender之类的)，而不是一个。

What is a Render Pipeline

什么是渲染管线？“Render Pipeline”，渲染管线，是许多技术的总称，用于把物体(Objects，比如三维物体)显示到屏幕上。概括性讲，渲染管线包括：

- Culling
- Rendering Objects
- Post processing

除了上面这些高级概念，渲染管线的每一个环节或者子任务，还可以进一步分解，你可以选择如何去完成它。例如：可以使用一下方式渲染物体:

- Multi-pass rendering - one pass per object per light 即一个物体被"渲染多次(多个pass)"，每一个影响到该物体的light，都在一个单独的pass中渲染
- Single-pass - one pass per object 即每个物体中一个pass中渲染，也即一个物体只“渲染”一次
- Deferred - Render surface properties to a g-buffer, perform screen space lighting 即先在g-buffer pass中把法线等几何属性“渲染”到g-buffer纹理中，然后在Lighting pass中进行屏幕空间的光照计算

当自定义SRP时，上面这些就是需要你做出决策的地方。每种技术都有许多权衡(trade offs)需要考虑，没有一种技术是完美的。

What is a Scriptable Render Pipeline

如果**Render Pipeline**是为了完成渲染而需要执行的许多步骤，则**Scriptable Render Pipeline**是一个可以让用户使用Unity的C#脚本控制的pipeline，用于以用户定义的方式完成渲染。

The Problem (Built-in pipelines存在的问题)

之前Unity已经提供了一些built-in pipelines，一些适合用于手游或者VR游戏(如forward renderer)，另一些用于高级终端游戏上，如主机游戏 (如 deferred renderer)。这些Unity内置的拿来即用的渲染方案具有通用性，“黑盒性”，但也有一些缺陷：

- 只能用于做它们设计好的任务
- 它们被设计的非常具有通用性，功能过于宽泛，意味着它们可以完成任何需求，但没有任何完美精通或者适用的地方。(But they need to do everything, they are masters at nothing)
- 它们不是那可配置的。(black box with injection points)
- 扩展和修改容易产生错误(小的内部改变可能产生大的外部影响)
- 有许多bugs无法修复(因为擅自“修复”可能会破坏项目)。

The Solution (用SRP来解决)

SRP API的出现就是为了解决上面的问题。它把渲染部分从Unity内置的“黑盒”概念 转变成了用户可控的每个项目可脚本控制的概念。也就是说，使用SRP，每一个项目都可以有一个符合自己的特色的渲染管线。使用SRP api可以对how rendering进行精细粒度的自定义，从低层到高层(form the low level to the high level)。

SRP 概述

从高层次用户角度来看，SRP可以分为两部分：SRP asset和SRP instance。在制作custom rendering pipeline时，两者都很重要。

SRP Asset

SRP Asset是project asset，表示渲染管线的一个具体配置，例如：

- 是否支持阴影？
- 使用什么级别的shader quality?
- shadow distance值是多少？
- 默认材质配置

用户想要控制的东西可以保存为配置的一部分，基本上是任何需要序列化的东西。SRP Asset表示SRP的类型(type)和其中的设置(settings)。

SRP Instance

SRP Instance是实际完成渲染的类(class)。当Unity发现SRP被启用时，Unity会查看当前选择的SRP Asset并且要求它提供一个渲染实例("rendering instance")，即SRP asset要返回一个实例包含一个'Render'函数。通常这个实例会缓存许多对应SRP asset中的配置信息。

SRP instance表示一个已知的渲染管线配置(pipeline configuration)。从渲染器角度看，调用动作可能如下：

- Clearing the framebuffer
- Performing scene culling
- Rendering sets of objects

- Doing blits from one frame buffer to another
- Rendering shadows
- Applying post process effects

SRP instance表示一个实际将要被实施的渲染(*actual* rendering)。

SRP Asset

SRP Asset包含一个接口，用户用它配置渲染管线。当Unity第一次开始渲染时，Unity会调用 SRP Asset上的`InternalCreatePipeline`函数，然后，SRP Asset会返回一个可用的SRP instance。

SRP Asset是一个ScriptableObject，这意味着它可以保存为一个project asset(见下面的BasicAssetPipe)。

为了使项目启用一个SRP asset，需要在**Edit>Project Settings>Graphics**的Scriptable Render Pipeline Settings栏中选择你生成的SRP Asset。这样设置好后，Unity就会使用SRP Asset中的配置进行rendering了，而不再使用standard Unity rendering。

An Simple Asset Example

SRP Asset的职责是包含配置信息和返回一个渲染实例。若SRP Asset上的某个设置发生了变化，则所有由这个Asset创建的实例会被销毁，并且用新的设置创建新的实例，来进行下一帧的渲染。

下面的代码是一个简单的pipeline asset类。它包含一个color，由它返回的实例用来清除屏幕。`CreateBasicAssetPipeline`是在Editor下使用的一个工具， 在菜单栏上点**SRP-Demo**，再点**01 - Create Basic Asset Pipeline**即可创建一个BasicAssetPipe对应的SRP Asset，它在项目中的路径为`Assets/BasicAssetPipe.asset`。

```
1  [ExecuteInEditMode]
2  public class BasicAssetPipe : RenderPipelineAsset
3  {
4      public Color clearColor = Color.green;
5
6  #if UNITY_EDITOR
7      // Call to create a simple pipeline
8      [UnityEditor.MenuItem("SRP-Demo/01 - Create Basic
   Asset Pipeline")]
9      static void CreateBasicAssetPipeline()
10     {
11         var instance =
   ScriptableObject.CreateInstance<BasicAssetPipe>();
12
    UnityEditor.AssetDatabase.CreateAsset(instance,
   "Assets/BasicAssetPipe.asset");
13     }
14 #endif
15
16     // Function to return an instance of this pipeline
```

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

A complete asset example

除了返回实例和保存配置，SRP asset也用于做许多辅助功能，如：

- Default material to use when creating 3d objects
- Default material to use when creating 2d objects
- Default material to use when creating particle systems
- Default material to use when creating terrain systems

The Rendering entry point

SRP asset控制渲染配置，但是最终渲染是由SRP Render Pipeline instance完成的。SRP instance对应的类(class)是渲染逻辑(rendering logic)实际存在的地方。

SRP Instance的最简单的形式仅仅包含一个`Render`函数。`Render`函数有两个参数:

- 一个`ScriptableRenderContext`类型的参数，相当于一个队列，其中的元素是command buffer，可以把将要完成的渲染操作入队(enqueue)。
- 一个相机数组`Camera[]`，表示已经启用的，用于渲染的相机列表。

A basic pipeline

下面这个`BasicPipeInstance`类就是上面的`BasicAssetPipe`的`IRenderPipeline`函数返回的SRP Instance。

```
public class BasicPipeInstance : RenderPipeline
{
    private Color m_ClearColor = Color.black;

    public BasicPipeInstance(Color clearColor)
    {
        m_ClearColor = clearColor;
    }

    public override void Render(ScriptableRenderContext context, Camera[] cameras)
    {
        // does not so much yet :()
        base.Render(context, cameras);

        // clear buffers to the configured color
```

```
        var cmd = new CommandBuffer();
        cmd.ClearRenderTarget(true, true,
m_ClearColor);
        context.ExecuteCommandBuffer(cmd);
        cmd.Release();
        context.Submit();
    }
}
```

上面代码所表示的pipeline仅仅完成了简单地用给定的颜色(由SRP Asset指定)清除屏幕的操作。注意:

- Unity现有的`CommandBuffers`被用来完成许多操作(此处用于完成`ClearRenderTarget`)。
- `CommandBuffers`是根据于对应的context进行调试的(scheduled)。
- “渲染”的最后一步是调用`Submit`，它将(引起)执行所有入队的command。

`RenderPipeline`的`Render`函数就是你输入渲染代码，来自定义renderer的地方。Culling, Filtering, Changing render targets和Drawing操作都是在这里完成的。这就是你构造渲染器的地方!

SRP使用延迟执行的方式进行渲染。作为用户，你的任务就是用`ScriptableRenderContext`构建一个命令队列，然后去执行它们。

Culling

Culling(剔出)是指明将要在屏幕上渲染什么的过程。

Unity的Culling过程包含两类：

- Frustum culling: 视锥体剔出，计算在视锥体内的物体(这些物体会被渲染)。
- Occlusion culling: 遮挡剔出，计算被别的物体遮挡的物体，这些物体将不会被渲染。

渲染开始时，首先要计算需要渲染哪些物体。这涉及到获取相机，然后完成剔出(cull)操作(从给定相机的角度)。Culling操作返回一个对于给定相机有效的物体(objects和lights)列表，这些物体用于该相机进行后面的渲染步骤。

Culling in SRP

在SRP，用户通常站在相机的视角来渲染对象。这和Unity built-in rendering一样。SRP提供了一些有用的Culling相关的API。通常流程如下：

```
// Create an structure to hold the culling paramaters
ScriptableCullingParameters cullingParams;

//Populate the culling paramaters from the camera
if (!CullResults.GetCullingParameters(camera,
stereoEnabled, out cullingParams))
```

```
    continue;

// if you like you can modify the culling paramaters here
cullingParams.isOrthographic = true;

// Create a structure to hold the cull results
CullResults cullResults = new CullResults();

// Perform the culling operation
CullResults.Cull(ref cullingParams, context, ref cullResults);
```

`cullResults`被用来完成接下来的rendering。

SRP Drawing

经过上面的步骤，该剔出的物体已经被剔出了，现在可把cull results渲染到屏幕上了。

需要提前做一些决策(考虑到以下因素):

- 完成渲染的硬件
- 想要实现的具体样子(specific look)和风格/感觉(feel)，即要实现的效果
- 项目的类型

如，一个2D手游和一个PC上的第一人称游戏所使用的渲染管线肯定会差异特别大。可能要做下面这些抉择：

- HDR vs LDR
- Linear vs Gamma
- MSAA vs Post Process AA
- PBR Materials vs Simple Materials
- Lighting vs No Lighting
- Lighting Technique
- Shadowing Technique

目前，我们将要展示一个简单的没有光照、可以渲染一些不透明物体的渲染器。

Filtering: Render Buckets and Layers

通常，渲染对象(rendering object)有一个具体的分类：透明的、不透明的、sub surface，或者别的类型。Unity使用队列表示什么时候一个对象将会被渲染，即相同分类的对象被放在同一个队列中，这些队列也被称为“桶”(bucket)。在SRP中，用户指定使用哪些桶进行渲染。

除了“桶”的概念，标准的Unity layers也可以被使用。

这提供了额外的过滤能力。

```
// Get the opaque rendering filter settings
var opaqueRange = new FilterRenderersSettings();

//Set the range to be the opaque queues
opaqueRange.renderQueueRange = new RenderQueueRange()
{
    min = 0,
    max =
(int)UnityEngine.Rendering.RenderQueue.GeometryLast,
};

//Include all layers
opaqueRange.layerMask = ~0;
```

Draw Settings: How things should be drawn

上面的filtering和culling决定了将要渲染哪些对象。接下来，我们需要决定怎样渲染它们。SRP提供了许多可配置选项。用于配置的数据结构是`DrawRenderSettings`。它允许配置以下选项：

- Sorting - 物体渲染的顺序，如：从前到后(front to back)或者从后到前(back to front)。
- Per Renderer flags - Unity应该向shader传入什么'built in'设置，包括：per object light probes, per object light maps等。
- Rendering flags - 使用哪种batching算法，instancing vs non-instancing。
- Shader Pass - 当前draw call应该使用哪个shader pass

```
// Create the draw render settings
// note that it takes a shader pass name
var drs = new DrawRendererSettings(myCamera, new
ShaderPassName("Opaque"));

// enable instancing for the draw call
drs.flags = DrawRendererFlags.EnableInstancing;

// pass light probe and lightmap data to each renderer
drs.rendererConfiguration =
RendererConfiguration.PerObjectLightProbe |
RendererConfiguration.PerObjectLightmaps;

// sort the objects like normal opaque objects
drs.sorting.flags = SortFlags.CommonOpaque;
```

Drawing

现在我们已经有了发起一次draw call所需要的三样东西：

- Culling results (剔出结果)
- Filtering rules (过滤规则)
- Drawing rules (绘制规则)

下面的代码发起了一次draw call。在SRP中，你一般不渲染单独的一个或几个网格(individual meshes)，而是发起一次draw call，一次渲染一大批网格。这能减少执行开销。

发起一次draw call，需要结合上面我们已经构建的参数。

```
// draw all of the renderers
context.DrawRenderers(cullResults.visibleRenderers, ref drs, opaqueRange);

// submit the context, this will execute all of the queued up commands.
context.Submit();
```

这段代码会把对象渲染到当前绑定的渲染目标上(render target)。也可以通过command buffer来切换不同的渲染目标。

参考:

1. Scriptable Render Pipeline Wiki
2. Unity SRP Overview: Scriptable Render Pipeline