**聚光灯阴影**

本文翻译自Catlike Coding，原作者：Jasper Flick。

本文经原作者授权，转载请说明出处。

原文链接在下：

https://catlikecoding.com/unity/tutorials/scriptable-render-pipeline/spotlight-shadows/

**本章内容如下：**

* 同一张纹理的渲染和读取
* 摄像机放到光源的位置渲染
* 为阴影映射添加一个pass
* 阴影映射纹理的采样
* 同时支持hard阴影和soft阴影
* 一个图集中最多支持16张阴影映射纹理

这是Unity可编程渲染管线系列教程的第四章。在本章中，我们将添加最多16个支持阴影的聚光灯。

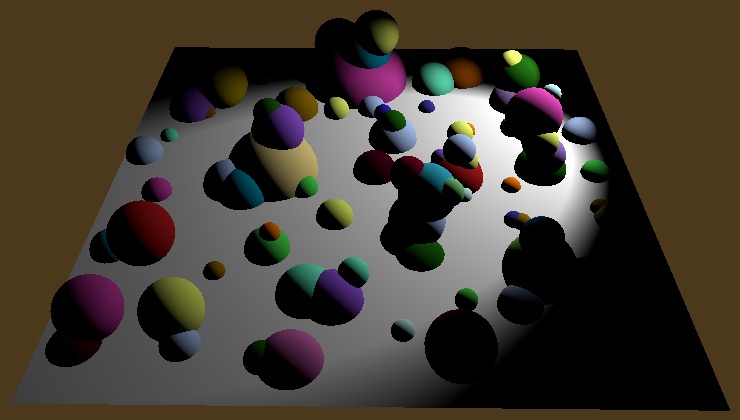
本教程使用Unity 2018.3.0f2完成。

**一、带有阴影的聚光灯**

阴影非常重要，既可以增加真实感，又可以使物体之间的空间关系更加明显。如果没有阴影，就很难判断某些物体是浮在一个表面上还是紧贴在一个表面上。

Rendering 7, Shadows教程介绍了unity默认的渲染管线的阴影是如何实现的，但是不适用于我们的单pass向前渲染路径。尽管如此，阴影映射纹理技术仍然很有用。在本教程中，我们只实现聚光灯的阴影，因为它们是最简单。

我们从一个阴影光开始，首先创建一个包含几个物体和一个聚光灯的场景，并使用一个平来接收阴影。所有物品都应使用我们的Lit Opaque材质球。



**1.1阴影映射纹理**

有几种不同的方法来处理阴影，但我们将使用阴影映射纹理技术。这意味着我们将从灯光的角度渲染场景。我们只对这次渲染的深度信息感兴趣，因为它能告诉我们光线到达表面的距离，超过这个距离的地方就在阴影中。

在使用相机渲染阴影映射纹理之前，我们首先要创建它。为了能够稍后对阴影映射纹理进行采样，我们必须渲染为单独的渲染纹理而不是通常的帧缓冲区。在MyPipeline添加一个RenderTexture字段。

RenderTexture shadowMap;

创建一个单独的函数来渲染阴影，并将渲染上下文作为参数。它要做的第一件事就是通过调用静态方法RenderTexture.GetTemporary，来创建一个阴影映射纹理的对象。该方法要么创建一个新的渲染纹理，要么重新使用一个尚未清理的旧纹理。因为我们每帧都需要阴影映射纹理，所以它将一直被重用。

RenderTexture.GetTemporary需要纹理的尺寸、深度通道的位数以及纹理格式作为参数。我们使用512×512的固定大小，16位的深度通道，以及RenderTextureFormat.Shadowmap 的纹理格式。

void RenderShadows (ScriptableRenderContext context) {

shadowMap = RenderTexture.GetTemporary(

512, 512, 16, RenderTextureFormat.Shadowmap

);

}

设置纹理的filterMode和wrapMode。

shadowMap = RenderTexture.GetTemporary(

512, 512, 16, RenderTextureFormat.Shadowmap

);

shadowMap.filterMode = FilterMode.Bilinear;

shadowMap.wrapMode = TextureWrapMode.Clamp;

阴影映射纹理需要在常规渲染之前渲染，因此在剔除之后，设置常规摄像机之前调用RenderShadows。

void Render (ScriptableRenderContext context, Camera camera) {

…

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

RenderShadows(context);

context.SetupCameraProperties(camera);

…

}

另外，在我们提交完渲染上下文后，需要释放渲染纹理。把阴影映射纹理传给RenderTexture.ReleaseTemporary方法，并清除我们的字段。

void Render (ScriptableRenderContext context, Camera camera) {

…

context.Submit();

if (shadowMap) {

RenderTexture.ReleaseTemporary(shadowMap);

shadowMap = null;

}

}

**1.2阴影命令缓冲区**

我们为所有阴影工作使用单独的命令缓冲区，这样我们就可以在帧调试器中区分阴影和常规渲染。

CommandBuffer cameraBuffer = new CommandBuffer {

name = "Render Camera"

};

CommandBuffer shadowBuffer = new CommandBuffer {

name = "Render Shadows"

};

阴影渲染的相关代码要放在BeginSample和EndSample命令之间，就像常规渲染一样。

void RenderShadows (ScriptableRenderContext context) {

shadowMap = RenderTexture.GetTemporary(

512, 512, 16, RenderTextureFormat.Shadowmap

);

shadowMap.filterMode = FilterMode.Bilinear;

shadowMap.wrapMode = TextureWrapMode.Clamp;

shadowBuffer.BeginSample("Render Shadows");

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

shadowBuffer.EndSample("Render Shadows");

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

}

**1.3设置渲染目标**

在我们渲染阴影之前，我们需要调用CoreUtils.SetRenderTarget方法，来告诉GPU把阴影映射纹理渲染到shadowMap，该方法需要阴影命令缓冲区和shadowMap作为参数。我们需要在BeginSample之前调用CoreUtils.SetRenderTarget方法，这样，在帧调试器中就不会出现额外嵌套的Render Shadows。

CoreUtils.SetRenderTarget(shadowBuffer, shadowMap);

shadowBuffer.BeginSample("Render Shadows");

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

由于我们只关心深度通道，因此只需要清除该通道。为SetRenderTarget方法添加第三个参数ClearFlag.Depth来清除深度通道。

CoreUtils.SetRenderTarget(

shadowBuffer, shadowMap,

ClearFlag.Depth

);

虽然不是必须的，但我们最好为阴影映射纹理设置加载和存储方式。我们都不关心纹理是怎么获取的，所以设置加载方式为：RenderBufferLoadAction.DontCare，这使得tile-based GPU更高效。我们需要稍后从纹理中进行采样，因此需要将其保存在内存中，所以我们设置存储方式为：RenderBufferStoreAction.Store。添加这些作为第三和第四个参数传递给CoreUtils.SetRenderTarget方法。

CoreUtils.SetRenderTarget(

shadowBuffer, shadowMap,

RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store,

ClearFlag.Depth

);

在帧缓冲区中，阴影映射纹理的清除操作出现在了常规摄像机渲染之前。



**1.4设置观察和投影矩阵**

我们想从光源的角度进行渲染，这意味着我们将聚光灯视为相机，因此，我们必须提供观察和投影矩阵。通过调用剔除结果的ComputeSpotShadowMatricesAndCullingPrimitives方法来获取观察和投影矩阵，它需要光源的index作为参数。由于场景中只有一个聚光灯，index参数是0，第二和第三个参数输出观察和投影矩阵。除此之外，还有一个输出参数：ShadowSplitData，我们目前不用它。

shadowBuffer.BeginSample("Render Shadows");

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

Matrix4x4 viewMatrix, projectionMatrix;

ShadowSplitData splitData;

cull.ComputeSpotShadowMatricesAndCullingPrimitives(

0, out viewMatrix, out projectionMatrix, out splitData

);

获取矩阵之后，通过调用命令缓冲区的SetViewProjectionMatricesshadow方法来设置矩阵。

cull.ComputeSpotShadowMatricesAndCullingPrimitives(

0, out viewMatrix, out projectionMatrix, out splitData

);

shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

**1.5渲染阴影映射物体（Shadow Casters）**

设置好观察和投影矩阵后，我们通过调用渲染上下文的DrawShadows方法来渲染所有阴影映射物体。该方法有一个DrawShadowsSettings类型的引用参数，我们可以通过构造函数来创建它，该方法将剔除结果和光源索引作为参数。

shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

var shadowSettings = new DrawShadowsSettings(cull, 0);

context.DrawShadows(ref shadowSettings);

这仅适用于我们的聚光灯的阴影类型设置为hard或soft。如果设置为none，那么Unity会认为它不是一个有效的阴影映射光源。

https://catlikecoding.com/unity/tutorials/scriptable-render-pipeline/spotlight-shadows/a-spotlight-with-shadows/hard-shadows.png

**二、阴影映射Pass（Shadow Caster Pass）**

目前为止，所有受光照影响的物体都应渲染到阴影映射纹理，但帧调试器告诉我们这并没有发生。那是因为DrawShadows方法需要使用特定的阴影映射Pass，我们的着色器中还没有这样的Pass。

**2.1阴影包含文件**

要创建阴影映射Pass，复制Lit.hlsl文件并将其命名为ShadowCaster.hlsl。我们只关心深度信息，因此从新文件中删除与片元位置无关的代码。片元着色器的输出是0。还要重命名pass函数和包含防护宏。

#ifndef MYRP\_SHADOWCASTER\_INCLUDED

#define MYRP\_SHADOWCASTER\_INCLUDED

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/Common.hlsl"

CBUFFER\_START(UnityPerFrame)

float4x4 unity\_MatrixVP;

CBUFFER\_END

CBUFFER\_START(UnityPerDraw)

float4x4 unity\_ObjectToWorld;

CBUFFER\_END

#define UNITY\_MATRIX\_M unity\_ObjectToWorld

#include "Packages/com.unity.render-pipelines.core/ShaderLibrary/UnityInstancing.hlsl"

struct VertexInput {

float4 pos : POSITION;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

struct VertexOutput {

float4 clipPos : SV\_POSITION;

};

VertexOutput ShadowCasterPassVertex (VertexInput input) {

VertexOutput output;

UNITY\_SETUP\_INSTANCE\_ID(input);

float4 worldPos = mul(UNITY\_MATRIX\_M, float4(input.pos.xyz, 1.0));

output.clipPos = mul(unity\_MatrixVP, worldPos);

return output;

}

float4 ShadowCasterPassFragment (VertexOutput input) : SV\_TARGET {

return 0;

}

#endif // MYRP\_SHADOWCASTER\_INCLUDED

这足以渲染阴影，但是如果投影物体和近裁剪平面相交，这可能会导致阴影中出现孔洞。为了防止这种情况，我们必须在顶点着色器中通过获取裁剪空间顶点的Z坐标和W坐标的最大值，来确保顶点位于近裁剪平面之后。

output.clipPos = mul(unity\_MatrixVP, worldPos);

output.clipPos.z = max(output.clipPos.z, output.clipPos.w);

return output;

然而，剪剪空间的细节使情况变得复杂。直观上，把近裁剪平面的深度值视为0，距离越远值越大。但实际上，除了OpenGL API之外，其它的图形API都是相反的，近裁剪平面的深度值为1。对于OpenGL，近裁剪平面的深度值为-1。我们可以通过"mmon.hlsl"含文件中的UNITY\_REVERSED\_Z和UNITY\_NEAR\_CLIP\_VALUE宏来处理所有情况。

//output.clipPos.z = max(output.clipPos.z, output.clipPos.w);

#if UNITY\_REVERSED\_Z

output.clipPos.z =

min(output.clipPos.z, output.clipPos.w \* UNITY\_NEAR\_CLIP\_VALUE);

#else

output.clipPos.z =

max(output.clipPos.z, output.clipPos.w \* UNITY\_NEAR\_CLIP\_VALUE);

#endif

**2.2第二个Pass**

为了把"hadowCaster"Pass添加到光照着色器，我们复制它的Pass语块，并在第二个Pass语块中添加一个Tags语块，然后将LightMode设置为ShadowCaster。最后包含"ShadowCaster.hlsl"而不是"Lit.hlsl",并使用适当的顶点和片元函数。

Pass {

HLSLPROGRAM

#pragma target 3.5

#pragma multi\_compile\_instancing

#pragma instancing\_options assumeuniformscaling

#pragma vertex LitPassVertex

#pragma fragment LitPassFragment

#include "../ShaderLibrary/Lit.hlsl"

ENDHLSL

}

Pass {

Tags {

"LightMode" = "ShadowCaster"

}

HLSLPROGRAM

#pragma target 3.5

#pragma multi\_compile\_instancing

#pragma instancing\_options assumeuniformscaling

#pragma vertex ShadowCasterPassVertex

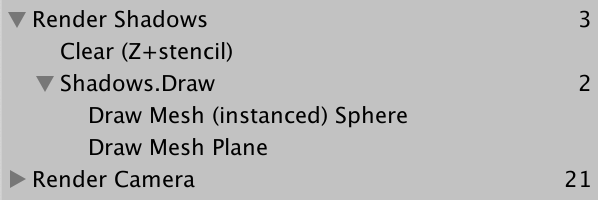
#pragma fragment ShadowCasterPassFragment

#include "../ShaderLibrary/ShadowCaster.hlsl"

ENDHLSL

}

现在物体被渲染到阴影映射纹理中来。由于此时物体不支持多光源，因此GPU实例化非常有效。



通过选择Shadows.Draw条目，您可以看到阴影贴图的最终内容。由于它是一个仅深度纹理，帧调试器将向我们显示深度信息，白色代表近和黑色远。