

Scriptable Render Pipeline 简介

Lei Xinyue

Table of Contents

1 Introduction

- Render Pipeline
- SRP Overview

2 Custom SRP

- SRP Asset
- SRP Instance
- Custom SRP Demos

3 LWRP vs HDRP

- LWRP
- HDRP

What is a Render Pipeline

什么是渲染 (Rendering)?

What is a Render Pipeline

Rendering(computer graphics) is the automatic process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of computer programs.(from Wikipedia)

What is a Render Pipeline

渲染就是使用计算机程序 (根据 2D/3D 模型/场景) 自动生成真实感或非真实感图像的过程。

What is a Render Pipeline

什么是渲染管线 (Render Pipeline)?

What is a Render Pipeline

渲染管线 (Render Pipeline), 是许多技术的总称.
概括性地讲, 渲染管线包括:

- ① Culling(剔出)
- ② Rendering Objects(渲染对象/物体)
- ③ Post processing(后处理)

Culling

Culling 是指明将要在屏幕上渲染什么的过程。

Culling

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，计算出完全被别的物体遮挡的物体，这些物体将不会被渲染。

▶ return

Culling

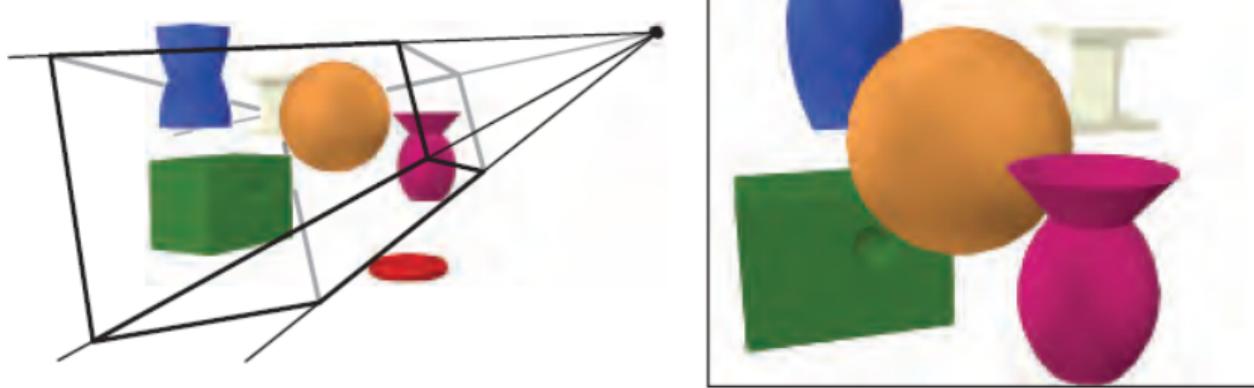


图 1: 左图中右上方的黑点代表 Virtual Camera, 截头锥体表示 frustum。右图是 Camera 看到的结果, 不在 frustum 内的物体不会被渲染

Rendering Objects

上面说的是概括性的概念，渲染管线的每一个环节或者子任务，还可以进一步分解

Rendering Objects

你可以选择如何去完成它。例如：可以使用以下方式渲染物体：

Rendering Objects

- 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

Rendering Objects

- 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
即每个物体在一个 pass 中渲染，也即一个物体只 “渲染” 一次

Rendering Objects

- 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
即每个物体在一个 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

Rendering Objects

当自定义 Scriptable Render Pipeline(SRP) 时，上面这些就是需要你做出决策的地方。

Rendering Objects

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

什么是可脚本配置的渲染管线？

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。

SRP

The **Scriptable Render Pipeline (SRP)** is an alternative to the Unity built-in render pipeline. With the SRP, you can control and tailor rendering via C# scripts.

The Problem

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）。

The Problem

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：

The Problem

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

The Problem

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)

The Problem

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)
- 可配置性不强。 (black box with injection points)

The Problem

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)
- 可配置性不强。 (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)

The Problem

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)
- 可配置性不强。 (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

SRP 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精细控制的、每个项目可脚本控制的概念。

The Solution

也就是说，使用 SRP，每一个项目都可以有一个甚至多个符合自己的特色的渲染管线。（见后面的 Demos）

SRP Overview

从高层次角度来看，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 Overview

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

SRP Overview

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

SRP Asset 表示 SRP 的类型 (type) 和其中的设置 (settings)。

SRP Overview

SRP Instance

SRP Overview

SRP Instance 是实际完成渲染的类 (class)。

SRP Overview

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")。通常这个实例会缓存许多对应 SRP asset 中的配置信息。
(Unity 会根据这个 rendering instance 来渲染场景)

SRP Overview

SRP instance 表示一个已知的渲染管线配置 (pipeline configuration)。从渲染器角度看，SRP instance 中可能发生的动作如下：

- Clearing the framebuffer
- Performing scene culling
- Rendering sets of objects
- Doing blits from one frame buffer to another
- Rendering shadows
- Applying post process effects

Table of Contents

1 Introduction

- Render Pipeline
- SRP Overview

2 Custom SRP

- SRP Asset
- SRP Instance
- Custom SRP Demos

3 LWRP vs HDRP

- LWRP
- HDRP

How to build a custom SRP?

怎样徒手创建一个自定义 SRP?

How to build a custom SRP?

- ① 创建一个 SRP Asset 资源
- ② 创建一个 SRP Instance 类
- ③ 启用 SRP，在 Unity 配置 Graphics Settings，使用第一步的 SRP Asset。

▶ 具体见下文

SRP Asset

创建一个 SRP Asset:

- ① 新建一个 C# 类，继承 UnityEngine.Experimental.Rendering.RenderPipelineAsset
- ② 重写 (override)InternalCreatePipeline 方法

SRP Asset

创建一个 SRP Asset:

- ① 新建一个 C# 类, 继承 UnityEngine.Experimental.Rendering.RenderPipelineAsset
- ② 重写 (override)InternalCreatePipeline 方法

RenderPipelineAsset 继承自 UnityEngine.ScriptableObject, 同时实现了
UnityEngine.Experimental.Rendering.IRenderPipelineAsset.

SRP Asset

创建一个 SRP Asset:

- ① 新建一个 C# 类, 继承 `UnityEngine.Experimental.Rendering.RenderPipelineAsset`
- ② 重写 (`override`)`InternalCreatePipeline` 方法

故 SRP Asset 是一个 `ScriptableObject`, 则它可以保存为一个 project asset。

SRP Asset

创建一个 SRP Asset:

- ① 新建一个 C# 类, 继承 UnityEngine.Experimental.Rendering.RenderPipelineAsset
- ② 重写 (override)InternalCreatePipeline 方法

protected override IRenderPipeline InternalCreatePipeline() 返回一个 RenderPipeline 实例。

SRP Asset

```
[ExecuteEditMode]
public class BasicAssetPipe : RenderPipelineAsset
{
    public Color clearColor = Color.green;

#if UNITY_EDITOR
    // Call to create a simple pipeline
    [UnityEditor.MenuItem("SRP-Demo/01 - Create Basic Asset Pipeline")]
    static void CreateBasicAssetPipeline()
    {
        var instance = ScriptableObject.CreateInstance<BasicAssetPipe>();
        UnityEditor.AssetDatabase.CreateAsset(instance, "Assets/BasicAssetPipe.asset");
    }
#endif

    // Function to return an instance of this pipeline
    protected override IRenderPipeline InternalCreatePipeline()
    {
        return new BasicPipeInstance(clearColor);
    }
}
```

图 2: 一个简单的 SRP Asset, 参见 SRP Instance 的代码

SRP Instance

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

SRP instance 对应的类 (class) 是渲染逻辑 (rendering logic) 实际存在的地方。

SRP Instance

SRP Instance 的最简单的形式仅仅包含一个 Render 函数。

SRP Instance

```
public override void Render(ScriptableRenderContext context, Camera[] cameras);
```

- 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。

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 do 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();
    }
}
```

图 3: 一个简单的 SRP instance, ▶ 参见 SRP Asset 的代码

SRP Instance

SRP 采用了延迟执行的模式 (使用 CommandBuffer 和 ScriptableRenderContext)。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。

SRP Instance

Render 函数很重要。Culling, Filtering, Changing render targets 和 Drawing 操作都是在这里完成的 (详见后文及 Demo)。 Render 就是你构造渲染器的地方!

SRP Instance

一切准备好后，记得启用创建的
SRP Asset:

SRP Instance

一切准备好后，记得启用创建的 SRP Asset:

在 **Edit>Project Settings>Graphics** 的 Scriptable Render Pipeline Settings 栏中选择刚才创建的 SRP Asset(默认是 None)。

▶ How to build custom srp

这样 Unity 就会使用 SRP Asset 中的配置进行 rendering 了，而不再使用 standard Unity rendering。

What? How? Where?

在深入定制 (实现) 渲染逻辑时，必须要考虑的三个问题：

- ① What: 要渲染什么? ▶ Culling Intro.
- ② How: 怎样去渲染?
- ③ Where: 渲染到哪里?(默认是屏幕,backbuffer)

Culling in SRP

什么是 Draw Call?

在 Unity 中，每次引擎准备数据并通知 GPU 的过程称为一次Draw Call。

Culling in SRP

什么是 Draw Call?

在 Unity 中，每次引擎准备数据并通知 GPU 的过程称为一次 **Draw Call**。
这一过程是逐个物体进行的，对于每个物体，不止 GPU 的渲染，引擎重新设置材质/Shader 也是一项非常耗时的操作。
(场景内可能有大量物体)

Culling in SRP

SRP 中，通常是站在相机的视角来渲染对象。这和 Unity built-in rendering 一样。

Culling in SRP

因为 CPU 准备顶点数据发起 DrawCall 的代价比较大。

Culling in SRP

因为 CPU 准备顶点数据发起 DrawCall 的代价比较大。
所以在渲染开始前，把所有完全处在相机视野之外的物体全部从渲染列表中剔除。

Culling in SRP

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 parameters
ScriptableCullingParameters cullingParams;

//Populate the culling parameters from the camera
if (!CullResults.GetCullingParameters(camera, stereoEnabled, out cullingParams))
    continue;

// if you like you can modify the culling parameters 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);
```

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**、
subsurface，或者别的类型。

Filtering: Render Buckets and Layers

Unity 使用队列表示一个对象什么时候将会被渲染，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)。

Filtering: Render Buckets and Layers

在 SRP 中，用户指定使用哪些桶进行渲染。

Filtering: Render Buckets and Layers

除了“桶”的概念，标准的 Unity layers 也可以被使用。

Filtering: Render Buckets and Layers

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。

Draw Settings: How things should be drawn

接下来，我们需要决定怎样渲染它们。

Draw Settings: How things should be drawn

SRP 提供了许多可配置选项。用于配置的数据结构是 `DrawRenderSettings`

Draw Settings: How things should be drawn

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

Draw Settings: How things should be drawn

```
// 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：

- ① Culling results (剔除结果)
- ② Filtering rules (过滤规则)
- ③ Drawing rules (绘制规则)

Drawing

下面的代码发起了一次 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();
```

Drawing

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。)

Custom SRP Demos

自定义渲染管线 Demos:

- ① Demo 1: ClearRenderTarget
- ② Demo 1: Transparency
- ③ Demo 3: custom SRP 给场景添加光照
- ④ Demo 4: 自定义 SRP, 降低 DrawCall 次数 (从 3700 降到 27)

Custom SRP Demos

Demo 4:

场景中，两种角色，每种角色（带有阴影）各创建 1500 个实例，开启 GPU Instancing。
分别用 Unity built-in renderpipeline 和 custom srp 进行渲染。

Custom SRP Demos

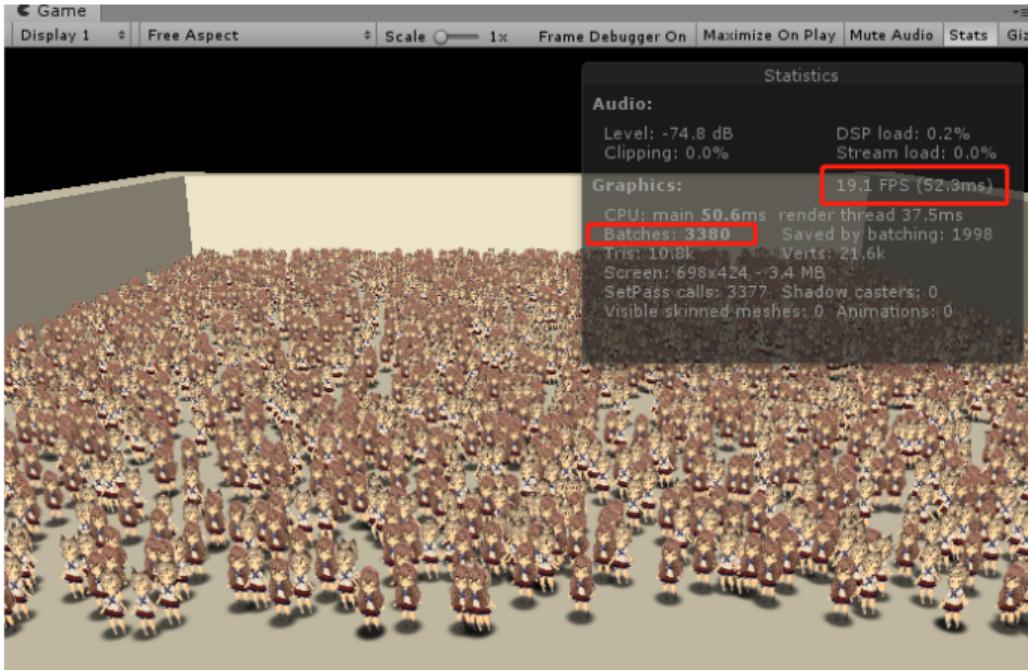


图 3: Unity built-in 渲染管线渲染结果

Custom SRP Demos

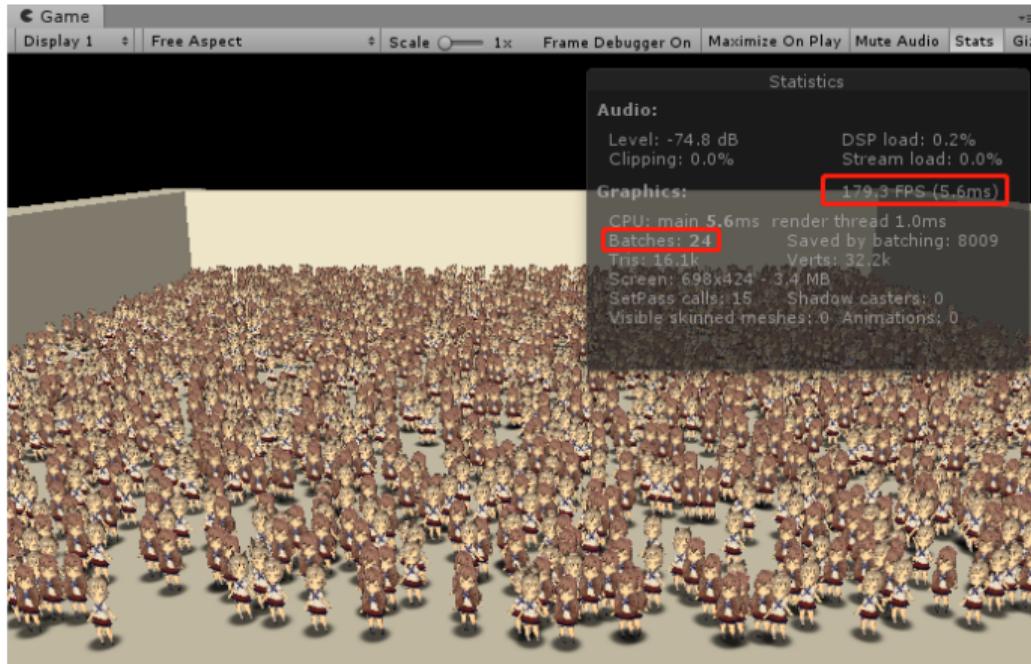


图 3: 作者 (Yusuke) 通过分析出现问题的原因，利用 SRP 自定义的渲染管线渲染结果

Custom SRP Demos

Built-in 渲染管线的问题：

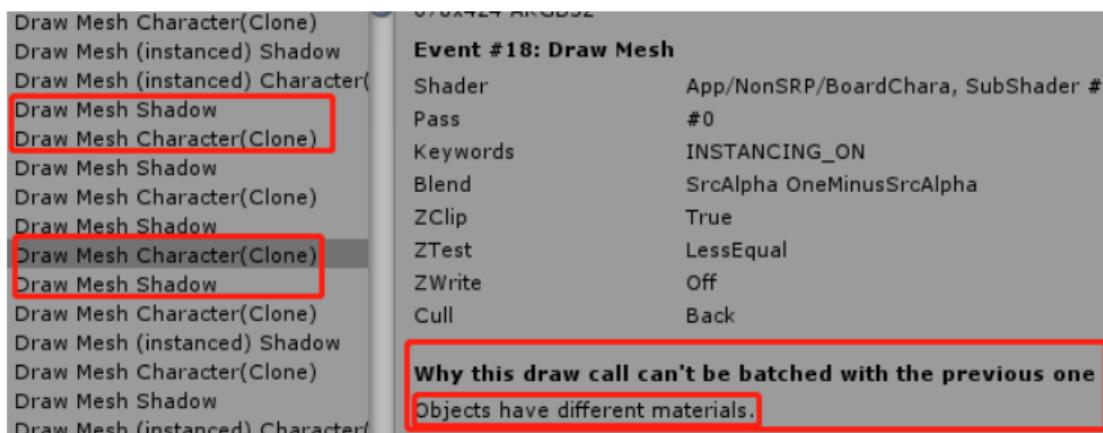


图 3：出现问题的原因：角色和阴影交替出现，导致 Batching 失败

Custom SRP Demos

Batching: Unity 一种提高 DrawCall 效率，减少 DrawCall 次数的技术。可以使一次 DrawCall 渲染多个物体，但要求这些物体具有相同的材质。

Custom SRP Demos

Batching: Unity 一种提高 DrawCall 效率，减少 DrawCall 次数的技术。可以使一次 DrawCall 渲染多个物体，但要求这些物体具有相同的材质。

GPU Instancing: 也是一种提高 DrawCall 效率，减少 DrawCall 次数的技术。可以一次渲染多个具有相同 Mesh 和 material 的物体。GPU Instancing 一次 DrawCall 只能绘制具有相同网格的物体，但每个实例可以有不同的参数，如:(color 和 Scale)

Custom SRP Demos

Characters 和 Shadows 的 Material 不一样，又交叉渲染，导致 Batching 失败。

Custom SRP Demos

解决方法：
分组渲染。

- ① 先渲染背景 (由四个 Cubes 组成的围墙和地面)
- ② 再渲染角色 (Characters)
- ③ 最后再渲染阴影 (Shadows)

Custom SRP Demos

针对这个场景的 Custom SRP 核心代码:

```
// 绘制背景
DrawBg(context, camera);
// 绘制角色, 使用basicPass
// SortFlags.OptimizeStateChanges: Sort objects to reduce draw state changes.
DrawCharacter(context, camera, basicPass, SortFlags.OptimizeStateChanges);
// 绘制透明的阴影
DrawShadow(context, camera);
```

图 3: 自定义渲染管线核心代码

Custom SRP Demos

```
var filterSettings = new FilterRenderersSettings(true)
{
    renderQueueRange = RenderQueueRange.opaque,
    layerMask = 1 << LayerDefine.BG
}:
context.DrawRenderers(cull.visibleRenderers, ref settings, filterSettings);
```

图 3: DrawBg 方法的核心代码

Custom SRP Demos

```
var filterSettings = new FilterRenderersSettings(true)
{
    renderQueueRange = RenderQueueRange.transparent,
    layerMask = 1 << LayerDefine.CHARA
};
context.DrawRenderers(cull.visibleRenderers, ref settings, filterSettings);
```

图 3: DrawCharacter 方法的核心代码

Custom SRP Demos

```
var filterSettings = new FilterRenderersSettings(true)
{
    renderQueueRange = RenderQueueRange.transparent,
    layerMask = 1 << LayerDefine.SHADOW
}:
context.DrawRenderers(cull.visibleRenderers, ref settings, filterSettings);
```

图 3: DrawShadow 方法的核心代码

Table of Contents

1 Introduction

- Render Pipeline
- SRP Overview

2 Custom SRP

- SRP Asset
- SRP Instance
- Custom SRP Demos

3 LWRP vs HDRP

- LWRP
- HDRP

Light Weight Render Pipeline

Lightweight Render Pipeline (LWRP), 轻量级渲染管线，是一个 Unity 预制的 Scriptable Render Pipeline (SRP)。

Light Weight Render Pipeline

LWRP 可以为移动平台提供图形渲染功能，但你也可以在高端主机和 PC 上使用 LWRP。

Light Weight Render Pipeline

LWRP 使用 single-pass 前向渲染 (forward rendering)。

Light Weight Render Pipeline

使用 LWRP，可以得到在几个平台上优化了的实时渲染性能。

Light Weight Render Pipeline

LWRP 支持以下平台:

- Windows and UWP
- Mac and iOS
- Android
- XBox One
- PlayStation4
- Nintendo Switch
- All current VR platforms

Light Weight Render Pipeline

使用 LWRP 的项目无法与 High Definition Render Pipelin (HDRP) 及 Unity built-in rendering pipeline 兼容。因此开发前，你要想清楚用哪一个渲染管线。

Light Weight Render Pipeline

LWRP 提供了各种 Shading Models 及对应的 Shaders，适用于不同的硬件平台。

① Physically Based Shading

- Lit
- Particles Lit

Light Weight Render Pipeline

LWRP 提供了各种 Shading Models 及对应的 Shaders，适用于不同的硬件平台。

① Physically Based Shading

- Lit
- Particles Lit

② Simple shading

- Simple Lit
- Particles Simple Lit

Light Weight Render Pipeline

LWRP 提供了各种 Shading Models 及对应的 Shaders，适用于不同的硬件平台。

① Physically Based Shading

- Lit
- Particles Lit

② Simple shading

- Simple Lit
- Particles Simple Lit

③ Baked Lit shading

- LWRP Baked Lit shader

Light Weight Render Pipeline

LWRP 提供了各种 Shading Models 及对应的 Shaders，适用于不同的硬件平台。

① Physically Based Shading

- Lit
- Particles Lit

② Simple shading

- Simple Lit
- Particles Simple Lit

③ Baked Lit shading

- LWRP Baked Lit shader

④ Shaders with no lighting

Light Weight Render Pipeline

LWRP 的详细使用见这篇文章:[Lightweight Render Pipeline](#)或者官网文档:[About the Lightweight Render Pipeline](#)

Heigh Definition Render Pipeline

High Definition Render Pipeline (HDRP), 高清晰度渲染管线，是一个 Unity 建立的高保真 (high-fidelity) Scriptable Render Pipeline，适用于比较新的 (即兼容 Compute Shader 的) 硬件平台。

Heigh Definition Render Pipeline

HDRP 仅支持以下平台 (且对应的设备要支持 Compute Shaders)：

- Windows and Windows Store, with DirectX 11 or DirectX 12 and Shader Model 5.0
- Modern consoles (Sony PS4 and Microsoft Xbox One)
- MacOS using Metal graphics
- Linux and Windows platforms with Vulkan
- Future: IOS with Metal, Android

Heigh Definition Render Pipeline

HDRP 仅支持以下平台 (且对应的设备要支持 Compute Shaders)：

- Windows and Windows Store, with DirectX 11 or DirectX 12 and Shader Model 5.0
- Modern consoles (Sony PS4 and Microsoft Xbox One)
- MacOS using Metal graphics
- Linux and Windows platforms with Vulkan
- Future: IOS with Metal, Android

HDRP 不支持 OpenGL 或 OpenGL ES 设备。

Heigh Definition Render Pipeline

HDRP 和 LWRP 是不兼容的。你必须决定好你的项目使用哪种渲染管线。

Heigh Definition Render Pipeline

HDRP 提供一个 Volume 组件，可以把场景分成几个区域，每个区域可独自进行光照处理。

Heigh Definition Render Pipeline

HDRP 提供一个 Volume 组件，可以把场景分成几个区域，每个区域可独自进行光照处理。每一个 volume 有一个 environment，因此你可以调整它的 sky 和 shadow settings。

Heigh Definition Render Pipeline

HDRP 提供了 Visual Environment 组件使你能够改变场景中 sky、fog 的类型。

Heigh Definition Render Pipeline

HDRP 提供了 Visual Environment 组件使你能够改变场景中 sky、fog 的类型。使用 volumetric fog 可以创建光线穿过雾的效果 (atmospheric light rays)，如下图：

Heigh Definition Render Pipeline



参考链接

- Scriptable Render Pipeline Manual
- Unity at GDC - Scriptable Render Pipeline Intro & Lightweight Rendering Pipeline
- Scriptable Render Pipeline: What You Need To Know
- ScriptableRenderPipeline Wiki
- HDRP
- LWRP
- Scriptable Render Pipeline
- 开发自定义 SRP，将 DrawCall 降低 180 倍
- CustomScriptRenderPipelineTest
- 一起来写 Unity 渲染管线吧
- 在 Unity 里写一个纯手动的渲染管线