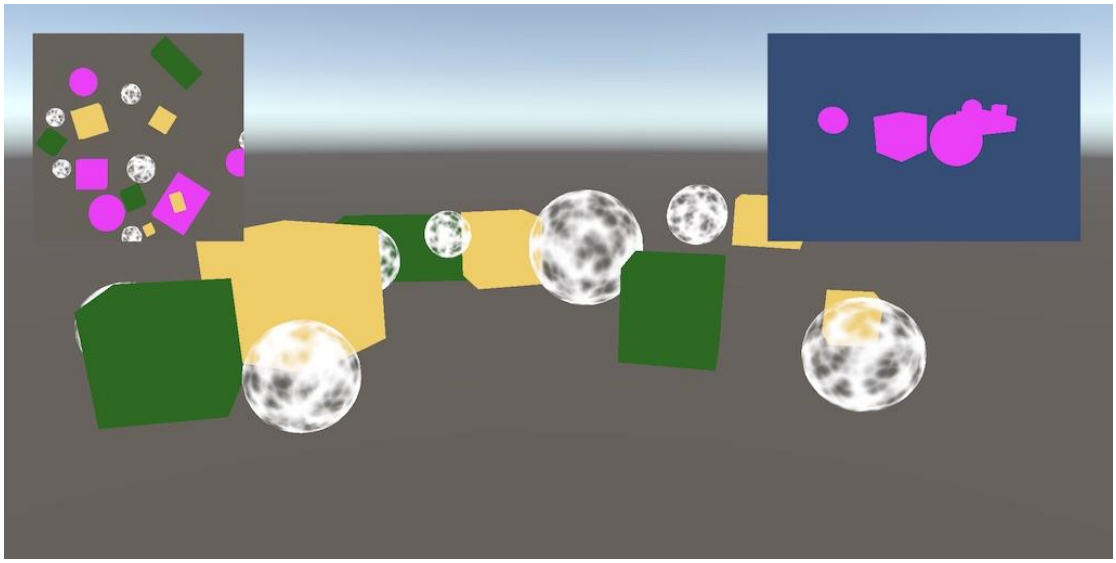
自定义渲染管线 *控制渲染流程*

创建一个渲染管线资产和实例
渲染一个摄像机视图
进行剔除、过滤和排序
拆分不透明、透明和不正确的绘制单元
使用多摄像机

这是创建自定义渲染管线系列教程的第一部分，内容包括一个可扩展的自定义渲染管线框架的初始化创建。

该系列假定你已经至少了解过物件管理器及程序网格系列教程。

该教程基于 Unity 2019.2.6f1。



使用自定义管线进行渲染

1. 新的渲染管线（A new Render Pipeline）

要渲染任何内容，Unity 都需要确定哪些内容需要渲染，包括渲染位置，渲染时间以及相关设置。这可能非常复杂，取决于涉及了多少效果。光线、阴影、透明度、图像效果、空间效果等等都需要按照正确的顺序处理并最后显示在图像中，这就是渲染管线的工作。

以前 Unity 只支持若干种内建流程用来渲染。Unity 2018 采用了可编程的渲染管线（简称 RP），使得我们可以在依赖剔除等 Unity 基础功能的前提下做任何事。Unity 2018 也基于这种方法加入了两种实验性的渲染管线：轻量级渲染管线（LWRP）和高精度渲染管线（HDRP）。其中 LWRP 在 Unity 2019 中不再是实验性功能，并且在 Unity 2019.3 中重新整合成通用渲染管线（Universal RP，简称 URP）。

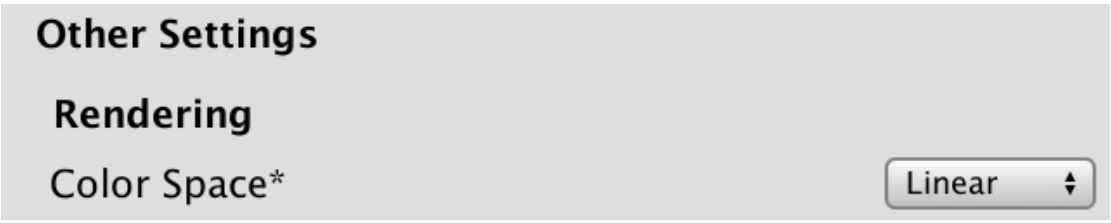
通用渲染管线注定会取代当前旧版本的渲染管线，原因在于它既具有普适性又便于自定义。该系列教程将从头开始创建一个完整的渲染管线，而不是对 URP 进行自定义调整。

本篇教程将建立一个使用前向渲染绘制无光照物体的最简渲染管线。完成之后，我们将在后面的教程中逐步扩展我们的管线，包括添加光照、阴影、不同的渲染方法以及更多高级功能。

1.1. 项目设置（Project Setup）

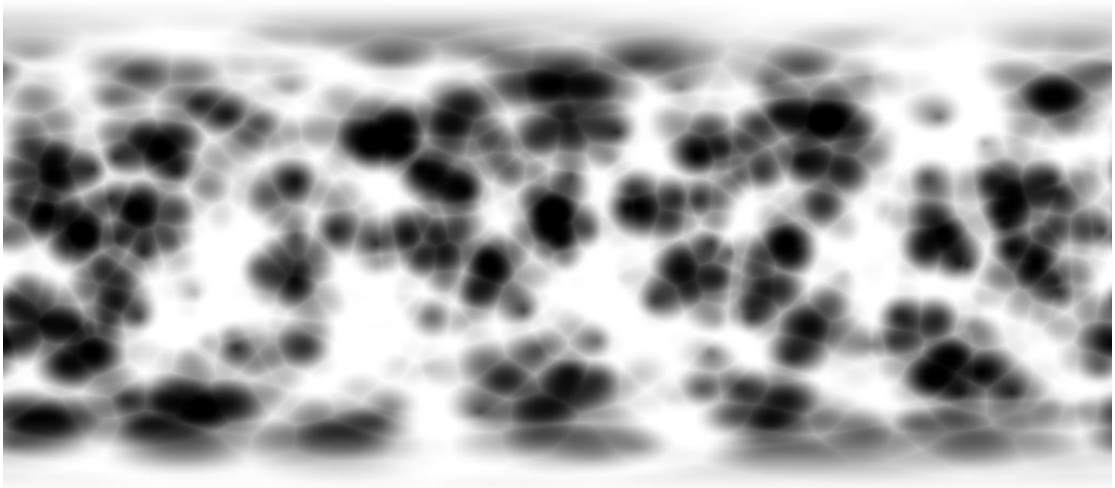
在 Unity 2019.2.6 或更新的版本中创建一个新 3D 项目。我们要创建自己的管线，所以不选择任何一种管线项目模板。项目打开后，你可以打开包管理器，然后移除所有不需要的包。本篇教程中我们只需要使用 Unity UI 包来测试绘制 UI，所以保留它。

我们只需要在线性色彩空间下工作，但 Unity 2019.2 仍然默认使用伽马空间。打开用户设置（*Edit->Project Settings->Player*），将 *Other Settings* 下的 *Color Space* 改成 *Linear*。



Color Space 设置成 Linear

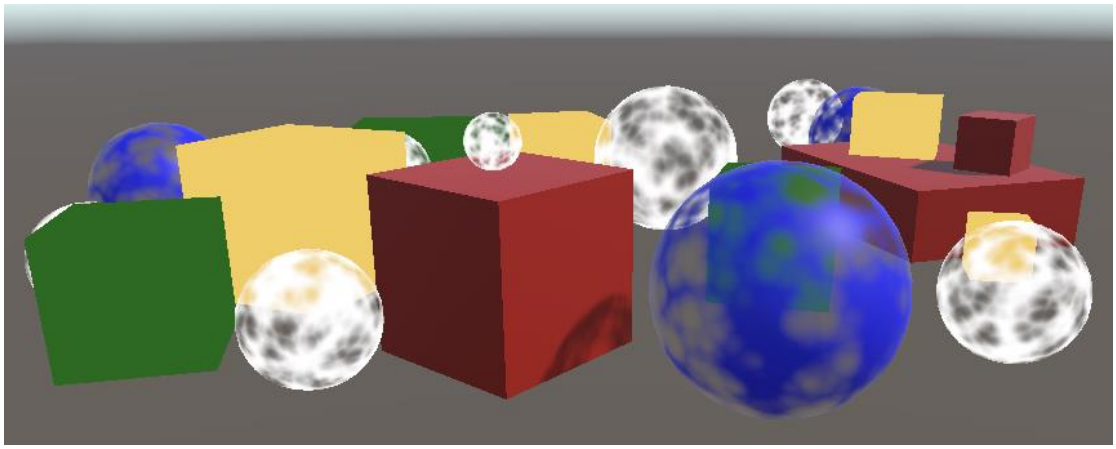
在默认场景中放一些物件，并使用标准材质，无光照不透明和透明等多种不同材质。其中 Unlit/Transparent 的 Shader 需要一张贴图，这里提供一张可用的球体 uv 贴图。



黑色背景下的球体 uv 贴图

我在测试场景中放了一些不透明的立方体。红色的材质球使用了 Standard shader，绿色和黄色的使用了 Unlit/Color shader。蓝色的球体使用了 Standard shader 并将 Render Mode 设

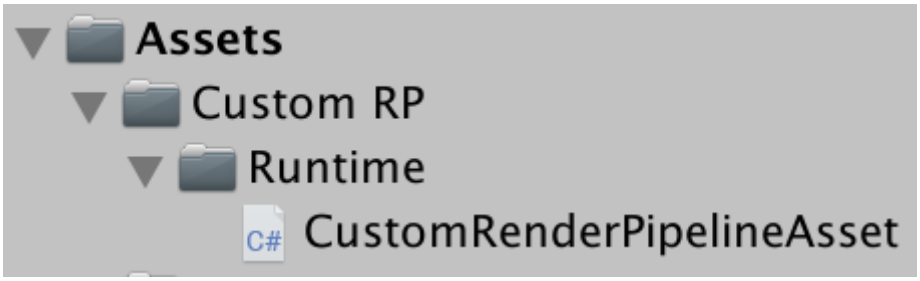
置为 Transparent，白色球体使用 Unlit/Transparent shader。



测试场景

1.2. 管线资产（Pipeline Asset）

目前，Unity 还在使用默认渲染管线。为了把它替换成自定义渲染管线，我们先创建为它创建一个资产类型。我们粗略地使用与 Unity 的 URP 相同的文件夹结构。创建一个名为 Custom RP 资产文件夹，并包含一个名为 Runtime 的子文件夹，在 Runtime 中新建一个 CustomRenderPipilineAsset 的 C#脚本。



文件夹结构

这个资产类型必须继承于 UnityEngine.Rendering 下的 RenderPipilineAsset 类。

```
1.  using UnityEngine;
2.  using UnityEngine.Rendering;
3.
4.  public class CustomRenderPipelineAsset : RenderPipelineAsset {}
```

这个渲染管线资产的主要目的是给 Unity 提供一个掌握渲染管线对象实例的方法。这个资产本身只是一个存储设置的句柄空间。我们暂时不需要任何设置，所以只需要让 Unity 有方法来获取我们的管线对象实例。这可以通过重写抽象方法 CreatePipiline 来完成，该方法会返回一个 RenderPipeline 实例。但我们尚未定义自定义渲染管线类型，所以先设定返回值为 null。

```
1.  protected override RenderPipeline CreatePipeline () {
```

```
2.      return null;
3.  }
```

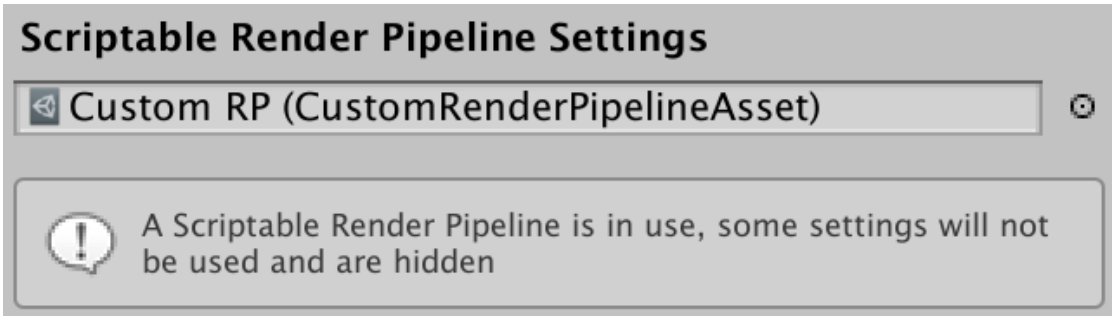
现在我们需要给项目添加一个该类型的资产。为了实现这个功能，添加一个CreateAssetMenu 属性给 CustomRenderPipelineAsset。

```
1.  [CreateAssetMenu]
2.  public class CustomRenderPipelineAsset : RenderPipelineAsset { … }
```

这洋就在 Asset/Create 菜单中添加了一个入口。为了整齐起见，把它移到名为 Rendering 的子菜单中。我们可以通过把 menuName 参数设置为 Rendering/Custom Render Pipeline 来实现。这个参数可以在属性后面的圆括号里直接设置设置。

```
1.  [CreateAssetMenu(menuName = "Rendering/Custom Render Pipeline")]
2.  public class CustomRenderPipelineAsset : RenderPipelineAsset { … }
```

使用创建的新菜单把该资产添加到项目，然后到 Edit->Project Settings->Graphics 中，在 Scriptable Render Pipeline Settings 下选择刚才新建的管线。



选择 Custom RP

替换默认渲染管线让一些内容发生了变化。首先，Graphics Settings 里的许多选项不见了，这在信息面板中有提示。其次，我们在没有提供有效替代的情况下禁用了默认渲染管线，所以现在没有渲染任何东西。如果你打开并启用帧调试器面板（Window->Analysis->Frame Debugger），你会发现游戏窗口中实际上没有东西被绘制。

1.3. 渲染管线实例（Render Pipeline Instance）

创建一个 CustomRenderPipeline 类，然后将脚本文件放到 CustomRenderPipelineAsset 同文件夹下。这是我们管线资产返回的渲染管线实例，因此必须继承 RenderPipeline。

```
1.  using UnityEngine;
2.  using UnityEngine.Rendering;
3.
4.  public class CustomRenderPipeline : RenderPipeline {}
```

RenderPipeline 定义了一个名为 Render 的 protected 级别的方法，我们要重写它以创建一个具体的管线。该方法有两个参数：一个 ScriptableRenderContext 和一个 Camera 数组。我们

先空着这个方法。

```
protected override void Render (
	ScriptableRenderContext context, Camera[] cameras
) {}
```

让 CustomRenderPipelineAsset.CreatePipeline 返回一个 CustomRenderPipeline 的新实例。这使得我们有了一个合法并且有功能的管线，尽管它现在还没有渲染任何内容。

```
protected override RenderPipeline CreatePipeline () {
	return new CustomRenderPipeline();
}
```

2. 渲染（Rendering）

Unity 每一帧都会调用管线实例中的 Render 方法。这个方法传递一个向原生引擎提供连接的上下文结构，我们可以用它来进行渲染。它也传递了一个摄像机数组，因为场景中可以存在多个激活的摄像机。按顺序渲染所有提供的摄像机是渲染管线的工作。

2.1. 摄像机渲染器（Camera Renderer）

每个摄像机都会独立渲染，所以我们将这个工作放到一个专用于渲染摄像机的新类里，而不是在 CustomRenderPipeline 里渲染所有的摄像机。将它命名为 CameraRenderer，并且添加一个 Render 的公共方法，包含一个 context 和 camera 参数。为了方便使用，我们把这些参数保存在字段中。

```
using UnityEngine;
using UnityEngine.Rendering;

public class CameraRenderer {

	ScriptableRenderContext context;

	Camera camera;

	public void Render (ScriptableRenderContext context, Camera camera) {
		this.context = context;
		this.camera = camera;
	}
}
```

在创建 CustomRenderPipeline 的时候创建一个渲染器实例，然后通过循环来渲染所有的摄像机。

```
CameraRenderer renderer = new CameraRenderer();

protected override void Render (
	ScriptableRenderContext context, Camera[] cameras
) {
	foreach (Camera camera in cameras) {
		renderer.Render(context, camera);
	}
}
```

我们的摄像机渲染器跟通用渲染管线的可编程渲染器差不多。将来这个写法可以轻松支持为每一个摄像机设置不同渲染类型，比如一个用于第一人称视角，一个用于 3D 地图的叠加，或者前向渲染和延迟渲染。但现在我们先用同样的方式渲染所有摄像机。

2.2. 绘制天空盒（Drawing the Skybox）

CameraRenderer.Render 方法的作用是绘制摄像机看到的所有几何体。为了清晰起见，我们将具体的功能拆分到独立的 DrawVisibleGeometry 里。我们从绘制一个默认天空盒开始，这个工作可以通过执行 context 上的 DrawSkybox 方法实现，这个方法带有 camera 参数。

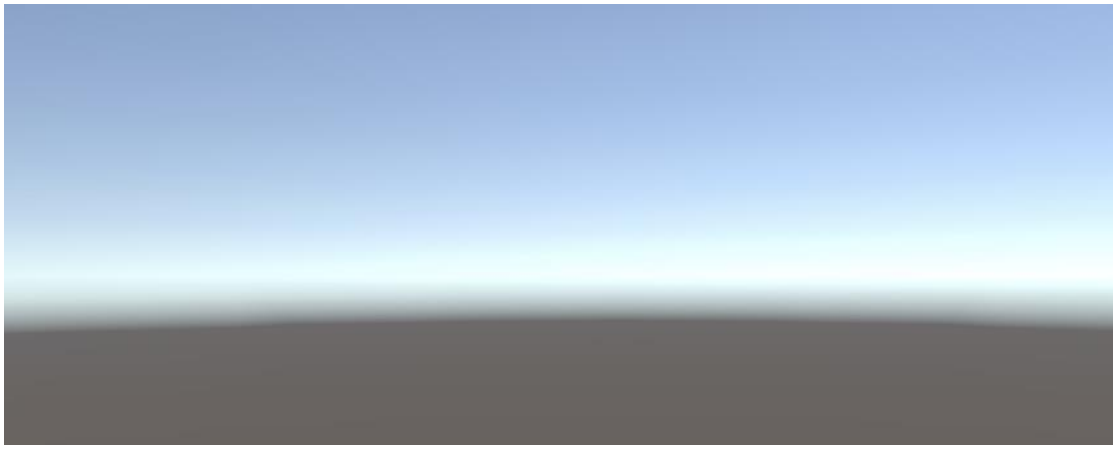
```
public void Render (ScriptableRenderContext context, Camera camera) {
	this.context = context;
	this.camera = camera;

	DrawVisibleGeometry();
}

void DrawVisibleGeometry () {
	context.DrawSkybox(camera);
}
```

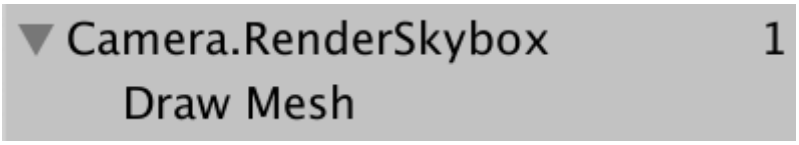
现在还不能显示天空盒，因为我们发给上下文的指令还在缓冲区里。我们必须通过调用上下文的 Submit 来提交要执行的工作队列。我们在 DrawVisibleGeometry 后面调用一个分离的 Submit 方法。

```
public void Render (ScriptableRenderContext context, Camera camera) {
	this.context = context;
	this.camera = camera;

	DrawVisibleGeometry();
	Submit();
}

void Submit () {
```

```
10.     context.Submit();
11. }
```

天空盒终于在 Game 和 Scene 窗口中出现了。当你启用 Frame Debugger，也能看到对应的条目，它作为 Camera.RenderSkybox 被列出，其中包含一个独立的 Draw Mesh，这表示实际的绘制。这些条目对应的是 Game 窗口的渲染，帧调试器不会反映出其他窗口的绘制。





绘制了天空盒

现在我们注意到摄像机的视角不会影响天空盒的渲染，因为我们只给 DrawSkybox 传递了摄像机，但是这只能决定天空盒是否会被绘制，这是由摄像机的清除标记控制的。

为了正确渲染天空盒和整个场景，我们要设置视图-投影矩阵。这个变换矩阵包含摄像机的位置和方向（视图矩阵）和摄像机的透视或正交投影（投影矩阵）。在着色器中，它对应 unity_MatrixVP，这是一个绘制几何体时使用的着色器属性之一。当你选中 frame debugger 里的一条绘制指令，就能查看到这个矩阵数据。

目前 unity_MatrixVP 矩阵总是不变的。我们得通过 SetupCameraProperties 方法将摄像机数据应用到上下文中，这将设置矩阵数据和一些其他属性。在调用 DrawVisibleGeometry 方法之前用一个分离的 Setup 方法来完成这个步骤。

```
1.  public void Render (ScriptableRenderContext context, Camera camera) {
2.      this.context = context;
3.      this.camera = camera;
4.
5.      Setup();
6.      DrawVisibleGeometry();
7.      Submit();
8.  }
9.
10. void Setup () {
11.     context.SetupCameraProperties(camera);
```

```
12. }
```

天空盒正确对齐

2.3. 命令缓冲区（Command Buffers）

上下文会延迟实际的渲染，直到我们提交渲染。在那之前，我们对其为之后的执行进行配置并添加命令。一些类似于绘制天空盒的指令可以通过一条专门的方法执行，但是其他的命令必须通过一个独立的命令缓冲区执行。我们需要这样的一个缓冲区来绘制场景中的其他几何体。

为了获得一个缓冲区，我们需要创建一个新的 CommandBuffer 实例。我们只需要一个缓冲区，所以为 CameraRender 创建一个默认缓冲区并把引用存在字段中。为了在 frame debugger 里分辨，我们给缓冲区一个名称，比如 Render Camera。

```
1.  const string bufferName = "Render Camera";
2.
3.  CommandBuffer buffer = new CommandBuffer {
4.      name = bufferName
5.  };
```

我们可以使用命令缓冲区来注入分析器采样，用来同时在分析器和帧调试器中标记，需要在 Setup 和 Submit 方法开头调用 BeginSample 和 EndSample 方法。两个方法都需要提供相同的采样名，我们使用缓冲区的名称。

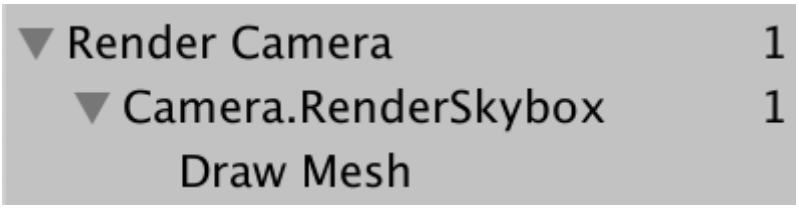
```
1.  void Setup () {
2.      buffer.BeginSample(bufferName);
3.      context.SetupCameraProperties(camera);
4.  }
5.
6.  void Submit () {
7.      buffer.EndSample(bufferName);
8.      context.Submit();
```

```
}
```

为了执行缓冲区，调用上下文的 ExecuteCommandBuffer 并将缓冲区作为传递参数。这步从缓冲区里复制了命令却没有清除它们，如果我们想重复使用，就必须明确的清除它们，因为执行和清理都是一起执行，所以将它们添加到一个方法中更方便。

```
void Setup () {
	buffer.BeginSample(bufferName);
	ExecuteBuffer();
	context.SetupCameraProperties(camera);
}

void Submit () {
	buffer.EndSample(bufferName);
	ExecuteBuffer();
	context.Submit();
}

void ExecuteBuffer () {
	context.ExecuteCommandBuffer(buffer);
	buffer.Clear();
}
```

现在 Camera.RenderSkybox 采样嵌套在 Render Camera 里了。

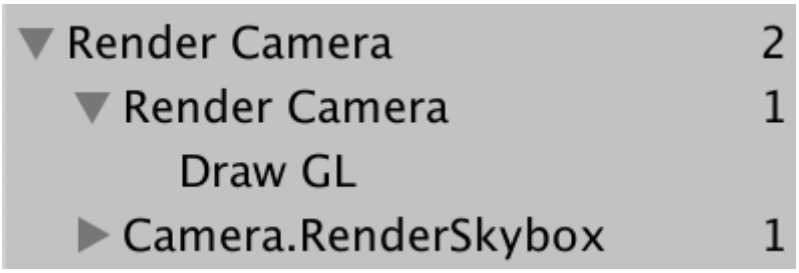


Render Camera 采样

2.4. 清理渲染目标（Clearing the Render Target）

我们绘制的任何内容最后都会渲染到摄像机的渲染目标上，它默认情况下是帧缓冲区，但也可以是一张渲染纹理。之前绘制的任何内容都始终存在，这会干扰我们正在绘制的图像。为了确保正确的渲染，我们必须清理渲染目标来去除旧的内容。这可以通过调用命令缓冲区的 ClearRenderTarget 来实现，这个方法属于 Setup 方法。

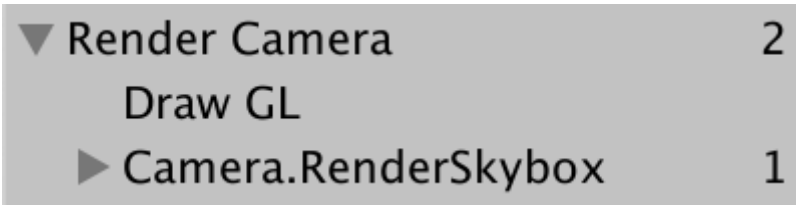
```
void Setup () {
	buffer.BeginSample(bufferName);
	buffer.ClearRenderTarget(true, true, Color.clear);
	ExecuteBuffer();
	context.SetupCameraProperties(camera);
}
```



嵌套在采样中的清理

现在帧调试器为清理操作显示了一条 Draw GL 嵌套在一个额外的 Render Camera 层级里。这是因为 ClearRenderTarget 把样本中的清理用命令缓冲区的名字封装了一层。我们可以把清理指令放在自己的样本之前来避免这个多余的嵌套。这样就可以合并两个相邻的 Render Camera 样本域。

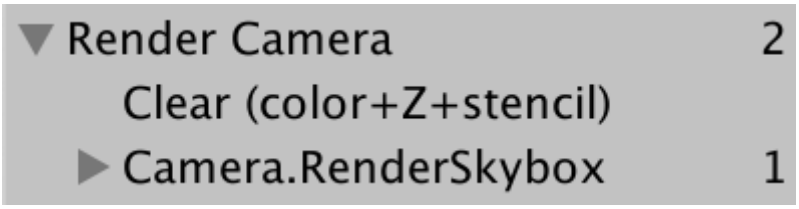
```
1. void Setup () {
2.     buffer.ClearRenderTarget(true, true, Color.clear);
3.     buffer.BeginSample(bufferName);
4.     //buffer.ClearRenderTarget(true, true, Color.clear);
5.     ExecuteBuffer();
6.     context.SetupCameraProperties(camera);
7. }
```



嵌套之外的清理

Draw GL 条目代表使用 Hidden/InternalClear 着色器绘制了一个全屏四边形到渲染目标，但这不是最有效率的清理方法。使用这个方法是因为我们在设置摄像机属性之前执行了清理。如果我们调换这两步的顺序就能获得快速清理的方法。

```
1. void Setup () {
2.     context.SetupCameraProperties(camera);
3.     buffer.ClearRenderTarget(true, true, Color.clear);
4.     buffer.BeginSample(bufferName);
5.     ExecuteBuffer();
6.     //context.SetupCameraProperties(camera);
7. }
```



正确的清理

现在我们看到的 Clear(color+Z+stencil)意味着颜色和深度缓冲区都被清理了。Z 代表深度缓冲区，并且模板数据是同一缓冲区的一部分。

2.5. 剔除（Culling）

目前我们看到了天空盒，但是没有放在场景里的其他物件。我们只要渲染对于摄像机可见的物体而非绘制所有物体。我们从场景中所有带有渲染器组件的物体开始，然后剔除掉那些落在摄像机视锥以外的部分。

为了搞清楚哪些可以被剔除，我们需要用到 ScriptableCullingParameters 这个结构体来追踪多个摄像机设置和矩阵。我们可以调用摄像机的 TryGetCullingParameters 来代替我们给它赋值。某些摄像机设置可能产生退化，因此该方法会返回一个标记是否成功获得结构体的值。为了获取参数数据，我们得通过在参数前加 out 的方式将其设置成输出参数。在一个分离的 Cull 方法中执行，然后返回是否成功。

```
bool Cull () {
	ScriptableCullingParameters p
	if (camera.TryGetCullingParameters(out p)) {
		return true;
	}
	return false;
}
```

当用作输出参数时，可以在参数列表中内联变量声明。

```
bool Cull () {
	//ScriptableCullingParameters p
	if (camera.TryGetCullingParameters(out ScriptableCullingParameters p))
		return true;
	}
	return false;
}
```

在 Render 中的 Setup 之前调用 Cull，如果失败则中止。

```
public void Render (ScriptableRenderContext context, Camera camera) {
	this.context = context;
	this.camera = camera;
```

```
4.
5.      if (!Cull()) {
6.          return;
7.      }
8.
9.      Setup();
10.     DrawVisibleGeometry();
11.     Submit();
12. }
```

实际的剔除是通过上限为调用 Cull 来实现的，这会产生一个 CullingResults 结构。如果参数获取成功，就在 Cull 里执行并将结果存在字段中。在这种情况下，我们需要在其前面加上 ref 来将剔除参数作为引用参数传递。

```
1.  CullingResults cullingResults;
2.
3.  …
4.
5.  bool Cull () {
6.      if (camera.TryGetCullingParameters(out ScriptableCullingParameters p)) {
7.          cullingResults = context.Cull(ref p);
8.          return true;
9.      }
10.     return false;
11. }
```

2.6 绘制几何体（Drawing Geometry）

一旦我们知道哪些可见，我们就可以开始绘制。这是通过在上下文中调用 DrawRenderers 并以剔除结果作为参数来完成的，告诉它使用哪些渲染器。除此之外，我们需要提供绘制设置和筛选设置。它们都是结构体，DrawingSettings 和 FilteringSettings，我们先用他们的默认构造函数。它们都需要作为引用参数传递，在 DrawVisibleGeometry 方法里的绘制天空盒之前执行。

```
1.  void DrawVisibleGeometry () {
2.      var drawingSettings = new DrawingSettings();
3.      var filteringSettings = new FilteringSettings();
4.
5.      context.DrawRenderers(
6.          cullingResults, ref drawingSettings, ref filteringSettings
7.      );
8.
9.      context.DrawSkybox(camera);
10. }
```

我们还没有看到任何东西，因为我们小孩需要声明可以使用哪些着色器 pass。在本篇教程里只支持 Unlit 着色器，我们需要获取 SRPDefaultUnlit pass 的着色器标签 ID，这只要执行一次，然后保存为静态字段。

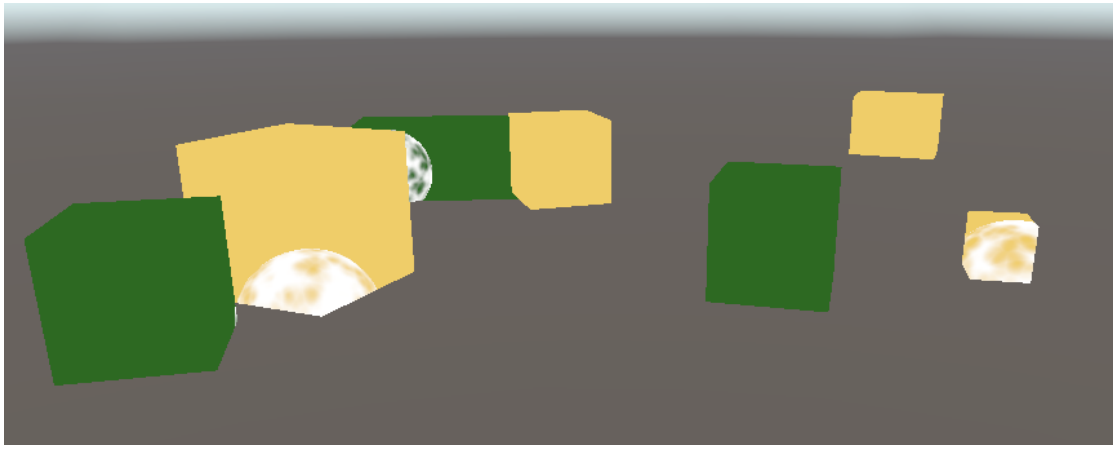
```
static ShaderTagId unlitShaderTagId = new ShaderTagId("SRPDefaultUnlit");
```

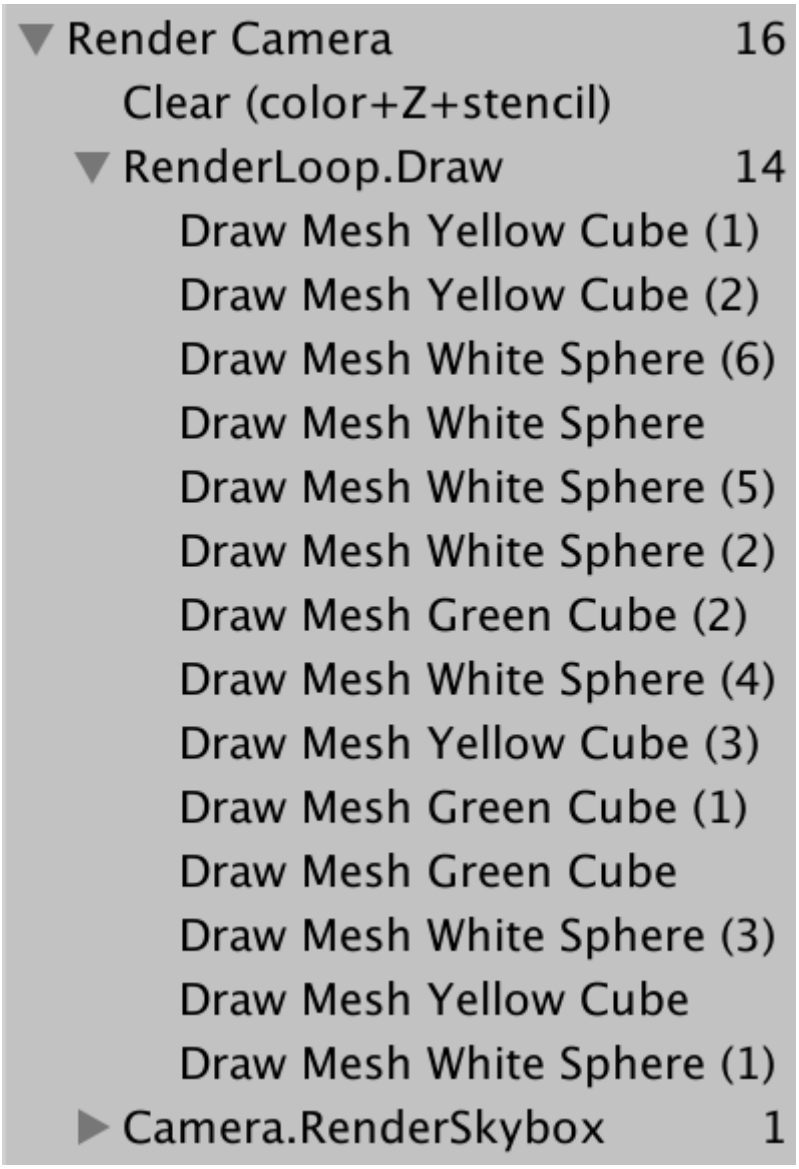
把它作为 DrawingSetttings 构造函数的第一个参数，再加上一个新的 SortingSettings 结构变量。把摄像机传给排序设置的构造函数，因为它决定了是正交还是基于距离的排序。

```
1.  void DrawVisibleGeometry () {
2.      var sortingSettings = new SortingSettings(camera);
3.      var drawingSettings = new DrawingSettings(
4.          unlitShaderTagId, sortingSettings
5.      );
6.      …
7.  }
```

除此之外我们还需要声明允许哪些渲染队列。把 RenderQWueueRange.all 传入 FiltingSettings 的构造函数，这样包含所有队列。

```
var filteringSettings = new FilteringSettings(RenderQueueRange.all);
```



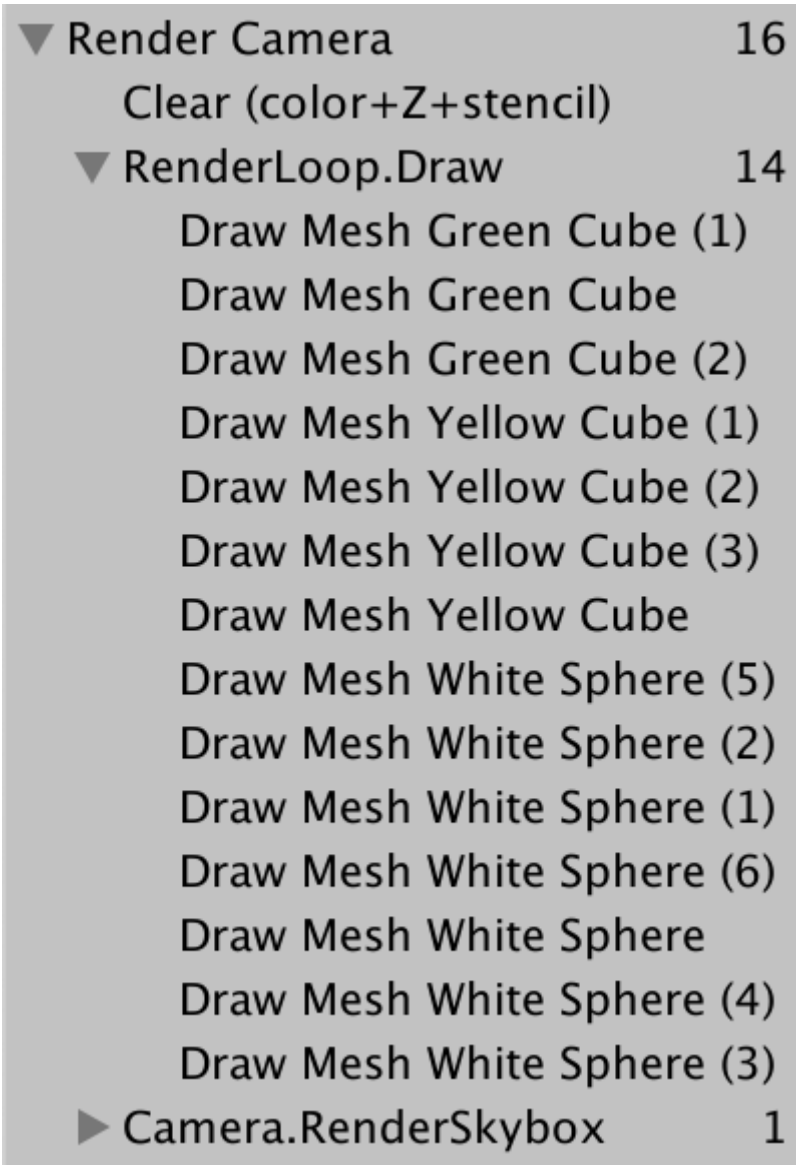


绘制不受光几何体

只有那些使用了不受光着色器的物体被绘制了。在帧调试器里，所有的绘制指令都在 RenderLoop.Draw 组下面列出来了，对于透明物体来说，出现了一些奇怪的现象，我们先看一下这些物体的绘制顺序。你可以按顺序选中帧调试器里的绘制指令或者用方向键来查看绘制顺序。

绘制顺序是随意的。我们通过设置排序设置的 criteria 参数来指定特定的绘制顺序，使用 SortingCriteria.Common.Opaque。

```
1. var sortingSettings = new SortingSettings(camera) {
2.     criteria = SortingCriteria.CommonOpaque
3. };
```



Common opaque 排序

现在物体从前往后或多或少绘制出来了，对不透明物体来说比较完美。如果某个物体在别的物体之后绘制，那么隐藏的片元就会跳过，这样可以加速渲染。Common opaque 排序选项还参考了一些其他条件，比如渲染队列和材质。

2.7 分开渲染不透明物体和透明物体（Drawing Opaque and Transparent Geometry Separately）

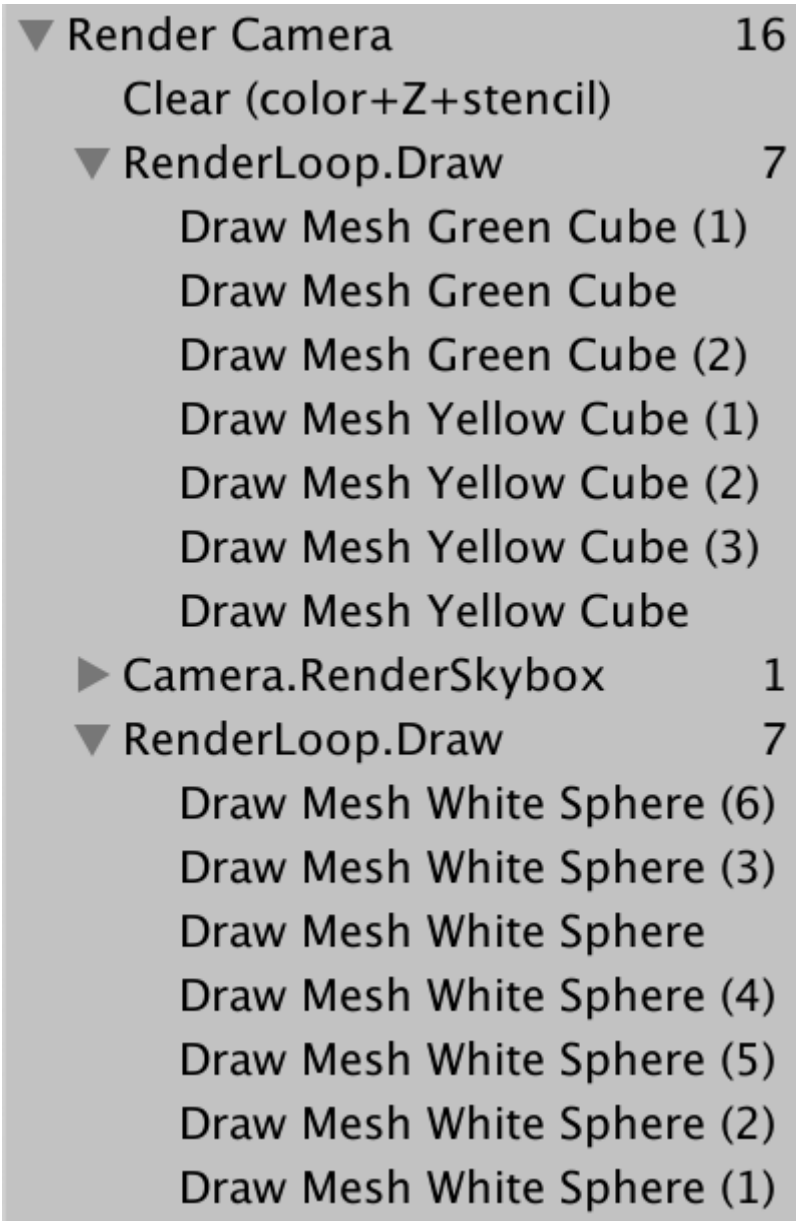
帧调试器告诉我们，透明物体被绘制了，但是天空盒绘制在所有不在不透明物体前面的物体上。天空盒在不透明物体之后绘制，所以跳过了它所有隐藏的片元，但是却覆盖了透明物体。这是因为透明着色器不写入深度缓冲区。它们不会遮挡任何后面的物体，因为我们的视线可以透过它们。解决办法是最先绘制不透明物体，然后是天空盒，最后再是不透明物体。

把 DrawRenderers 方法的参数改成 RenderQueueRange.opaque，我们就可以去除场景中的透明物体。

```
var filteringSettings = new FilteringSettings(RenderQueueRange.opaque);
```

在绘制天空盒之后再次调用 DrawRenderers。在这之前，先将渲染队列范围改成 RenderQueueRange.transparent。同样将排序标准改成 SortingCriteria.CommonTransparent，再设置绘制设置的排序参数。这样反转了透明物体的绘制顺序。

```
context.DrawSkybox(camera);

sortingSettings.criteria = SortingCriteria.CommonTransparent;
drawingSettings.sortingSettings = sortingSettings;
filteringSettings.renderQueueRange = RenderQueueRange.transparent;

context.DrawRenderers(
    cullingResults, ref drawingSettings, ref filteringSettings
);
```



不透明，天空盒，透明

3. 编辑器渲染（Editor Rendering）

现在渲染管线可以正确绘制无光照物体，我们还可以做一些提升使用体验的内容。

3.1. 绘制遗留的着色器

由于我们的管线只支持无光照着色器 pass，使用了不同 pass 的物体就没有被渲染，因此不可见。尽管这是正确的，但它掩盖了场景中的某些对象使用错误的着色器的事实。 因此，

无论如何，还是分别渲染它们。

如果从默认的 Unity 项目开始，然后切换到我们的渲染管线，那么场景中就会有物体使用了错误的着色器。为了覆盖所有 Unity 默认的着色器，我们必须使用 Always, ForwardBase, PrepassBase, Vertex, VertexLMRGBM, VerterLM 这些 pass 的着色器标签 ID。把它们存在一个静态数组里。

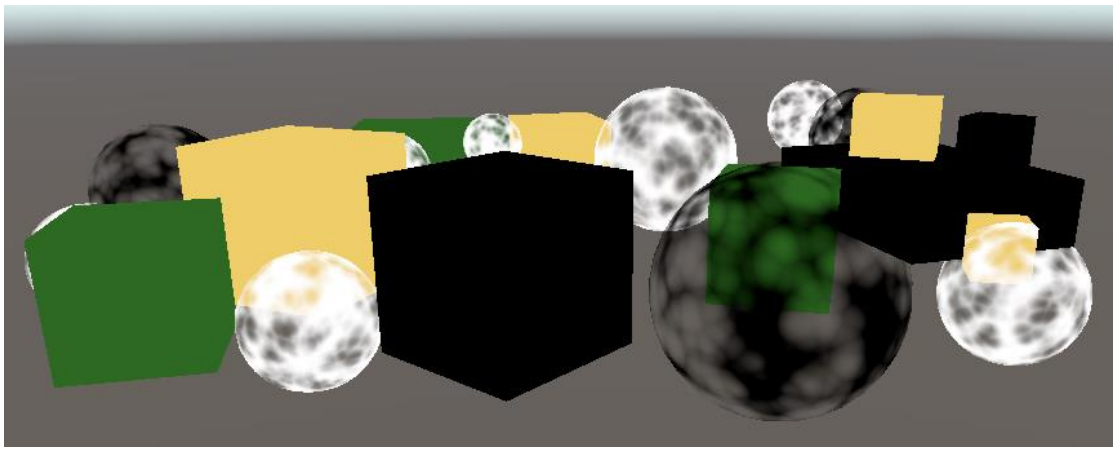
```
static ShaderTagId[] legacyShaderTagIds = {
	new ShaderTagId("Always"),
	new ShaderTagId("ForwardBase"),
	new ShaderTagId("PrepassBase"),
	new ShaderTagId("Vertex"),
	new ShaderTagId("VertexLMRGBM"),
	new ShaderTagId("VertexLM")
};
```

在绘制可见几何体之后，使用一个单独方法绘制所有不受支持的着色器，先从第一个开始。由于 pass 不正确，绘制结果无论如何都是错误的，所以我们不需要关心其他设置，使用默认的过滤设置 FilteringSettings.defaultValue 就可以。

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

	Setup();
	DrawVisibleGeometry();
	DrawUnsupportedShaders();
	Submit();
}

…

void DrawUnsupportedShaders () {
	var drawingSettings = new DrawingSettings(
		legacyShaderTagIds[0], new SortingSettings(camera)
	);
	var filteringSettings = FilteringSettings.defaultValue;
	context.DrawRenderers(
		cullingResults, ref drawingSettings, ref filteringSettings
	);
}
```

我们可以通过调用绘制设置中的 SetShaderPassName，并传入绘制顺序索引和标签作为参入来绘制多个 pass。为数组中所有 pass 执行这步操作，从第二个开始，因为我们在绘制设置的构造过程中已经设置好了第一个 pass。

```
var drawingSettings = new DrawingSettings(
    legacyShaderTagIds[0], new SortingSettings(camera)
);
for (int i = 1; i < legacyShaderTagIds.Length; i++) {
    drawingSettings.SetShaderPassName(i, legacyShaderTagIds[i]);
}
```



Standard 着色器绘制成了黑色

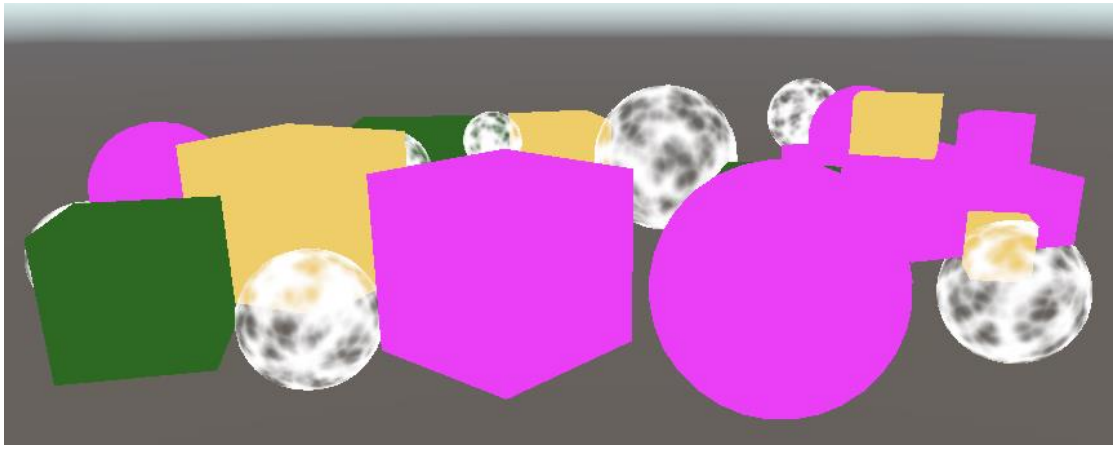
使用 Standard 着色器绘制的物体出现了，但现在是纯黑的，因为渲染管线还没有设置所需的 shader 属性。

3.2. 错误材质（Error Material）

为了清晰地表现那些物体使用了不受支持的材质，我们将使用 Unity 的错误着色器来绘制他们。创建一个新材质，使用这个着色器来作为参数，这个着色器可以通过调用 Shader.Find 方法并传入 Hidden/InternalErrorShader 字符串作为参数来获得。然后将它赋给绘制设置的 overrideMaterial 属性。

```
static Material errorMaterial;

…

void DrawUnsupportedShaders () {
    if (errorMaterial == null) {
        errorMaterial =
            new Material(Shader.Find("Hidden/InternalErrorShader"));
    }
    var drawingSettings = new DrawingSettings(
        legacyShaderTagIds[0], new SortingSettings(camera)
    ) {
        overrideMaterial = errorMaterial
```

```
    };
    …
}
```

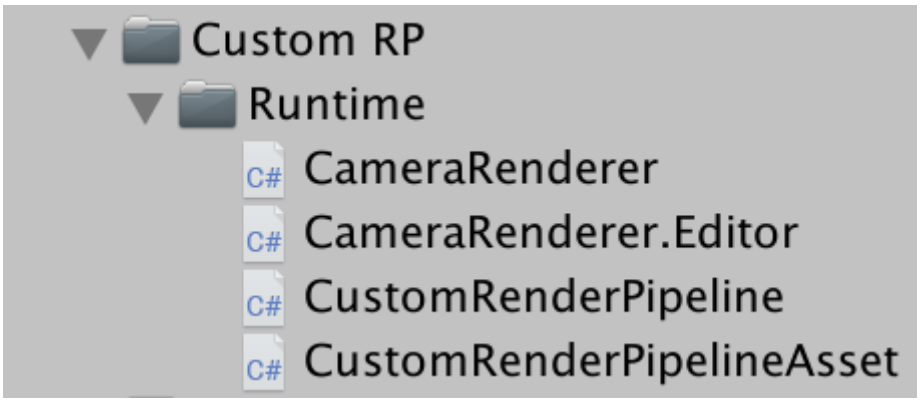


用洋红色的错误着色器渲染

现在所有不合理的物体都可见并且明显错误了。

3.3. 分部类（Partial Class）

在开发中绘制不合理物体很有用，但在发布版应用里可不是。所以我们把所有 CameraRenderer 里只在编辑器种生效的代码放到一个单独的分部类文件中。先复制原始的 CameraRenderer 脚本文件，然后重命名为 CameraRenderer.Editor。



一个类，两个脚本文件

然后把原有 CameraRenderer 改成分部类，并且从中移除标签数组，错误材质和 DrawUnsupportedShaders 方法。

```
public partial class CameraRenderer { … }
```

清理另一个分部类文件，让它只包含刚才移除的内容。

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

```
3.
4.  partial class CameraRenderer {
5.
6.      static ShaderTagId[] legacyShaderTagIds = { … };
7.
8.      static Material errorMaterial;
9.
10.     void DrawUnsupportedShaders () { … }
11. }
```

编辑器部分的内容只需要在编辑器中存在，所以用 UNITY_EDITOR 做条件判断。

```
1.  partial class CameraRenderer {
2.
3.  #if UNITY_EDITOR
4.
5.      static ShaderTagId[] legacyShaderTagIds = { … }
6.      };
7.
8.      static Material errorMaterial;
9.
10.     void DrawUnsupportedShaders () { … }
11.
12. #endif
13. }
```

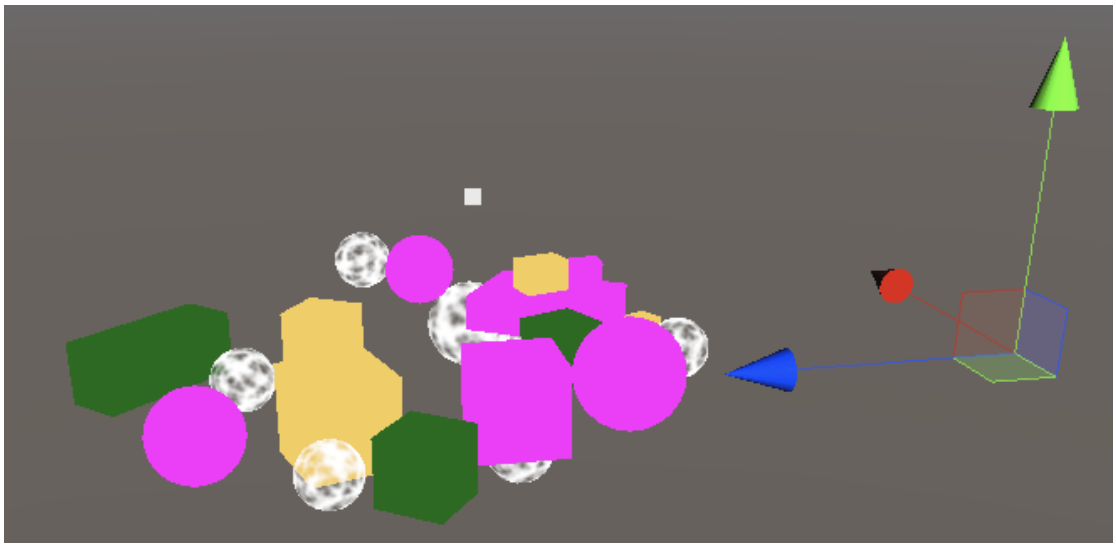
然而现在编译版本会失败，因为另外那部分始终包含了对 DrawUnsupportedShaders 的调用，而它只存在与编辑器中。为了解决这个问题，我们将这方法也分部化。为此，我们在方法前面声明 partial 关键字，这类似于抽象方法的声明。我们可以在类定义的任何地方这么做，所以我们把它放在编辑器部分。完整方法声明同样需要用 partial 关键字标记。

```
1.      partial void DrawUnsupportedShaders ();
2.
3.  #if UNITY_EDITOR
4.
5.      …
6.
7.      partial void DrawUnsupportedShaders () { … }
8.
9.  #endif
```

现在构建项目成功了。编译会剥离所有未以完整声明结尾的分部方法。

3.4. 绘制辅助工具（Drawing Gizmos）

目前我们的渲染管线还没有绘制辅助工具，不论是在场景窗口还是游戏窗口，即使启用了它们。



场景中没有辅助工具

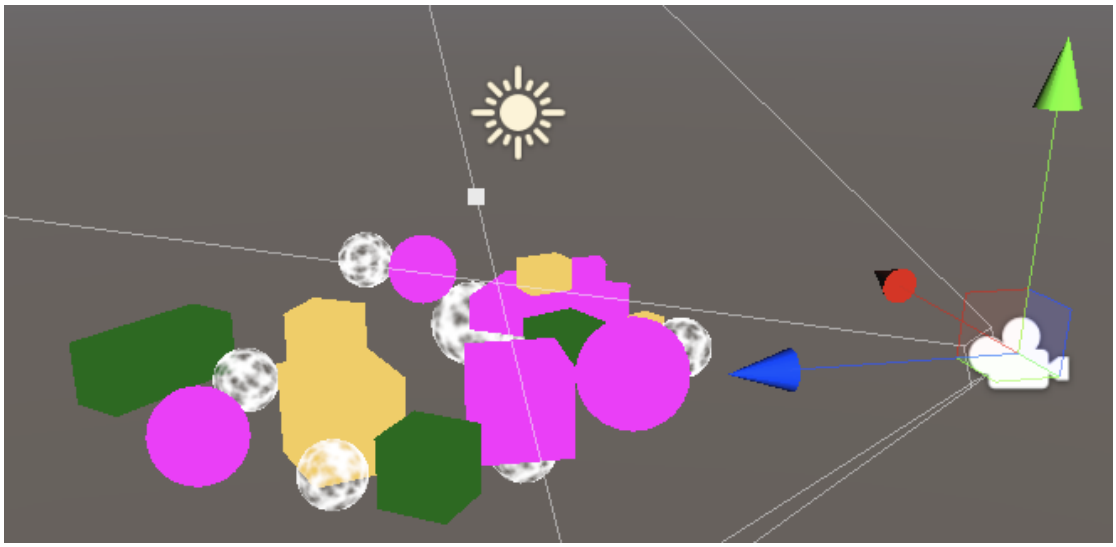
我们可以通过调用 UnityEditor.Handles.ShouldRenderGizmos 来判断是否需要绘制 gizmos。如果需要，我们就调用 context 里的 DrawGizmos，传入摄像机参数和需要渲染的 gizmo 子集参数。它包含两个子集，图像预处理和后处理。目前我们还不支持图像效果，所以我们两个都调用。只在编辑器方法中执行。

```
using UnityEditor;
using UnityEngine;
using UnityEngine.Rendering;

partial class CameraRenderer {

	partial void DrawGizmos ();

	partial void DrawUnsupportedShaders ();

#if UNITY_EDITOR

	…

	partial void DrawGizmos () {
		if (Handles.ShouldRenderGizmos()) {
			context.DrawGizmos(camera, GizmoSubset.PreImageEffects);
			context.DrawGizmos(camera, GizmoSubset.PostImageEffects);
		}
	}
```

```
21.
22.     partial void DrawUnsupportedShaders () { … }
23.
24. #endif
25. }
```

Gizmos 应该在所有物体之后绘制。

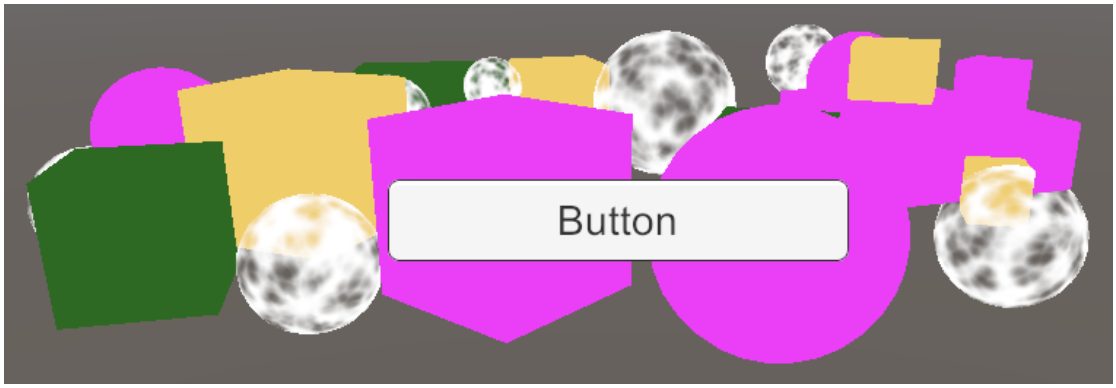
```
1.  public void Render (ScriptableRenderContext context, Camera camera) {
2.      …
3.
4.      Setup();
5.      DrawVisibleGeometry();
6.      DrawUnsupportedShaders();
7.      DrawGizmos();
8.      Submit();
9.  }
```



‘包含了 Gizmos 的场景

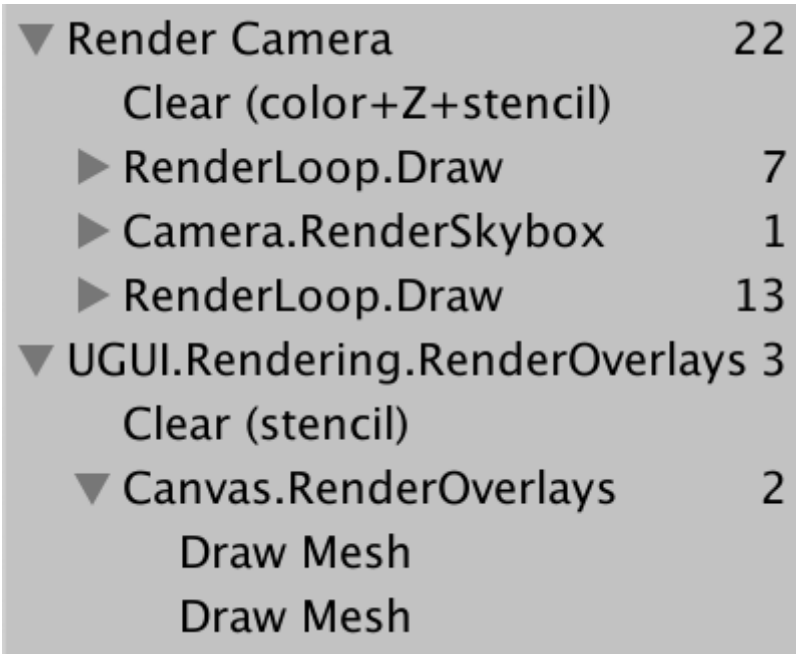
3.5. 绘制 Unity UI

我们注意到的另一个内容是 Unity 的游戏内 UI。比如，创建一个简单的 UI，通过 GameObject/UI/Button 来创建一个按钮，它在游戏窗口中显示，但不在场景窗口中。



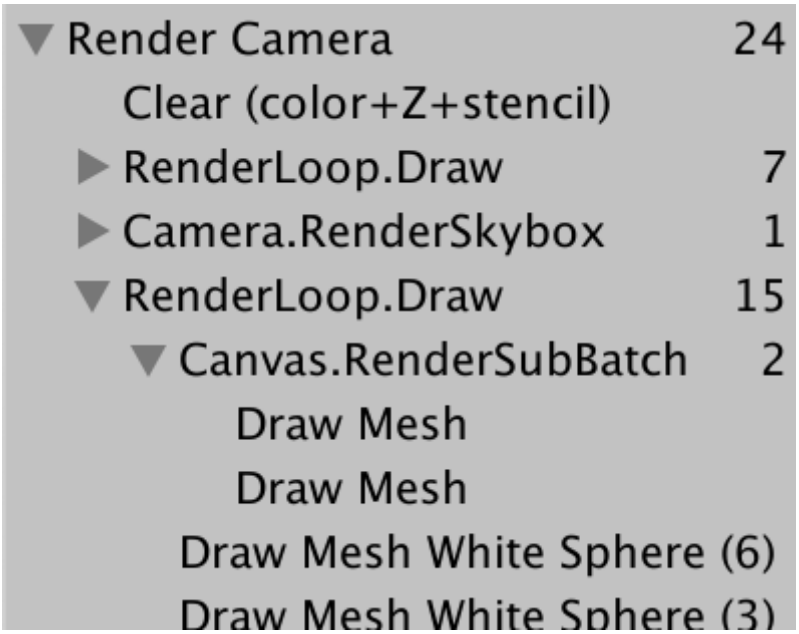
游戏窗口中的 UI 按钮

帧调试器告诉我们，UI 是被单独绘制的，并不在我们的渲染管线里。



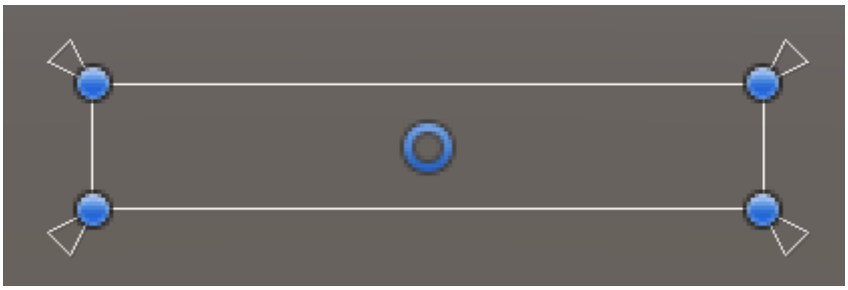
帧调试器里的 UI

不管怎样，现在的情况是画布组件的 Render Mode 设置成了默认的 Screen Space – Overlay。把它改成 Screen Space – Camera，然后使用主摄像机作为渲染摄像机，这样会使其变成透明集合体的一部分。



帧调试器里的 Screen-space-camera UI

当 UI 在场景窗口中渲染时，总是使用 World Space，所以通常都很大。但当我们通过场景窗口编辑 UI 时，它没有被绘制。



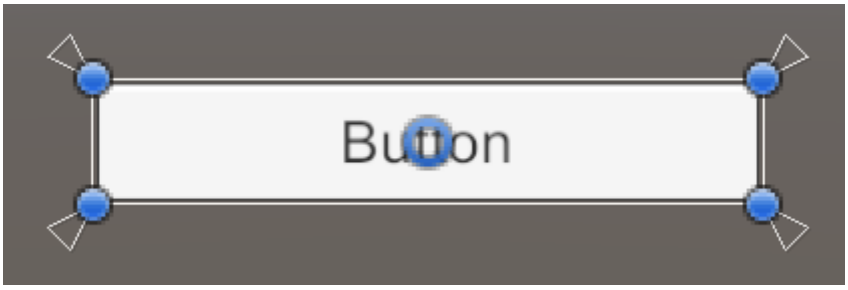
场景窗口中 UI 不可见

当在场景窗口中渲染时，我们必须明确地将 UI 添加到世界几何坐标系里，这通过调用 ScriptableRenderContext.EmitWorldGeometryForSceneView 方法并传入摄像机来实现。同样只在编辑器模式下执行 PrepareForSceneWindo 方法。我们只在摄像机的 cameraType 属性等于 CameraType.SceneView 时才进行渲染。

```
1.  partial void PrepareForSceneWindow ();
2.
3.  if UNITY_EDITOR
4.
5.  …
6.
7.  partial void PrepareForSceneWindow () {
8.      if (camera.cameraType == CameraType.SceneView) {
9.          ScriptableRenderContext.EmitWorldGeometryForSceneView(camera);
10.     }
11. }
```

因为这可能会向场景中添加几何体，所以必须在剔除之前执行。

```
1.  PrepareForSceneWindow();
2.  if (!Cull()) {
3.      return;
4.  }
```



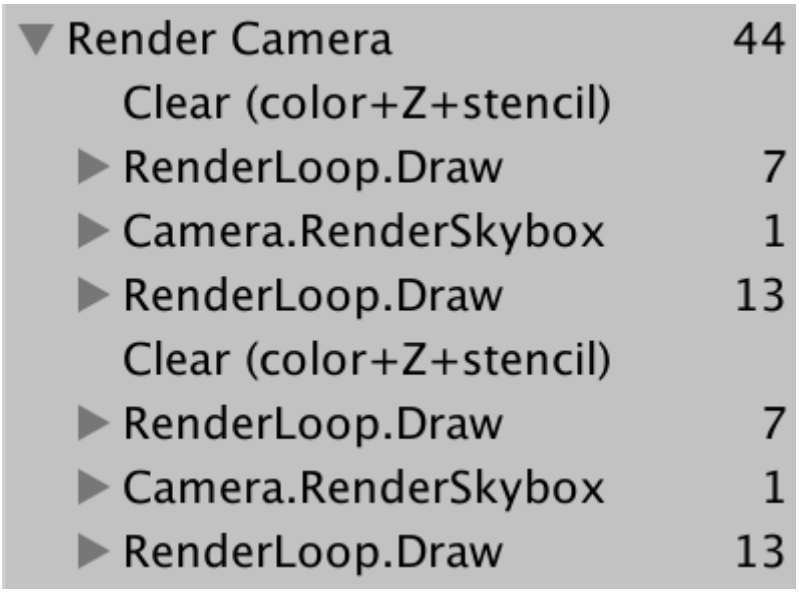
场景中 UI 可见

4. 多摄像机（Mutliple Cameras）

场景中可以存在不止一个摄像机。如果这样，我们需要确保它们能够同时工作。

4.1. 两个摄像机

每个摄像机都有一个 Depth 值，对于默认的主摄像机来说是-1。它们按照深度值的升序来渲染。为了弄明白这一点，复制 Main Camera，重命名为 Secondary Camera，然后设置 Depth 为 0。给它另一个标签也是个好主意，因为 MainCamera 应该只被用于一个摄像机。



两个摄像机都被包含在同一个采样组里

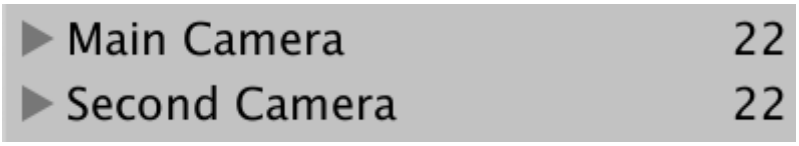
场景现在被渲染了两次。图像结果和以前完全一样，因为在两次渲染之间被清理了。帧调试器表明了这一点，但因为相同名字的采样组会被合并，所以我们最终看到的是一个 Render Camera 组。

如果每个摄像机有各自的组会更清晰。为了实现这个，添加一个仅编辑器的方法 PrepareBuffer，缓冲区名字等于摄像机名字。

```
	partial void PrepareBuffer ();

#if UNITY_EDITOR

	…

	partial void PrepareBuffer () {
		buffer.name = camera.name;
	}

#endif
```

在准备场景窗口之前调用。

```
PrepareBuffer();
PrepareForSceneWindow();
```



每个摄像机独立的采样区

4.2.处理可变缓冲区名称（Dealing with Changing Buffer Names）

尽管帧调试器现在为每个摄像机显示了一个单独的样本层级结构，但当我们进入运行模式时，Unity 控制台里全是警告，认为 BeginSample 和 EndSample 计数必须匹配。我们的样本和缓冲区使用了不同的名称，这会造成困扰。除此之外，每次访问摄像机的 name 属性时，我们都分配了内存，所以我们不想在构建版本中这样做。

为了应对上述两个问题，我们添加个 SampleName 字符串属性。如果在编辑模式中，就在 PrepareBuffer 里设置它和缓冲区的名字，否则就使用一个 Render Camera 的常量。

```
#if UNITY_EDITOR

```

```
	…

	string SampleName { get; set; }

	…

	partial void PrepareBuffer () {
		buffer.name = SampleName = camera.name;
	}

#else

	const string SampleName = bufferName;

#endif
```

对 Setup 和 Submit 里的样本使用 SampleName。

```
void Setup () {
	context.SetupCameraProperties(camera);
	buffer.ClearRenderTarget(true, true, Color.clear);
	buffer.BeginSample(SampleName);
	ExecuteBuffer();
}

void Submit () {
	buffer.EndSample(SampleName);
	ExecuteBuffer();
	context.Submit();
}
```

检查 Profiler（Window/Analysis/Profiler），我们可以发现不同，先运行编辑器模式。切换到 Hierarchy 模式，对 GC Alloc 列进行排序。你会看到一个调用 GC.Alloc 两次的条目，一共分配了 100 字节，这是由于检索摄像机名称引起的。再往下看你会看到具体样本：Main Camera 和 Secondary Camera。

Overview	Total	Self	Calls	GC Alloc
▼ PlayerLoop	73.0%	0.3%	2	148 B
▼ RenderPipelineManager.DoRenderLoop_In	2.1%	0.6%	1	100 B
GC.Alloc	0.0%	0.0%	2	100 B
▶ Second Camera	0.1%	0.0%	1	0 B
▶ Main Camera	0.1%	0.0%	1	0 B
RenderLoop.CleanupNodeQueue	0.0%	0.0%	2	0 B

具有单独样本和 100B 分配的 Profiler

接下来，构建一个启用 Development Build 和 Autoconnect Profiler 的版本。运行，然后确保

Profiler 已连接并正在记录。这是我们看不到 100 字节的内存分配并且只有一个单独的 Render Camera 样本。

Overview	Total	Self	Calls	GC Alloc
▼ PlayerLoop	99.9%	0.7%	1	48 B
▼ PostLateUpdate.FinishFrameRendering	4.5%	0.1%	1	48 B
GC.Alloc	0.0%	0.0%	1	48 B
▶ UIEvents.IMGUIRenderOverlays	0.0%	0.0%	1	0 B
▶ UIEvents.CanvasManagerRenderOverlay	0.0%	0.0%	1	0 B
▼ RenderPipelineManager.DoRenderLoop	4.1%	0.6%	1	0 B
▶ Render Camera	0.3%	0.0%	2	0 B
RenderLoop.CleanupNodeQueue	0.0%	0.0%	2	0 B

构建版本的 Propfiler

通过将摄像机名称检索包装在名为 Editor Only 的探查器示例中，我们可以清楚地知道，我们仅在编辑器中分配内存，而不在构建中分配内存。 在这种情况下，我们需要从 UnityEngine.Profiling 命名空间调用 Profiler.BeginSample 和 Profiler.EndSample。 仅 BeginSample 需要传递名称。

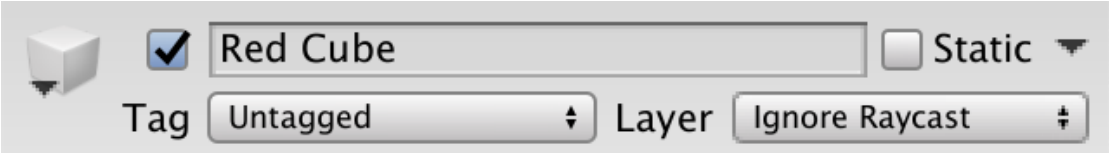
```
1.  using UnityEditor;
2.  using UnityEngine;
3.  using UnityEngine.Profiling;
4.  using UnityEngine.Rendering;
5.
6.  partial class CameraRenderer {
7.
8.      …
9.
10. #if UNITY_EDITOR
11.
12.     …
13.
14.     partial void PrepareBuffer () {
15.         Profiler.BeginSample("Editor Only");
16.         buffer.name = SampleName = camera.name;
17.         Profiler.EndSample();
18.     }
19.
20. #else
21.
22.     string SampleName => bufferName;
23.
24. #endif
25. }
```

Overview	Total	Self	Calls	GC Alloc
PlayerLoop	78.7%	0.7%	2	148 B
RenderPipelineManager.DoRenderLoop_In	5.3%	1.3%	1	100 B
Editor Only	0.0%	0.0%	2	100 B
GC.Alloc	0.0%	0.0%	2	100 B
Second Camera	0.2%	0.0%	1	0 B
Main Camera	0.3%	0.0%	1	0 B

Edlitor-only 下的内存分配

4.3. 层级（Layers）

摄像机也可以被配置成只能看到某些层级的物体，这可以通过调整其 Culling Mask 来完成。为了查看实际效果，我们把所有使用 Standard 着色器的对象移动到“Ignore Raycast”层级。

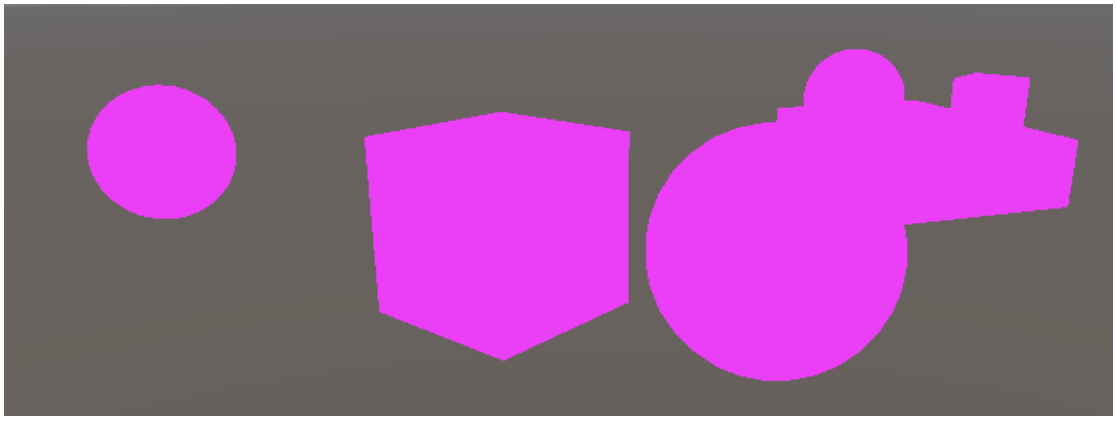


层级切换到 Ignore Raycast

从 Main Camera 的 Culling Mask 中去除该层级。

剔除 Ignore Raycast 层级

由于 Secondary Camera 最后渲染，我们最后只看到了非法对象。



在游戏窗口中只有 Ignore Raycast 层级

4.4. 清理标记（Clear Flags）

我们可以通过调整第二个摄像机的清理标记来同时获得所有摄像机的渲染结果。它们由 CameraClearFlags 枚举定义，我们可以通过摄像机的 clearFlags 属性检索该枚举。在 Setup 方法的清理之前执行。

```
1.  void Setup () {
2.      context.SetupCameraProperties(camera);
3.      CameraClearFlags flags = camera.clearFlags;
4.      buffer.ClearRenderTarget(true, true, Color.clear);
5.      buffer.BeginSample(SampleName);
6.      ExecuteBuffer();
7.  }
```

Camera ClearFlags 定义了 4 个值。从 1 到 4 分别是 Skybox, Color, Depth 和 Nothing。这些事实上不是完全独立的标记值，而是表现了清理内容的递减。除了最后一种情况，深度缓冲区都会被清理，所以标记值为不大于 Depth。

```
1.  buffer.ClearRenderTarget(
2.      flags <= CameraClearFlags.Depth, true, Color.clear
3.  );
```

只有当标记设置为 Color 时，我们才真正需要清除颜色缓冲区。因为天空盒的存在，无论如何最后都会替换掉以前的颜色数据。

```
1.  buffer.ClearRenderTarget(
2.      flags <= CameraClearFlags.Depth,
3.      flags == CameraClearFlags.Color,
4.      Color.clear
5.  );
```

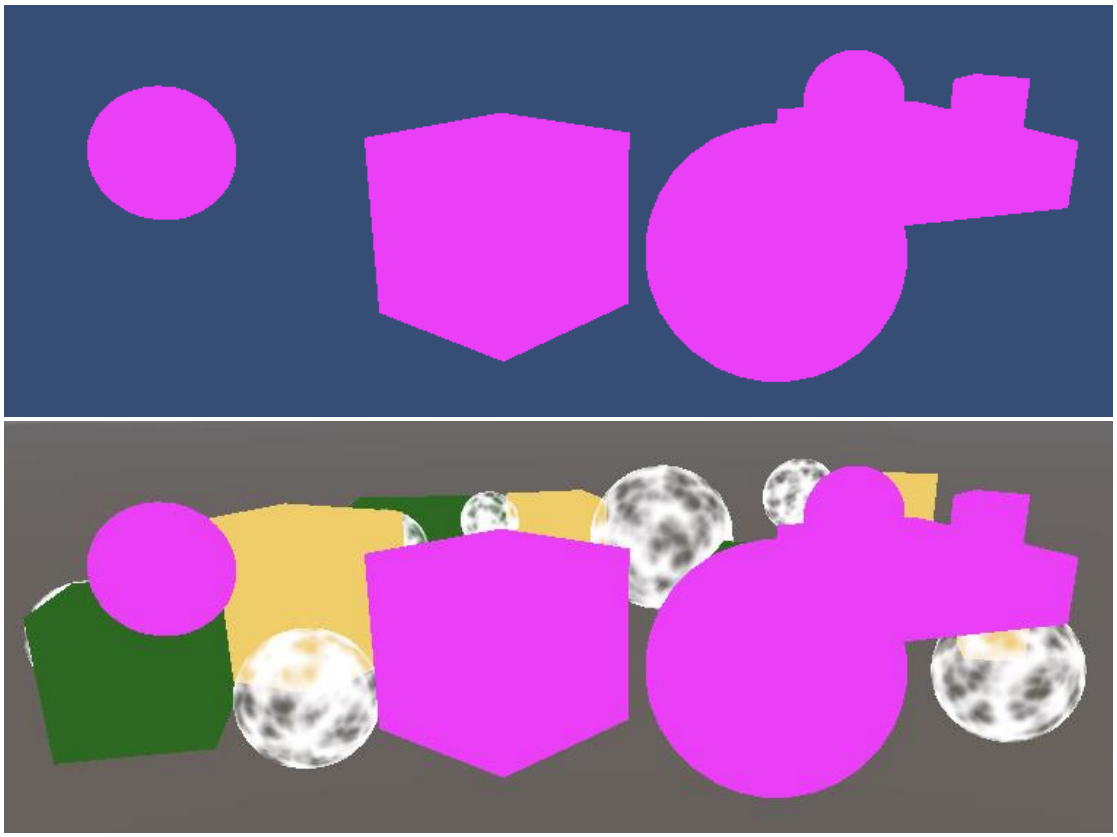
如果要清除为纯色，则必须使用相机的背景色。 但是，由于我们要在线性颜色空间中进行

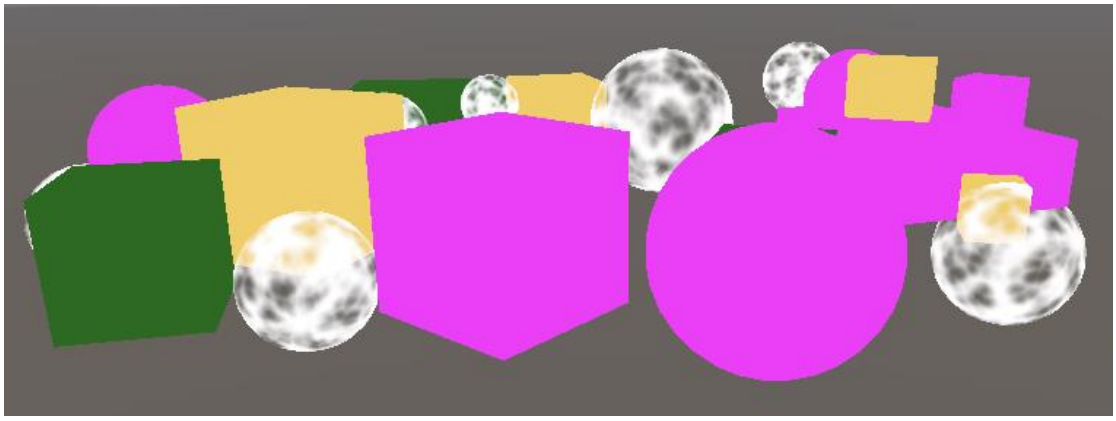
渲染，因此必须将该颜色转换为线性空间，因此最终需要使用camera.backgroundColor.linear。在所有其他情况下，颜色无关紧要，因此我们可以使用 Color.clear 就足够了。

```
buffer.ClearRenderTarget(
	flags <= CameraClearFlags.Depth,
	flags == CameraClearFlags.Color,
	flags == CameraClearFlags.Color ?
		camera.backgroundColor.linear : Color.clear
);
```

因为 Main Camera 是第一个渲染的摄像机，所以其清理标记应设置为 Skybox 或 Color。 启用帧调试器后，我们总是从清除缓冲区开始，但这通常不能保证。

Secondary Camera 的清理标记决定了两个摄像机渲染的组合方式。对于天空盒或颜色，先前的结果将被完全替换。如果只清除深度，Secondary Camera 将会正常渲染，但不会绘制天空盒，因此先前的内容会显示为背景。当什么都不清理时，深度缓冲区将被保留，因此最后无光照对象将遮挡无效对象，像是一台摄像机绘制的一样。然而，由于前一个摄像机绘制的透明对象没有深度信息，所以表现与之前的天空盒相似。





清理 Color，Depth 和 Nonthing

通过调整相机的 Viewport Rect，也可以将渲染区域缩小到整个渲染目标的一小部分。其余渲染目标保持不受影响。 在这种情况下，将使用 Hidden/InternalClear 着色器进行清除。模板缓冲区用于将渲染限制在视口区域。

Secondary camera 的缩小视口，清理标记为 Color

请注意，每帧渲染一台以上的摄像机意味着必须同时进行多次剔除，设置，分类等操作。 每个独立的视角使用一台摄像机通常是最有效率的方法。

下一篇教程是 Draw Calls。