

Custom Pipeline

Taking Control of Rendering

- 建立render pipeline asset並實例它
- 渲染攝影機所見的畫面
- 執行culling、filtering 和sorting
- 區分不透明物件、透明物件和無效的pass
- 執行不止一個攝影機

關於創建自定義腳本化渲染管線的系列教程的第一部分。它涵蓋了我們將在未來擴展的基本渲染管道的初始創建。

1 新的渲染管線

要渲染任何東西，Unity必須決定他要畫的形狀是什麼、要在哪裡畫、何時要畫等等設定。這會根據你有包含哪些效果而使得渲染變得複雜。光線光源、陰影、透明物件、屏幕特效、體積特效等等，為了確立好他們操作的優先順序來完成最後的成像，這就是渲染管線在做的事情。

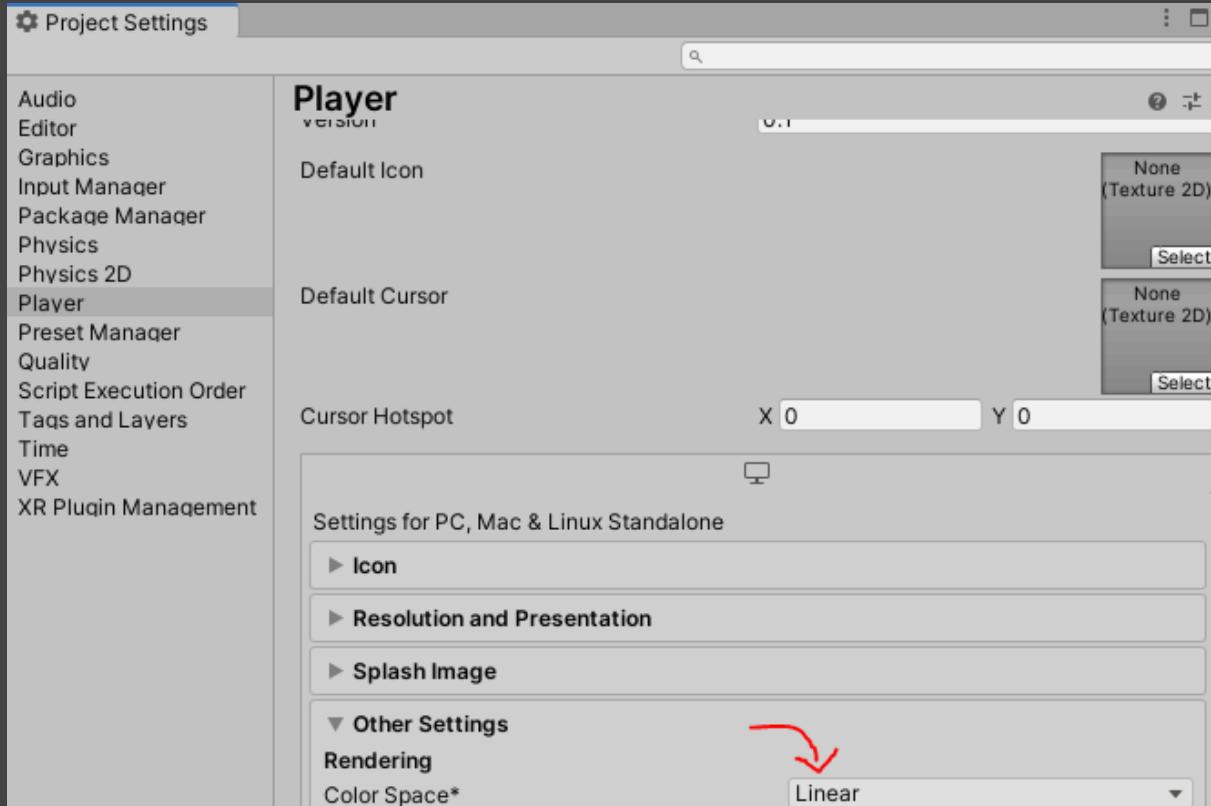
在過去，Unity只支援很少的內建渲染物件方式。Unity2018提出了Scriptable Render pipeline - SRP，雖然我們依然要依賴Unity的基礎建設，但我們可以自己實踐想要的作法。在Unity 2019中Universal RP預設是用來取代legacy RP。

這個教程的目的是使用forward rendering繪制無光照的模型，並將這個做為渲染管線的最小基礎雛型。若是成功，則後面能添加光照、陰影和不同的渲染方式與更高級的功能

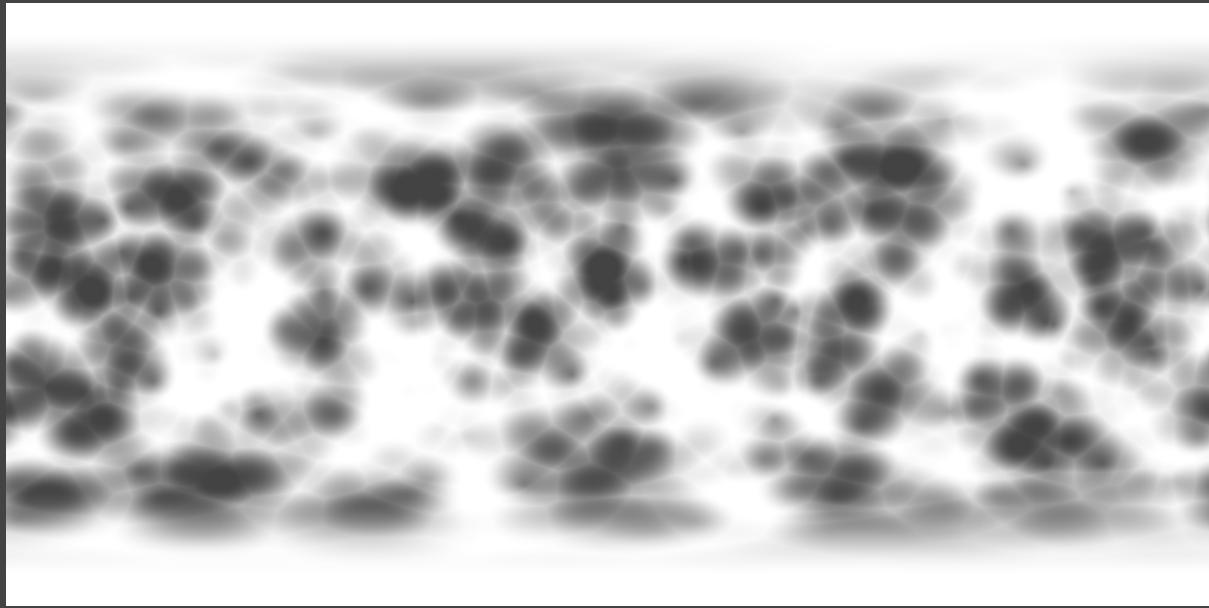
1.1 專案設定

新建一個3D專案在Unity 2019.4或以上的版本，因為我們要客製自己的渲染管線，所以不要選擇任何RP Template來初始專案，開啟專案後確認Package Manager中只有Unity UI這個Package。

接著我們需要設定專案的Color space，將他改為Linear。前往Edit/Project Settings 接著找到Player，切換到Other Settings中調整。



我們要先用物件填滿default的空場景，混合使用standard、unlit opaque和transparent材質在物件上。因為Unlit/Transparent只能作用在texture上，所以使用了下方uv sphere貼圖。

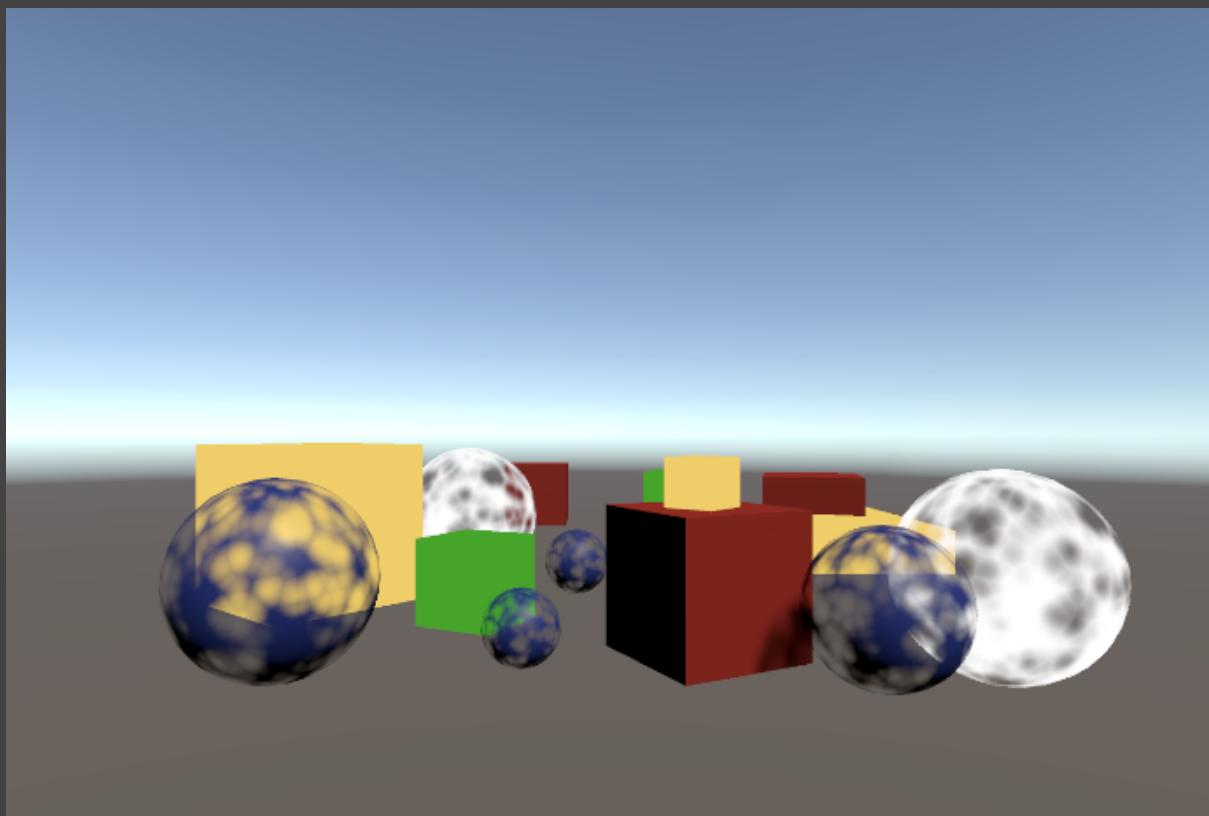


紅色材質使用的是Standard shader

黃色、綠色材質使用的是Unlit/Color shader

藍色材質使用的是Standard shader 並將Rendering Mode設為Transparent

白色材質使用的是Unlit/Transparent shader



1.2 Pipeline Asset

Graphics

Scriptable Render Pipeline Settings

None (Render Pipeline Asset)

Unity現在使用的是預設渲染管線，為了取代他，我們需要製作出自己的渲染管線。我先大致配置好目前Asset Folder結構，這個結構與Unity的Universal RP是一致的。並建立一個C#腳本CustomRenderPipelineAsset在Runtime folder下。



這個腳本的型別必須是由UnityEngine.Rendering中RenderPipelineAsset擴展出來的。

```
using UnityEngine.Rendering;

public class CustomRenderPipelineAsset : RenderPipelineAsset
{
}
```

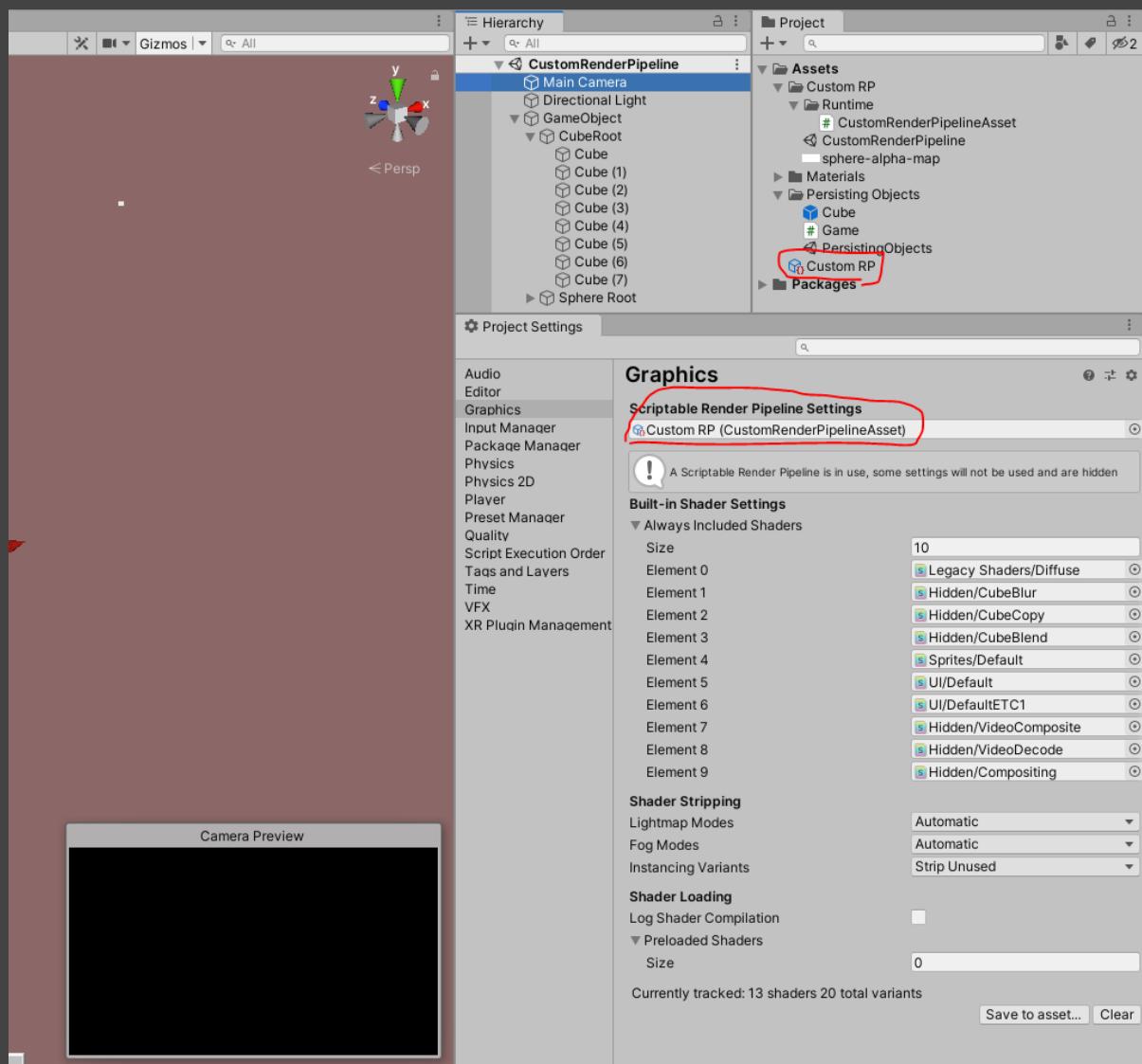
Pipeline Asset的主要目的是為了讓Unity能取得負責渲染工作的渲染管線實例物件。這個asset本身只處理和存放設定。為了讓Unity取得我們的客製化管線，我們需要override CreatePipeline方法，這個方法會回傳RenderPipeline實例，由於我們還沒有自定義出RenderPipeline，所以這邊我們先回傳null。

```
protected override RenderPipeline CreatePipeline()
{
    return null;
}
```

現在，我們需要將我們的客製化管線加到Project中，所以我們需要在CustomRenderPipeline上方加上CreateAssetMenu這個屬性。並設定menuName將生成Pipeline Asset功能放在Rendering/Custom Render Pipeline中

```
[CreateAssetMenu(menuName = "Rendering/Custom Render Pipeline")]
public class CustomRenderPipelineAsset : RenderPipelineAsset
```

接著我們需要將新創立出來的asset放到Graphics Project settings中。



將預設的RP替換成Custom RP後，第一件事是會發現許多設定消失了，可以看到info訊息所描述的內容。第二件事是Game、Scene與Camera視窗中都失去了功用。同時你打開Frame Debug會發現沒有繪製任何東西在Game view。

1.3 Render Pipeline Instance

建立一個CustomRenderPipeline類別並將這個腳本和剛剛的CustomRenderPipelineAsset放在同一個目錄下，這個是我們要回傳的RP Instance，所以他需要extend RenderPipeline。



```
using UnityEngine.Rendering;

public class CustomRenderPipeline : RenderPipeline
{
```

```
}
```

RenderPipeline中定義了一個受保護的抽象方法Render, 我需要override這個方法來建立pipeline。他有2個參數: ScriptableRenderContext和Camera陣列, 我們先建立空的方法吧。

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

接著我們需要讓CustomRenderPipelineAsset的CreatePipeline方法中, 回傳我們的CustomRenderPipeline實例, 這是一個有效有功用的管線, 雖然他還無法渲染任何東西。

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

2 Rendering

在每個frame中Unity都會在RP Instance內呼叫Render。它會傳遞context struct來提供與原生引擎的連結來讓我使用渲染。同時也會傳遞一個攝影機陣列，因為一個場景中可能有多個在使用的攝影機。RP需要負責渲染所有已排序並激活起來的攝影機。

2.1 Camera Renderer

每個攝影機都是獨立渲染的。不讓CameraRenderPipeline負責一次渲染多個前向渲染攝影機，我們需要新的class來渲染單一攝影機，這邊命名為CameraRenderer，並給他一個public Render方法。

```
using UnityEngine;
using UnityEngine.Rendering;

public class CameraRenderer
{
    ScriptableRenderContext context;
    Camera camera;

    public void Render(ScriptableRenderContext _context, Camera _camera)
    {
        context = _context;
        camera = _camera;
    }
}
```

在CameraRenderPipeline中創建一個實例來執行渲染，並使用他來渲染所有的攝影機。

```
private CameraRenderer cameraRenderer = new CameraRenderer();

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

我們的攝影機現在已經粗略的和Universal RP相似，這個方法可以支援許多不同攝影機的渲染方式。例如，一支攝影機給第一人稱視角，一支攝影機用來繪制3D Map，或是Forward與Deferred渲染，但在現在我們將所有的攝影機都用相同的方式渲染。

2.2 畫天空盒

CameraRender.Render的工作是繪製出所有攝影機能看到的幾何物件。分離出一個特殊的task在DrawVisibleGeometry方法。我們要讓他先繪製出天空盒，繪製天空盒的方法可以透過context以攝影機作為傳入參數去呼叫DrawSkybox方式。

```
public void Render(ScriptableRenderContext _context, Camera _camera)
{
    context = _context;
    camera = _camera;

    DrawVisibleGeometry();
}

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

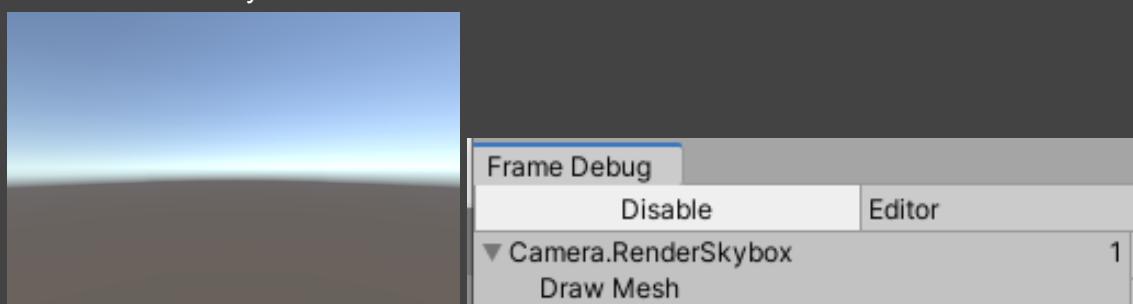
現在天空盒還不會出現在畫面上，因為這個context請求還在buffer中，我們需要讓context發送submit來啟用剛剛的請求。

```
public void Render(ScriptableRenderContext _context, Camera _camera)
{
    context = _context;
    camera = _camera;

    DrawVisibleGeometry();
    Submit();
}

private void Submit()
{
    context.Submit();
}
```

終於我們可以在Game view和Scene view中看見天空盒了，你也可以在frame debug中看到Camera.RenderSkybox，他繪制了一個mesh在底下，表示他花了一個Draw call。

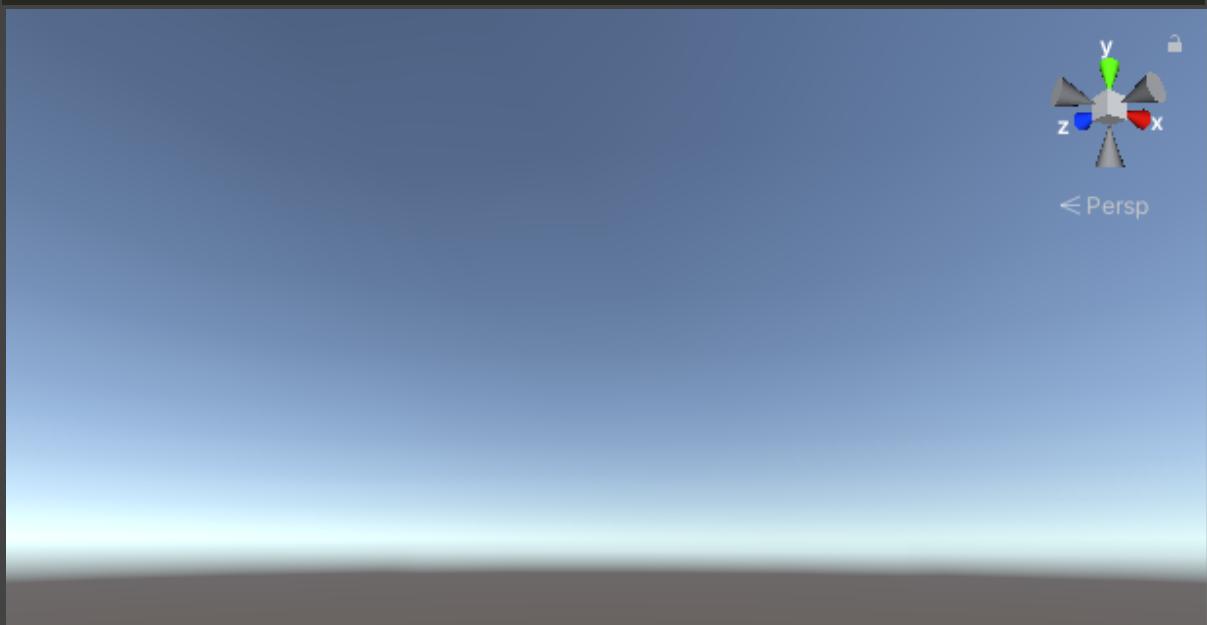


要注意的是現在的天空盒並不會受到攝影機的視角改變而有變化，這是因為我們只是呼叫了DrawSkybox，這個方法只是確定要不要畫天空盒，我們需要攝影機的清除標誌。
要正確的渲染天空盒並覆蓋整個場景，我們需要設定view-projection matrix。這個轉換矩陣結合了攝影機的位置和視角也就是view matrix，還有透視攝影機或正交投影攝影機也就是project matrix。在shader中會以unity_MatrixVP表示。
現在unity_MatrixVP永遠都是一樣的，我們需要透過SetupCameraProperties方法將攝影機的屬性餵給context。我們在DrawVisibleGeometry前加上Setup方法。

```
public void Render(ScriptableRenderContext _context, Camera _camera)
{
    context = _context;
    camera = _camera;

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

private void Setup()
{
    context.SetupCameraProperties(camera);
}
```



2.3 Command Buffers

context會延遲當前要渲染的狀態，直到我們呼叫submit才更新。一些任何像是畫天空盒，你可以直接呼叫已經定義好的DrawSkybox方法，但是其它的命令沒有辦法直接的操作，需要透過Command buffer來分開進行。

為了取得Buffer我們需要創建一個新的CommandBuffer物件，因為我們目前只需要一個所以僅建立一個並賦予他名字Render Camera。

```
const string bufferName = "Render Camera";
private CommandBuffer commandBuffer = new CommandBuffer { name =
bufferName };
```

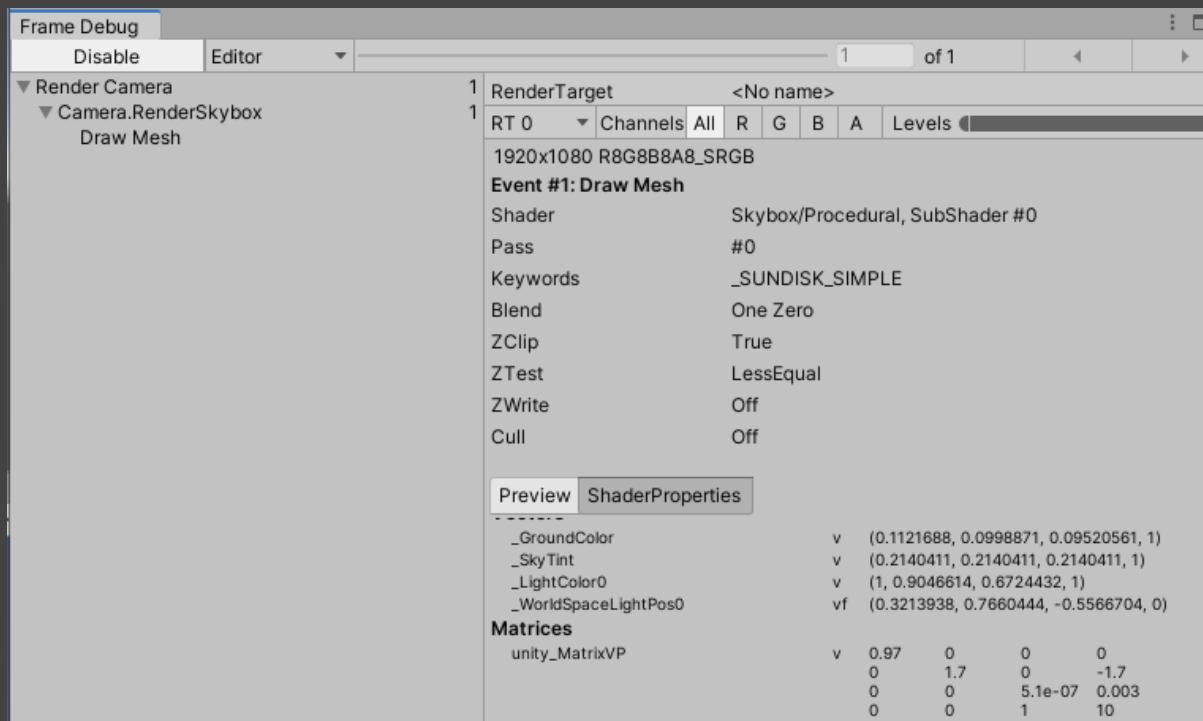
我們可以使用command buffer來導入profiler sample，他可以同時顯示在profiler和frame debug視窗中。可以在合適的地方透過BeginSample和EndSample來規劃出要Sample的區段，這個例子中以setup和submit分別來調用Begin和End方法。

為了執行這個command buffer我們需要在context帶入buffer呼叫ExecuteCommandBuffer。他會複製buffer內容，但不會清掉buffer內容，讓我們在後續仍可重複使用。

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

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

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



現在Camera.RenderSkyBox的sample會包含在Render Camera下方。

2.4 清除Render Target

無論我們繪製什麼內容，最後都會渲染在攝影機的Render Target上，預設是在frame buffer，也有可能會在Render Texture中。不管畫了什麼在這個target上他都會一直保存，這可能會影響到我們當下要渲染的畫面。為了保證渲染出來的畫面，我們需要執行clear來把舊的內容從target中清除。這個動作透過command buffer呼叫ClearRenderTarget方法來完成。

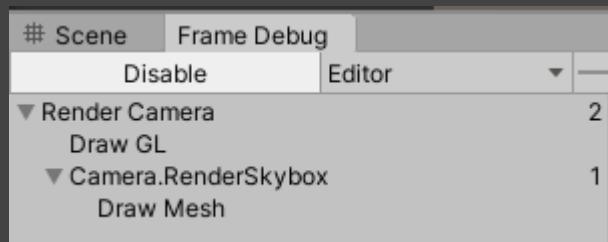
CommandBuffer.ClearRenderTarget需要至少3個參數，前2個分別是深度和顏色資料是否要清除，第3個是用來決定清掉後背景的顏色。

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



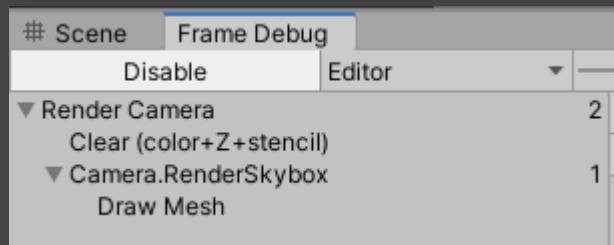
Frame Debug現在顯示Draw GL來執行clear動作，但他顯示在額外的一個Render Camera底下，這是因為ClearRenderTarget 使用CommandBuffer的名稱將清除包裝在一個Sample中，這時候我們需要調換一下程式執行的順序移除這個重複的狀態，將clear移到begin sample的操作之前，這會讓2個相鄰的camera sample 範圍merge在一起。

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



Draw GL表示使用Hidden/InternalClear shader畫了一個全屏的Quad到Render Target中，這在清除Render Target時不是一個很有效率的作法。這是因為我們在設定Camera屬性前執行了清除，如果我們調換了2個流程的順序，我們可以獲得較快的清除方式。

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



現在我們可以看到Clear (color+Z+stencil), 這裡指出顏色和深度都被清除。Z表示深度buffer(depth buffer), 而stencil是同一個buffer的一部份。

2.5 Culling

現在我們可以看到天空盒了，但卻無法在場景中看見我們早早就擺好的物件。我們需要繪製出攝影機能看到的物件。我們從所有具有Renderer Component的物件開始，並且要cull掉那些在攝影機視錐體外的物件。

要知道哪些物件可以被cull，我們需要追蹤所有的攝影機設定和矩陣，因此我們可以使用 `ScriptableCullingParameters` 結構。我們可以在攝影機上呼叫 `TryGetCullingParameters` 來取代自己填入相關參數。它回傳成的是 `cull` 參數是否有效，若能成功檢索就回傳 `true`，同時根據傳入的 `cullingParameters` 保存所成功獲取的攝影機剔除參數。若渲染的攝影機無效(視錐體為空、裁剪面設置無效等)，就返回 `false`。

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

    return false;
}
```

為何要寫 `out` 呢？

我們需要在 `Render` 的最前方執行 `Cull`，如果 `Cull` 失敗，就直接跳出 `Render`。

```
public void Render(ScriptableRenderContext _context, Camera _camera)
{
    context = _context;
    camera = _camera;

    if(!Cull())
    {
        return;
    }

    Setup();
    DrawVisibleGeometry();
    Submit();
}
```

真正的 `Cull` 是使用 `context` 去調用的，於是產生了 `CullingResults` 結構。如果 `Cull` 成功，會在 `Cull` 中執行此操作並將結果存儲在了 `CullingResults` 中。在這種情況下需要在前方添加 `ref` 關鍵字，將 `CullingParameters` 作為參數傳遞

```
private CullingResults cullingResults;
```

```
private bool Cull()
{
    if(camera.TryGetCullingParameters(out ScriptableCullingParameters
scriptableCullingParameters))
    {
        cullingResults = context.Cull(ref scriptableCullingParameters);
        return true;
    }

    return false;
}
```

2.6 Drawing Geometry

當我們知道有哪些是能被看見的，我們就可以開始執行渲染這些物件。我們將CullingResult當作參數交給context的DrawRenderers方法，同時要提供繪製設定(Drawing Setting)和過濾設定(Filter Setting) - **DrawingSettings** 和 **FilteringSettings**。我們先在一開始使用預設建構子，2個都作為ref參數來傳遞，並在DrawVisibleGeometry的繪制天空盒之前執行。

```
private void DrawVisibleGeometry()
{
    DrawingSettings drawingSettings = new DrawingSettings();
    FilteringSettings filteringSettings = new FilteringSettings();
    context.DrawRenderers(cullingResults, ref drawingSettings, ref
filteringSettings);

    context.DrawSkybox(camera);
}
```

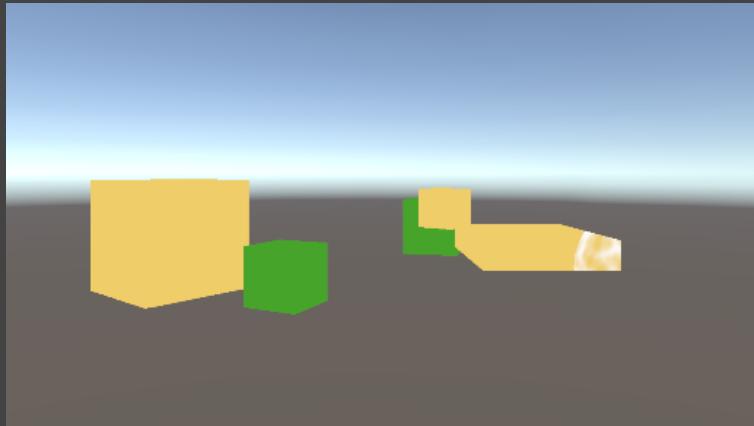
我們現在依然看不到任何物件，這是因為我們需要告訴他哪些shader pass是被允許通過的。若我們只支援unlit shader，我們就需要先設定存取的tag id 紿SRPDefaultUnlit pass，我們將這個id存在static 欄位中。

我們將這個SRPDefaultUnlit 作為參數設定給DrawingSettings，同時我們要建立SortingSettings結構資料。我們將攝影機傳遞給SortingSettings，這個用來決定正交投影或是距離為基礎的排序結構。
接著我們需要指出哪個RenderQueue可以通過。傳遞RenderQueueRange.all到FilteringSettings，讓他可以包含所有的物件。

```
private static ShaderTagId unlitShaderTag = new
ShaderTagId("SRPDefaultUnlit");

private void DrawVisibleGeometry()
{
    SortingSettings sortingSettings = new SortingSettings(camera);
    DrawingSettings drawingSettings = new
DrawingSettings(unlitShaderTag, sortingSettings);
    FilteringSettings filteringSettings = new
FilteringSettings(RenderQueueRange.all);
    context.DrawRenderers(cullingResults, ref drawingSettings, ref
filteringSettings);

    context.DrawSkybox(camera);
}
```



現在我們可以在Game view和Scene view中看到Unlit物件。

從FrameDebug中可以看到只有使用Unlit shader的物件會被繪製。所有的Draw calls都在RenderLoop.Draw中按順序排列。但你可以看到透明物件的表現有點奇怪。

The Frame Debug window displays the rendering process. The list of draw calls is as follows:

- 9 Render Camera
Clear (color+Z+stencil)
- 7 RenderLoop.Draw
Draw Mesh Unlit Orange 3
- Draw Mesh Unlit Transparent White 1
- Draw Mesh Unlit Green 1
- Draw Mesh Unlit Orange 2
- Draw Mesh Unlit Green 2
- Draw Mesh Unlit Orange 1
- Draw Mesh Unlit Transparent White 2
- 1 Camera.RenderSkybox
Draw Mesh

A note in the list states: "Why this draw call can't be batched with the previous one Objects have different materials."

Properties for the last draw call (Draw Mesh Unlit Transparent White 2) are shown in the right panel:

- RenderTarget: <No name>
- RT 0: 1920x1080 R8G8B8A8_SRGB
- Event #8: Draw Mesh
- Shader: Unlit/Transparent, SubShader #0
- Pass: #0
- Blend: SrcAlpha OneMinusSrcAlpha
- ZClip: True
- ZTest: LessEqual
- ZWrite: Off
- Cull: Back

Textures:
_MainTex: sphere-alpha-map

Vectors:
_MainTex_ST: v (1, 1, 0, 0)

Matrices:
unity_MatrixVP: v 0.97 0 0 0
0 1.7 0 -1.7
0 0 -0.0003 0.3
0 0 1 10

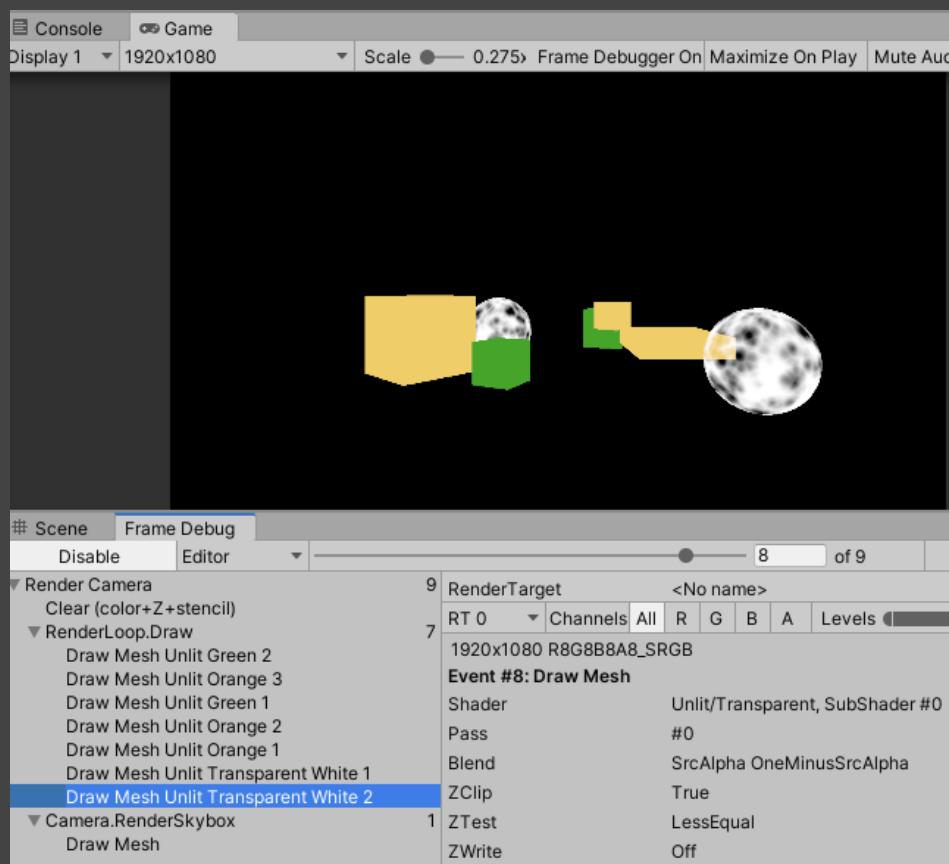
Drawing Order似乎有些混亂，我們可以透過criteria屬性強制設定一個特殊的order給SortingSettings，這邊使用`SortingCriteria.CommonOpaque`，這是個典型的不透明物件排序方法。

```

private void DrawVisibleGeometry()
{
    SortingSettings sortingSettings =
        new SortingSettings(camera)
    {
        criteria = SortingCriteria.CommonOpaque
    };
    DrawingSettings drawingSettings = new
DrawingSettings(unlitShaderTag, sortingSettings);
    FilteringSettings filteringSettings = new
FilteringSettings(RenderQueueRange.all);
    context.DrawRenderers(cullingResults, ref drawingSettings, ref
filteringSettings);

    context.DrawSkybox(camera);
}

```



物件現在或多或少是由前往後繪製，這是理想的不透明物件繪製的順序。如果某些物件的部分最終被繪製在其他物件的部分後面，則可以跳過其隱藏的fragment，從而加快渲染速度。常見的不透明排序選項還考慮了其他一些標準，包括Render Queue和materials。

2.7 分別繪製不透明和透明幾何物件

frame debug中顯示透明物件被繪製出來了，但在後面被天空盒的渲染把所有的都給蓋過去，因為天空盒會在不透明物件之後繪製，所以所有被遮蓋住的fragment會被忽略，但他會覆寫透明幾何物件。這是因為透明物件不寫入depth buffer。透明物件並沒有蓋住任何在他後方的物件，我們可以透過它們看到後面。解決辦法是，第一步先畫不透明物件，接著畫天空盒，最後繪製透明物件。

我們可以從DrawRenderers的一開始將RenderQueueRange改為Opaque來移除透明物件。並在天空盒的後面再呼叫一次DrawRenderers，並將RenderQueueRange改為Transparent，同時改變sorting criteria為CommonTransparent，接著將SortingSettings重新設定給DrawingSettings，這個會反轉透明物件的排序（由後往前）。

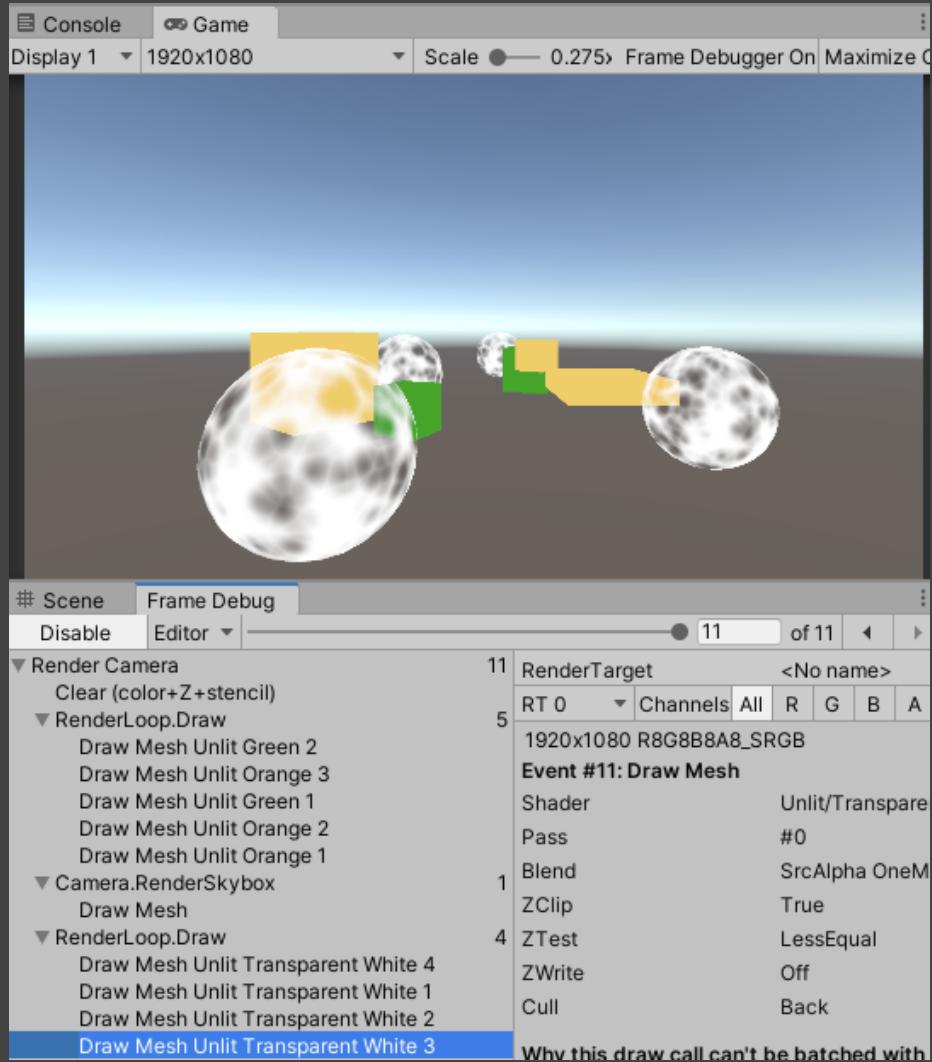
```
private void DrawVisibleGeometry()
{
    //不透明物件
    var filteringSettings = new
    FilteringSettings(RenderQueueRange.opaque);

    SortingSettings sortingSettings =
        new SortingSettings(camera)
    {
        criteria = SortingCriteria.CommonOpaque
    };
    DrawingSettings drawingSettings = new
    DrawingSettings(unlitShaderTag, sortingSettings);
    context.DrawRenderers(cullingResults, ref drawingSettings, ref
filteringSettings);

    // 天空盒
    context.DrawSkybox(camera);

    // 透明物件
    sortingSettings.criteria = SortingCriteria.CommonTransparent;
    drawingSettings.sortingSettings = sortingSettings;
    filteringSettings.renderQueueRange = RenderQueueRange.transparent;

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



什麼是draw order reversed?

由於透明物件不會寫入Depth buffer, 因此由前往後排序他們並沒有效能上的收益。當透明物件彼此錯落後, 我們需要由後往前排序讓彼此能正確的Blending。

不幸的是, 由後往前的排序並不能保證正確的Blending效果, 因為排序是以物件為中心, 而且只依照物件的位置來排序。當物件交錯且較大的透明物件時, 仍會有不正確的結果。有時候我們可以透過裁切幾何的一小部分來解決。

3 Editor Rendering

現在我們的RP已經可以正確的繪製Unlit物件，但還有少部分的事情我們需要改進。

3.1 Drawing Legacy Shaders

因為當前的pipeline只支援unlit shader，物件若是使用不一樣的pass就不會被渲染，導致看不見他們。這是正確的，這是因為場景中物件使用了錯的shader導致他被隱藏起來。所以我們要分開渲染他們。

為了涵蓋所有Unity預設shader我們需要使用ShaderTagID給Always, ForwardBase, PrepassBase, Vertex, VertexLMRGBM, 和 VertexLM 等pass，並將他們放在static array。

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

繪製所有不支援的shader在DrawVisibleGeometry之後的另一個方法，因為這是無效的pass所以結果必定是錯的，所以我們不在乎其它的設定，將filteringSettings設定為default value。

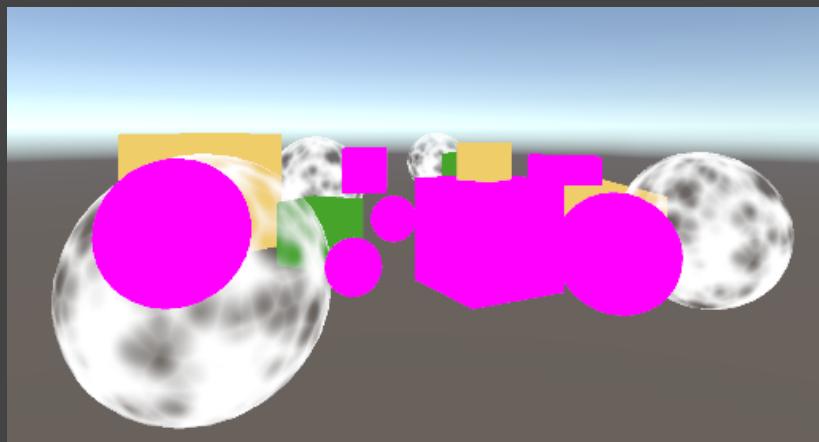
3.2 Error Material

為了能清楚的指出哪些物件使用的是無效/不支援的shader，我們使用unity的error shader。建構一個使用內建error shader的新材質球，我們可以透過Shader.Find來找到Hidden/InternalErrorShader。我們用static欄位來紀錄這個材質球，並將它塞給DrawSettings的overrideMaterial屬性。

```
private static Material errorMaterial;
private void DrawUnsupportedShaders()
{
    if(errorMaterial == null)
    {
        errorMaterial = new
Material(Shader.Find("Hidden/InternalErrorShader"));
    }

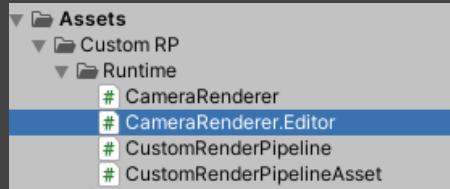
    var drawingSettings = new DrawingSettings(
        legacyShaderTagIdArray[0], new SortingSettings(camera)
    )
    {
        overrideMaterial = errorMaterial
    };

    for (int i = 1; i < legacyShaderTagIdArray.Length; i++)
    {
        drawingSettings.SetShaderPassName(i, legacyShaderTagIdArray[i]);
    }
    var filteringSettings = FilteringSettings.defaultValue;
    context.DrawRenderers(
        cullingResults, ref drawingSettings, ref filteringSettings
    );
}
```



3.3 Partial Class

繪製無效物件在開發時很有用，但在build出來的app上不是必要的。所以我們要將editor-only code放到額外的partial class檔案中。我們需要分裂原始的*CameraRenderer*腳本，並重新命名他為*CameraRenderer.Editor*。



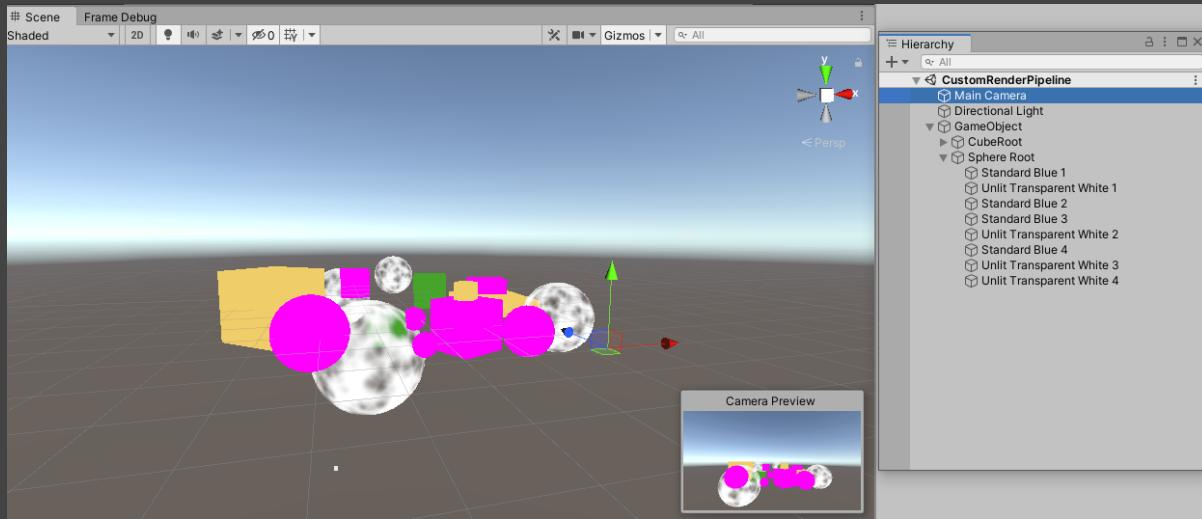
將原本的*CameraRenderer* class轉為partial class 並將tag array、error material和*DrawUnsupportedShaders()*移除掉。

由於*DrawUnsupportedShaders*只在UNITY_EDITOR存在，若是編出來的版本是沒有這個方法的，於是需要以partial的方式創造同名的空方法，讓版本能順利的Build完成。

```
partial void DrawUnsupportedShaders();  
  
#if UNITY_EDITOR  
partial void DrawUnsupportedShaders()  
{  
    ....  
}  
#endif
```

3.4 Drawing Gizmos

目前我們的RP沒有畫出gizmos, 不管是scene view還是game view。



我們可以透過調用UnityEditor.Handles.ShouldRenderGizmos來檢查否需要繪製gizmos。若是需要我們就要讓context攜帶camera參數來呼叫DrawGizmos, 接著第二個參數要指定使用來繪制的gizmos subset。有2個subsets給before和after image effect。尤於當前我們不支援image effect, 所以我們2個都調用。我們將這個操作寫在新的editor腳本並命名為DrawGizmos。

```
using UnityEditor;

public partial class CameraRenderer
{
    partial void DrawGizmos();

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

    partial void DrawUnsupportedShaders()
    {
        ...
    }
#endif
}
```

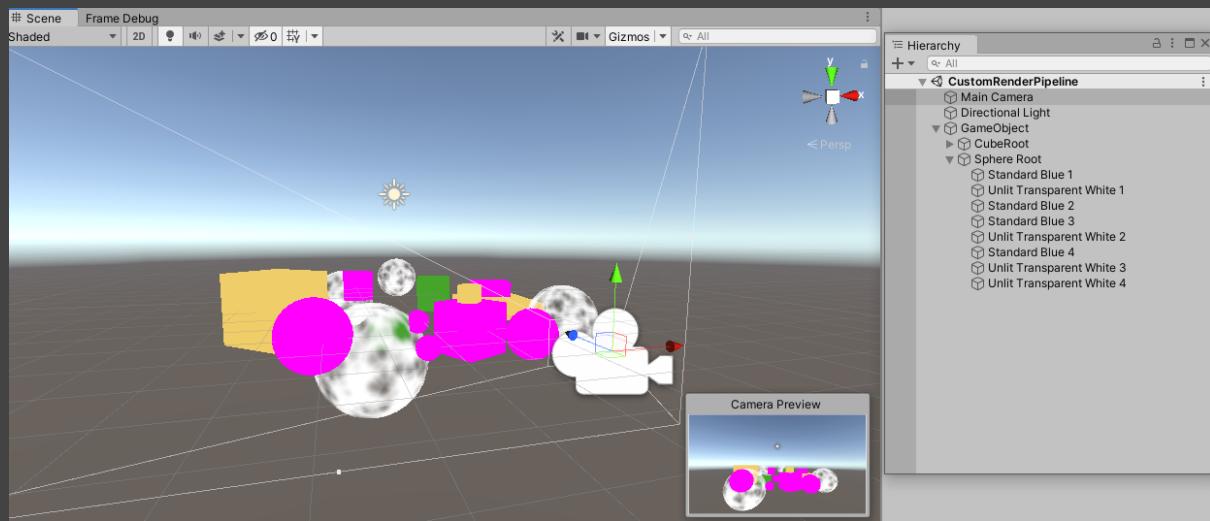
而Gizmos要在所有東西的渲染之後才畫，所以我們要回到RenderCamera補上這段code。

```
public void Render(ScriptableRenderContext _context, Camera _camera)
{
    context = _context;
    camera = _camera;

    if(!Cull())
    {
        return;
    }

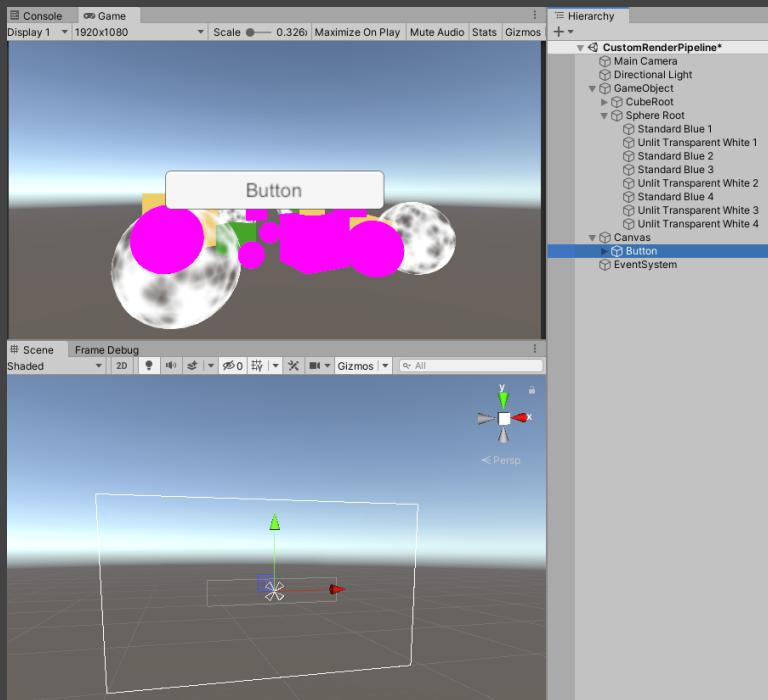
    Setup();
    DrawVisibleGeometry();
    DrawUnsupportedShaders();
    DrawGizmos();
    Submit();
}
```

現在我們已經可以看到Gizmos啦！

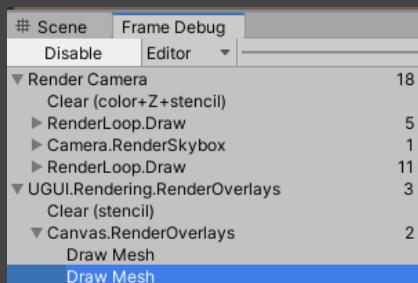


3.5 Drawing Unity UI

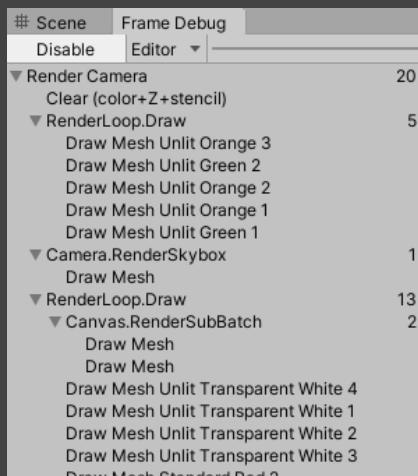
另一件我們需要關注的問題是遊戲中的使用者界面。例如，透過添加一個按鈕來建立一個簡單的UI，這個按鈕會顯示在Game view的最上方，但不會出現在scene view。



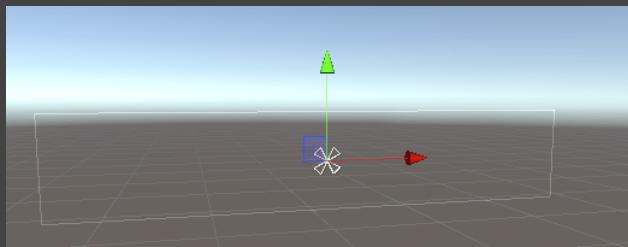
而在frame debug 中你會看到UI的繪製是在分開在另一個地方，不是在我們的RP下。



接著我們將Canvas的Render Mode從Screen Space - Overlay轉為Screen Space - Camera並使用Main Camera作為Render Camera, 這時候就會把UI的部分轉到透明物件的分類中。



UI在scene view會一直使用World space模式來渲染。但ui還是沒有在scene view 中被畫下來。UI依然是不可見的。



我們需要將UI繪製在scene view, 透過調用 ScriptableRenderContext.EmitWorldGeometryForSceneView, 我們在新的editor腳本中新增一個名為PrepareForSceneWindow的方法，並在camera type是scene camera時才有作用。

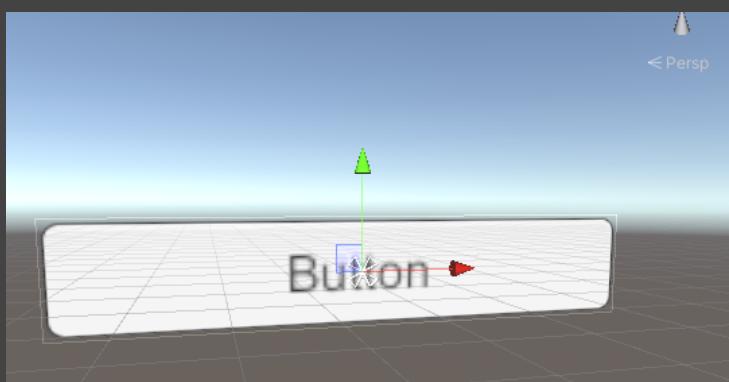
```
partial void PrepareForSceneWindow();

#if UNITY_EDITOR
    partial void PrepareForSceneWindow()
    {
        if(camera.cameraType == CameraType.SceneView)
        {

            ScriptableRenderContext.EmitWorldGeometryForSceneView(camera);
        }
    }
    ...
#endif
```

同時，這個方法需要在繪製幾何時，並在culling之前調用。

```
PrepareForSceneWindow();
if (!Cull())
{
    return;
}
```



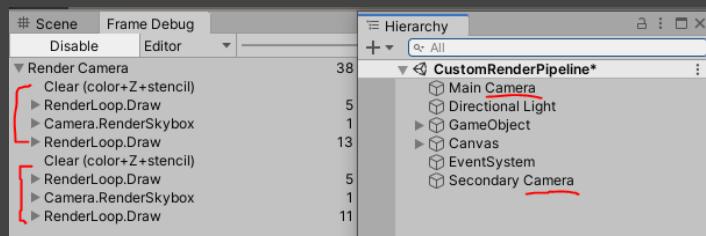
現在我們可以在scene view看到這個按鈕啦。

4 Multiple Cameras

真實的情形下你可能同時有多支正在作用的camera, 所以我們需要確保他們能夠同時運作。

4.1 Two Cameras

每支camera都有預設的深度值，其值為-1，繪制是以遞增的深度來排序。我們新加一個攝影機，並命名為Secondary Camera, 並將他的深度值改為0。



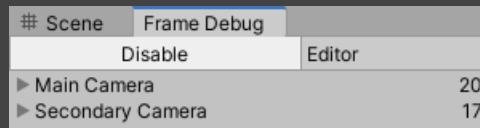
從frame debug中可以看到我們的場景被渲染了2次，但實際上我們看到的只有最終攝影機畫出來的畫面，也就是深度值比較大的攝影機看到的畫面。而2支攝影機目前使用相同的sample名稱- Render Camera, 這容易搞混到底哪個攝影機做了哪幾個步驟。

因為每支camera有自己的執行範圍，所以我們需要額外的scope，這個功能僅在editor生效，於是我們需要新增一個方法名為PrepareBuffer, 並讓buffer與camera名稱一致。

```
partial void PrepareBuffer();  
  
#if UNITY_EDITOR  
    partial void PrepareBuffer()  
    {  
        commandBuffer.name = camera.name;  
    }  
    ...  
#endif
```

並在CameraRender腳本中，將這個方法安插在PrepareForSceneWindow之前。

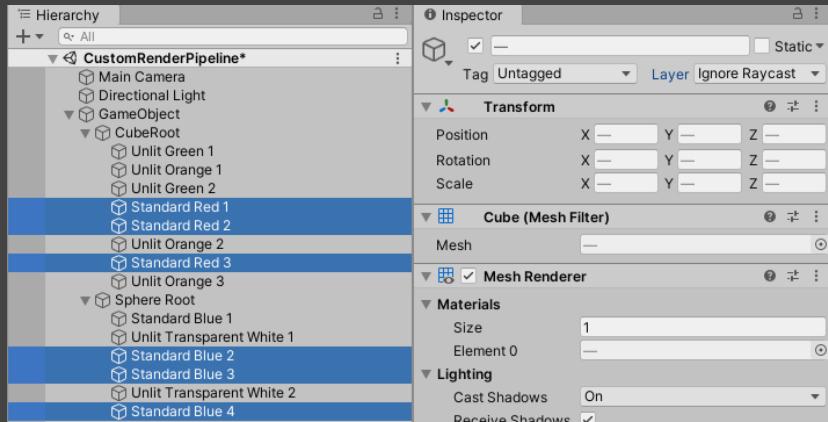
```
PrepareBuffer();  
PrepareForSceneWindow();
```



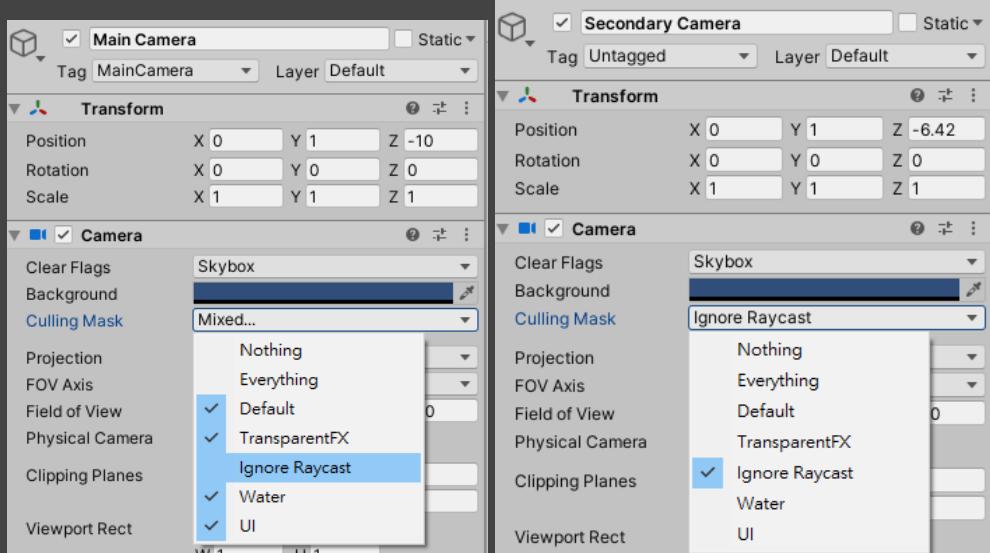
現在你可以看到2支攝影機分別繪製的是哪些物件，在這邊我2個攝影機放在不同位置，所以繪製的draw call會不太一樣。

4.2 Layers

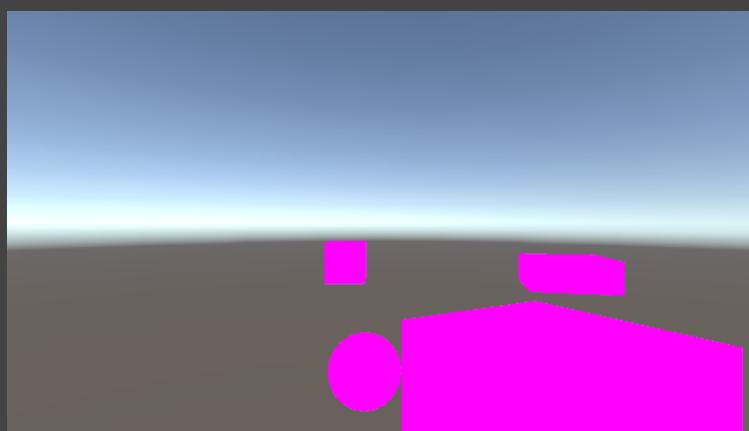
Camera可以透過設定Culling Mask來決定他能看到哪些Layer內的物件。為了測試，我們將所有Standard Shader物件的Layer設定為Ignore Raycast Layer。



並將這個Layer從Main Camera中移除，並只在Secondary Camera中設定Ignore Raycast layer。



由於Secondary Camera是最後渲染的攝影機(depth 0 [secondary] > depth -1 [main]), 所以我們會看到下方不合法的物件，因為當前我們還只會處理Unlit shader。



4.3 Clear Flags

我們可以透過調整Secondary Camera的clear flags屬性，讓2支攝影機能結合在一塊。
在Setup中從camera身上取得CameraClearFlags。

```
private void Setup()
{
    context.SetupCameraProperties(camera);
    CameraClearFlags flags = camera.clearFlags;
    commandBuffer.ClearRenderTarget(true, true, Color.clear);
    commandBuffer.BeginSample(SampleName);
    ExecuteBuffer();
}
```

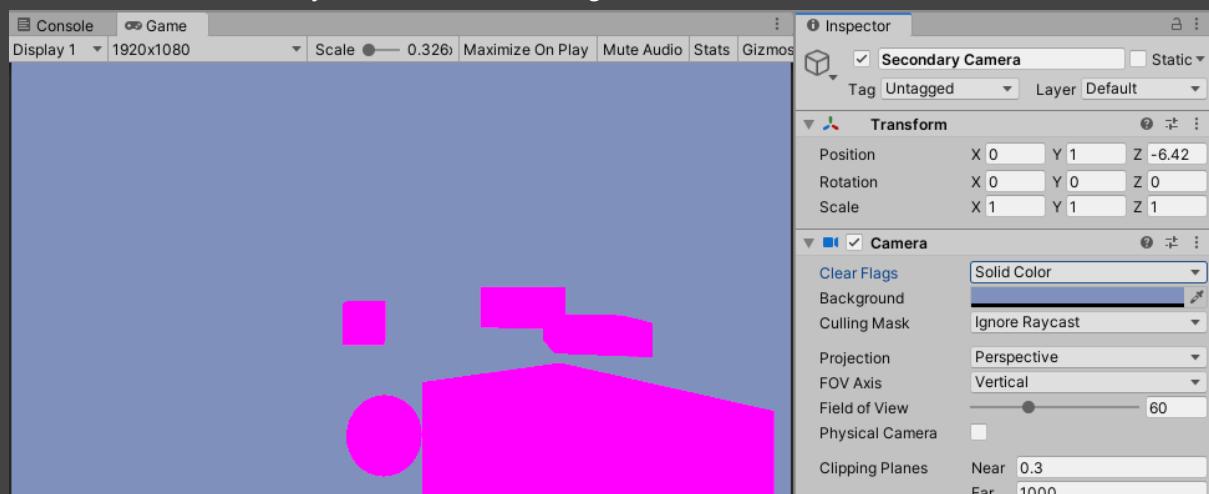
CameraClearFlags定義了4個數字，從1到4分別表示為，Skybox、Color、Depth和Nothing。他們表示清除的減少，除了最後一個Nothing外，其它都必須清除Depth buffer。

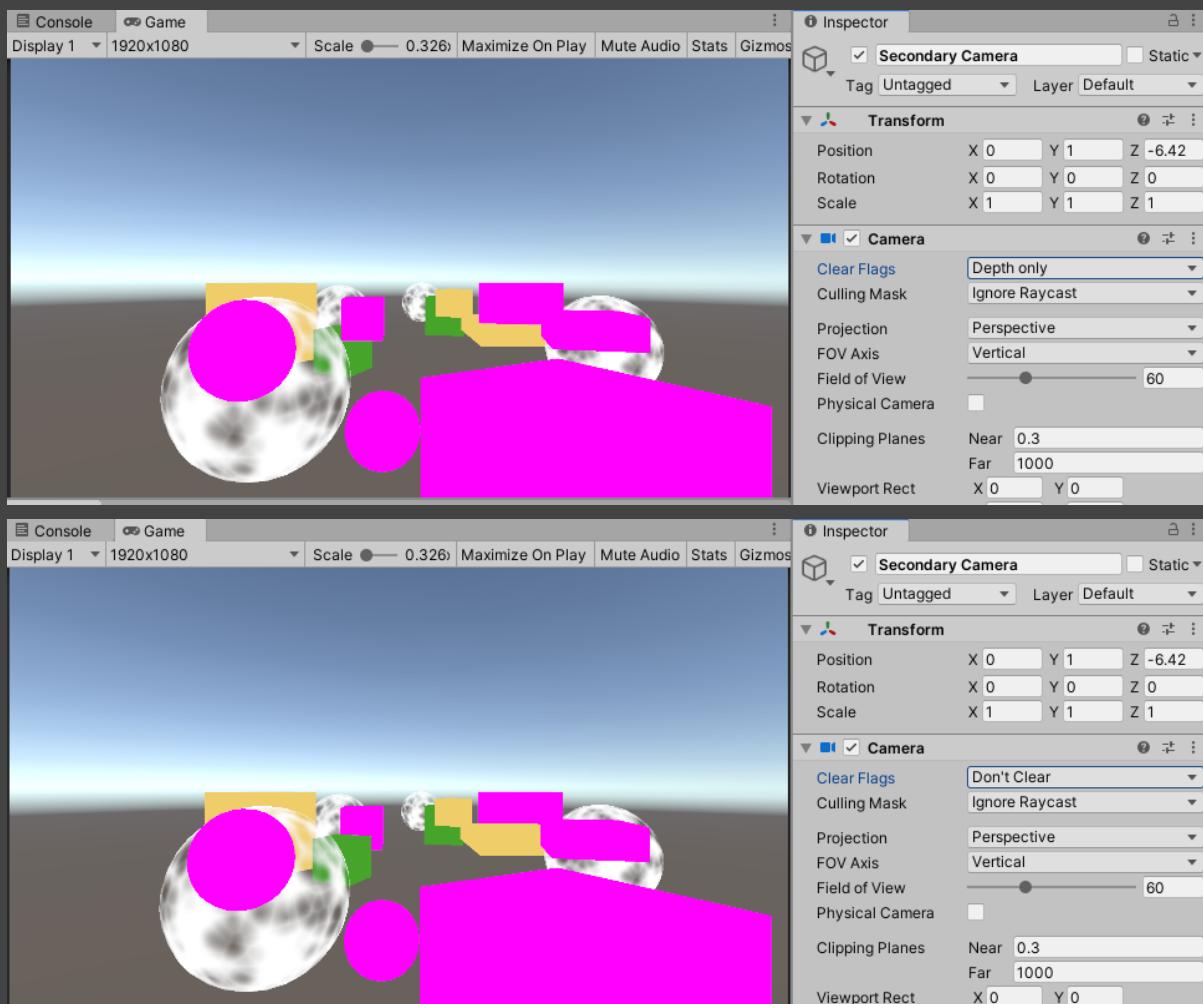
而顏色的清除只有在ClearFlags被設定為Color才執行。

若我們設定成clear 為 solid color，我們就需要使用攝影機的背景色，因為我們渲染在linear color space，我們需要將顏色轉到線性色彩空間，所以我們需要使用到 camera.backgroundColor.linear，若是其它情況的話直接填上Color.clear就好。

```
private void Setup()
{
    context.SetupCameraProperties(camera);
    CameraClearFlags flags = camera.clearFlags;
    commandBuffer.ClearRenderTarget(
        flags <= CameraClearFlags.Depth,
        flags == CameraClearFlags.Color,
        flags == CameraClearFlags.Color ?
            camera.backgroundColor.Linear : Color.clear);
    commandBuffer.BeginSample(SampleName);
    ExecuteBuffer();
}
```

我們分別調整Secondary Camera的Clear Flags來看看顯示的效果。





我們將Secondary Camera的Viewport Rect 縮小並調整到偏中間的位置，可以看到RenderTarget剩下的區域是不會受到Secondary Camera影響的。

