**定向光阴影**

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：

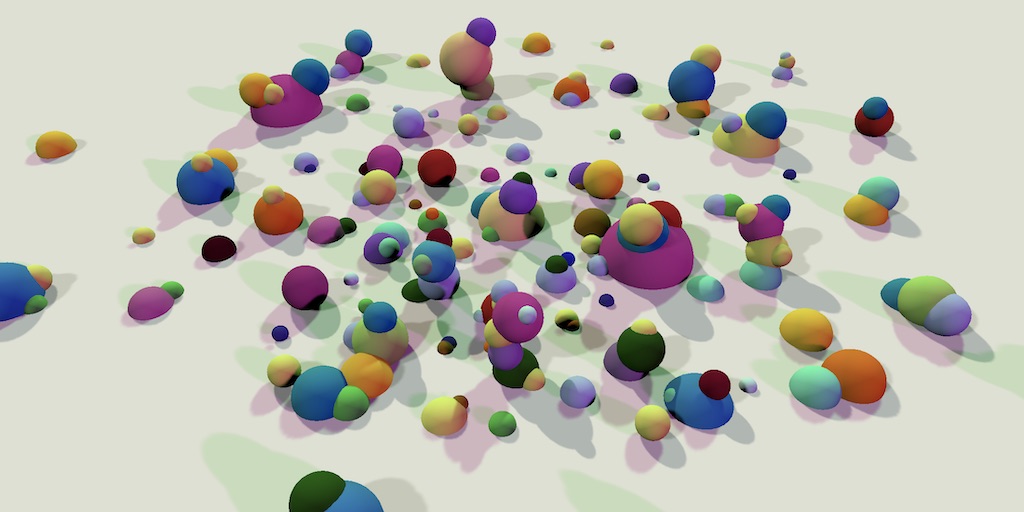
<https://catlikecoding.com/unity/tutorials/scriptable-render-pipeline/directional-shadows/>

**本章内容如下：**

* 支持多个带阴影的定向光源
* 控制阴影距离
* 定义一个单独的主光源
* 渲染并采样一个级联阴影贴图
* 处理剔除球

这是Unity可编程渲染管线系列教程的第五章。这次我们将添加对定向光阴影的支持。

本教程使用Unity 2018.3.0f2完成。



四个带阴影定向光源，一个带有级联（cascades）。

**一、定向光源的阴影**

从概念上讲，定向光源和聚光灯没有什么不同。它们是一样的，只是定向光是无限远的。我们对于聚光灯阴影的方法也适用于它们，只需要一些调整。我们加强管线以支持混合定向光源和聚光灯，让它们都可以投射阴影。

**1.1 配置阴影**

目前，ConfigureLights只在聚光灯的情况下处理阴影数据。但是当我们支持也支持定向光源时，同样的代码会被使用两次。因此复制相关的代码到单独的方法中，根据光源索引和阴影光源进行处理，然后返回阴影数据。

Vector4 ConfigureShadows (int lightIndex, Light shadowLight) {

Vector4 shadow = Vector4.zero;

Bounds shadowBounds;

if (

shadowLight.shadows != LightShadows.None &&

cull.GetShadowCasterBounds(lightIndex, out shadowBounds)

) {

shadowTileCount += 1;

shadow.x = shadowLight.shadowStrength;

shadow.y =

shadowLight.shadows == LightShadows.Soft ? 1f : 0f;

}

return shadow;

}

在ConfigureLights中为聚光的使用新的方法。

if (light.lightType == LightType.Spot) {

…

//Light shadowLight = light.light;

//Bounds shadowBounds;

//if (

// shadowLight.shadows != LightShadows.None &&

// cull.GetShadowCasterBounds(i, out shadowBounds)

//) {

// …

//}

shadow = ConfigureShadows(i, light.light);

}

并且对定向光源也使用它。

if (light.lightType == LightType.Directional) {

Vector4 v = light.localToWorld.GetColumn(2);

v.x = -v.x;

v.y = -v.y;

v.z = -v.z;

visibleLightDirectionsOrPositions[i] = v;

shadow = ConfigureShadows(i, light.light);

}

因为当渲染定向阴影贴图时有些不同，让我们表示正在这里处理定向光源。可以通过使用阴影数据的Z分量作为标志来实现。

shadow = ConfigureShadows(i, light.light);

shadow.z = 1f;

**1.2 渲染阴影**

目前为止，我们在阴影贴图中为定向光源保留了块（tiles），但是它目前还不能使用，因为RenderShadows正在计算聚光灯的矩阵。我们只能对聚光灯使用ComputeSpotShadowsMatricesAndCullingPrimitives，而必须使用不同的方法去计算定向光源的矩阵。

首先，从我们是否拥有有效阴影的检查中拉取ComputeSpotShadowMatircesAndCullingPrimitives的调用。这使得对定向光源做不同的检查变得轻松。

//if (!cull.ComputeSpotShadowMatricesAndCullingPrimitives(

// i, out viewMatrix, out projectionMatrix, out splitData

//)) {

// shadowData[i].x = 0f;

// continue;

//}

bool validShadows;

validShadows = cull.ComputeSpotShadowMatricesAndCullingPrimitives(

i, out viewMatrix, out projectionMatrix, out splitData

);

if (!validShadows) {

shadowData[i].x = 0f;

continue;

}

接下来，如果阴影数据指明我们有一个定向光源，则调用ComputeDirectionalShadowMatrixcesAndCullingPrimitives。这个方法有更多的参数，因为它允许我们使用并不在此做的阴影级联。我们提供的第一个参数是光源索引，接下来的是级联索引和级联的数量。对于索引为0和数量为1的情况，我们不使用级联。然后是一个三分量的向量，它定义了级联的拆分，为此我们将使用(1,0,0)。接着是整数形式的平铺大小，然后是阴影近平面的值，最后是矩阵和拆分数据的输出。

bool validShadows;

if (shadowData[i].z > 0f) {

validShadows =

cull.ComputeDirectionalShadowMatricesAndCullingPrimitives(

i, 0, 1, Vector3.right, (int)tileSize,

cull.visibleLights[i].light.shadowNearPlane,

out viewMatrix, out projectionMatrix, out splitData

);

}

else {

validShadows =

cull.ComputeSpotShadowMatricesAndCullingPrimitives(

i, out viewMatrix, out projectionMatrix, out splitData

);

}

if (!validShadows) {

shadowData[i].x = 0f;

continue;

}

这为我们提供了正确的定向光源阴影矩阵。除此之外，拆分数据也包含了一个有效的剔除球体。这个球体包围了所有需要被渲染来创建一个有效的定向光源贴图的对象。这对定向光阴影来说是有效的，不像聚光灯它们影响所有的对象。我们可以使用剔除球体去减少必须被渲染到阴影贴图的图形的数量，通过分配它到阴影设置的拆分数据中来实现这些。

var shadowSettings = new DrawShadowsSettings(cull, i);

shadowSettings.splitData.cullingSphere = splitData.cullingSphere;

context.DrawShadows(ref shadowSettings);

聚光灯没有有效的剔除球体，但是我们可以任意分配它们，因为是没有效果的。因此我们不必在此区分光源类型。

**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

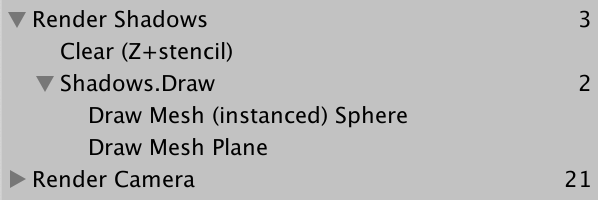
#pragma fragment ShadowCasterPassFragment

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

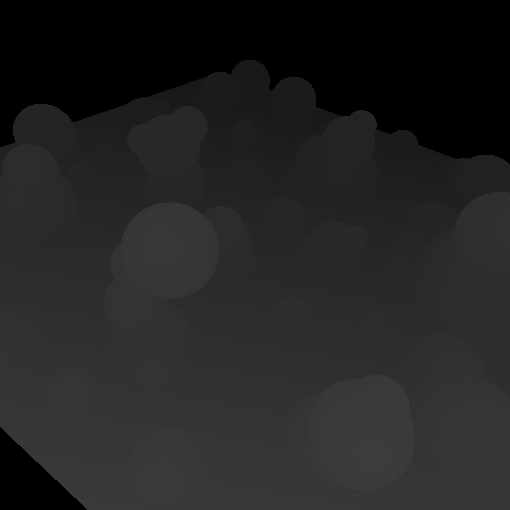
ENDHLSL

}

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。



在帧调试器中选择Shadows.Draw条目，可以看到阴影映射纹理的信息。犹豫它是一个深度纹理，所以帧调试器只向我们展示深度信息，白色代表近和黑色远。



因为阴影映射纹理是把摄像机放在聚光灯的位置渲染的，所以摄像机的方向与光源相匹配。如果光源旋转，阴影映射纹理也会发生对应的变化。

**三、阴影映射纹理的采样**

此时我们获得来一张阴影映射纹理，但我们还没有使用它。下一步是在以正常方式渲染物体时对阴影映射纹理进行采样。

**3.1从世界空间变换到阴影空间（Shadow Space）**

存储在阴影映射纹理中的深度信息仅在我们渲染阴影映射纹理时使用的裁剪空间中有效，我们称之为阴影空间。它与我们在以正常方式渲染物体时使用的空间不匹配。要知道片元相对于阴影映射纹理中深度的位置，我们必须将片元的坐标转换到阴影空间中。

第一步是在着色器中添加阴影映射纹理。我们通过着色器纹理变量来实现，我们将其命名为\_ShadowMap，并在MyPipeline获取其属性ID。

static int shadowMapId = Shader.PropertyToID("\_ShadowMap");

在命令缓冲区最后一次执行之前，通过调用它的SetGlobalTexture方法把阴影映射纹理绑定到着色器纹理变量上。

shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);

shadowBuffer.EndSample("Render Shadows");

接下来，在着色器中添加一个从世界空间到阴影空间的转换矩阵，将其名为\_WorldToShadowMatrix，并在MyPipeline获取其属性ID。

static int worldToShadowMatrixId =

Shader.PropertyToID("\_WorldToShadowMatrix");

该矩阵是将渲染阴影时使用的观察和投影矩阵相乘所得的，然后通过调用SetGlobalMatrix方法把它转递给GPU。

Matrix4x4 worldToShadowMatrix = projectionMatrix \* viewMatrix;

shadowBuffer.SetGlobalMatrix(worldToShadowMatrixId, worldToShadowMatrix);

shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);

我们可以通过SystemInfo.usesReversedZBuffer来判断裁剪空间的z维度是否反转。如果z维度反转，在矩阵相乘前需要把索引为2的行取反。通过直接调整矩阵的m20到m23字段来完成。

if (SystemInfo.usesReversedZBuffer) {

projectionMatrix.m20 = -projectionMatrix.m20;

projectionMatrix.m21 = -projectionMatrix.m21;

projectionMatrix.m22 = -projectionMatrix.m22;

projectionMatrix.m23 = -projectionMatrix.m23;

}

Matrix4x4 worldToShadowMatrix = projectionMatrix \* viewMatrix;

我们现在获得来从世界空间到阴影裁剪空间的转换矩阵。但是裁剪空间坐标分量的范围是从-1到1，而纹理坐标和深度的范围是从0到1。通过乘上一个额外的缩放和偏移矩阵我们可以将该范围变换烘焙到我们的转换矩阵中。我们可以通过Matrix4x4.TRS方法来生成所需的缩放和偏移矩阵。

var scaleOffset = Matrix4x4.TRS(

Vector3.one \* 0.5f, Quaternion.identity, Vector3.one \* 0.5f

);

Matrix4x4 worldToShadowMatrix =

scaleOffset \* (projectionMatrix \* viewMatrix);

但由于它是一个简单的矩阵，我们也可以简单地通过设置单位矩阵的适当字段来生成。

var scaleOffset = Matrix4x4.identity;

scaleOffset.m00 = scaleOffset.m11 = scaleOffset.m22 = 0.5f;

scaleOffset.m03 = scaleOffset.m13 = scaleOffset.m23 = 0.5f;

**3.2深度采样**

在Lit.hlsl文件中，为光源数据添加一个常量缓冲区，并在缓冲区中定义一个float4x4类型的矩阵\_WorldToShadowMatrix。

CBUFFER\_START(\_LightBuffer)

…

CBUFFER\_END

CBUFFER\_START(\_ShadowBuffer)

float4x4 \_WorldToShadowMatrix;

CBUFFER\_END

纹理资源不属于缓冲区，他们是单独定义的。因此，我们通过TEXTURE2D\_SHADOW宏来定义\_ShadowMap。

CBUFFER\_START(\_ShadowBuffer)

float4x4 \_WorldToShadowMatrix;

CBUFFER\_END

TEXTURE2D\_SHADOW(\_ShadowMap);

接下来，我们还必须定义用于纹理采样的采样器状态。通常这是通过SAMPLER宏完成的，但我们需要一个特殊的比较采样器，所以改用SAMPLER\_CMP宏。为了得到正确的采样器状态，它的命名必须是在纹理名字前加上"sampler"。

TEXTURE2D\_SHADOW(\_ShadowMap);

SAMPLER\_CMP(sampler\_ShadowMap);

创建一个以世界坐标为参数的ShadowAttenuation函数，返回值是光源阴影的衰减系数。首先要把世界坐标转换到阴影空间。

TEXTURE2D\_SHADOW(\_ShadowMap);

SAMPLER\_CMP(sampler\_ShadowMap);

float ShadowAttenuation (float3 worldPos) {

float4 shadowPos = mul(\_WorldToShadowMatrix, float4(worldPos, 1.0));

}

结果位置是定义在齐次坐标空间的，类似于裁剪空间的坐标转换。但我们需要常规坐标，因此将XYZ分量除以W分量。

float4 shadowPos = mul(\_WorldToShadowMatrix, float4(worldPos, 1.0));

shadowPos.xyz /= shadowPos.w;

现在我们可以使用SAMPLE\_TEXTURE2D\_SHADOW宏对阴影映射纹理进行采样来。它需要纹理，采样器状态和阴影空间的坐标作为参数。当坐标的z值小于阴影映射纹理中的深度值时，该宏返回1，这意味着该点距离光源更近，不在阴影中。否则，该宏返回值为0意味着该点在阴影中。因为采样器在双线性插值之前执行比较，所以阴影的边缘将混合阴影映射纹理的纹素。

shadowPos.xyz /= shadowPos.w;

return SAMPLE\_TEXTURE2D\_SHADOW(\_ShadowMap, sampler\_ShadowMap, shadowPos.xyz);

**3.3阴影中颜色的衰减**

为了影响光照，为DiffuseLight函数添加一个阴影衰减因子作为参数。把该因子和其它衰减因子一样乘以漫反射强度。

float3 DiffuseLight (

int index, float3 normal, float3 worldPos, float shadowAttenuation

) {

…

diffuse \*= shadowAttenuation \* spotFade \* rangeFade / distanceSqr;

return diffuse \* lightColor;

}

阴影不适用于顶点光照，因此在顶点着色器中用1作为衰减因子。

VertexOutput LitPassVertex (VertexInput input) {

…

output.vertexLighting = 0;

for (int i = 4; i < min(unity\_LightIndicesOffsetAndCount.y, 8); i++) {

int lightIndex = unity\_4LightIndices1[i - 4];

output.vertexLighting +=

DiffuseLight(lightIndex, output.normal, output.worldPos, 1);

}

return output;

}

在片元着色器中调用ShadowAttenuation方法，并把世界坐标作为参数传进去，再把结果传递给DiffuseLight方法。

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

…

float3 diffuseLight = input.vertexLighting;

for (int i = 0; i < min(unity\_LightIndicesOffsetAndCount.y, 4); i++) {

int lightIndex = unity\_4LightIndices0[i];

float shadowAttenuation = ShadowAttenuation(input.worldPos);

diffuseLight += DiffuseLight(

lightIndex, input.normal, input.worldPos, shadowAttenuation

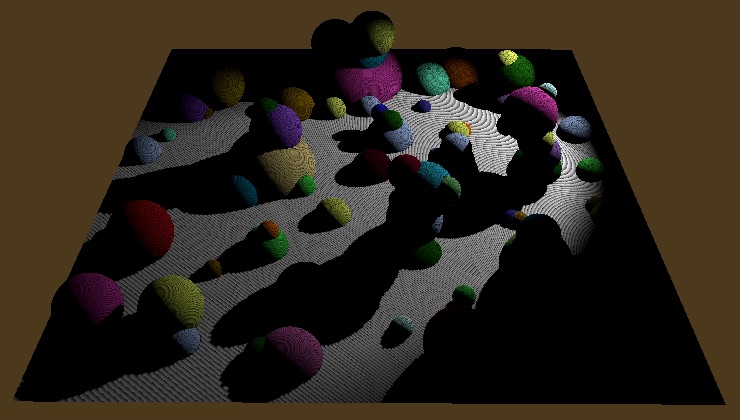
);

}

float3 color = diffuseLight \* albedo;

return float4(color, 1);

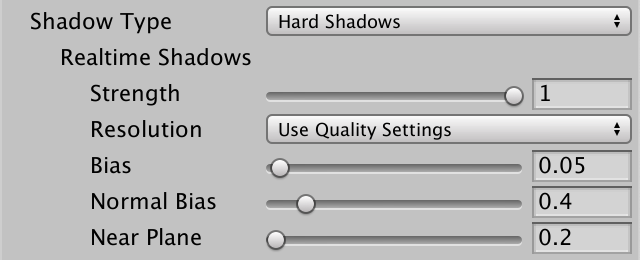
}



阴影终于出现来，但是带有严重的阴影“痤疮”（shadow acne）。

**四、阴影设置**

有多种方法可以控制阴影的质量和外观。我们将添加一些支持，包括：阴影分辨率，深度偏差，强度和soft阴影。可以通过光源的检视窗口设置这些配置。



**4.1阴影映射纹理的尺寸**

尽管光源的检视窗口可以设置阴影的分辨率，但这只能间接地调整阴影映射纹理的尺寸。在unity默认渲染管线中，阴影映射纹理的实际尺寸是通过质量设置控制的。我们使用自己的渲染管线，因此需要在MyPipelineAsset中添加阴影映射纹理的尺寸配置选项。

在MyPipelineAsset 中定义一个ShadowMapSize 枚举，枚举元素包括：256,512,1024,2048和4096。数字不能作为枚举标签，所以在每个数字前加上下划线。显示枚举选项时，Unity编辑器会忽略下划线。然后使用该枚举作为阴影映射纹理的配置字段。

public enum ShadowMapSize {

\_256,

\_512,

\_1024,

\_2048,

\_4096

}

[SerializeField]

ShadowMapSize shadowMapSize;

默认情况下，枚举选项的值从整数0开始。为了方便计算，我们可以通过为枚举选项分配显式数值来使其能够直接映射到对应的整数。

public enum ShadowMapSize {

\_256 = 256,

\_512 = 512,

\_1024 = 1024,

\_2048 = 2048,

\_4096 = 4096

}

这就意味着0不是一个合法的默认值，所以要为纹理尺寸字段设置一个默认值。

ShadowMapSize shadowMapSize = ShadowMapSize.\_1024;



把阴影映射纹理纹理的尺寸转成整数类型传递给自定义渲染管线的构造函数。

protected override IRenderPipeline InternalCreatePipeline () {

return new MyPipeline(

dynamicBatching, instancing, (int)shadowMapSize

);

}

在MyPipeline中添加一个字段保存纹理的尺寸，并在构造函数中初始化它。

int shadowMapSize;

public MyPipeline (

bool dynamicBatching, bool instancing, int shadowMapSize

) {

…

this.shadowMapSize = shadowMapSize;

}

在RenderShadows方法中获取纹理时，使用纹理尺寸字段。

void RenderShadows (ScriptableRenderContext context) {

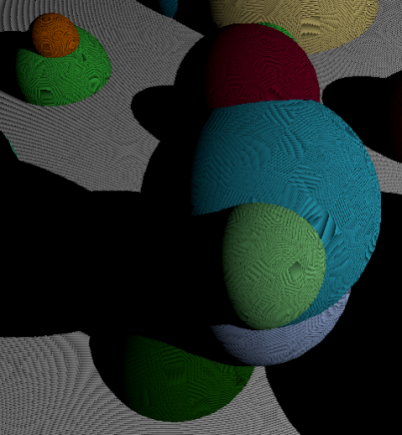
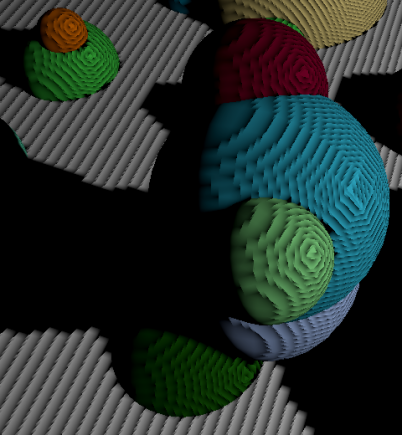
shadowMap = RenderTexture.GetTemporary(

shadowMapSize, shadowMapSize, 16, RenderTextureFormat.Shadowmap

);

…

}



256和4096尺寸的阴影映射纹理

**4.4阴影深度偏差**

阴影痤疮（shadow acne）是由从表面伸出的阴影映射纹理的纹素所引起的，详情请见Rendering 7, Shadows。我们将支持最简单的减轻阴影痤疮的方法，即在渲染阴影映射纹理时添加一个微小的深度偏移。由于每个光源都需要配置阴影偏差，所以我们要通过一个着色器属性\_ShadowBias把它传给GPU，并获取其属性ID。

static int shadowBiasId = Shader.PropertyToID("\_ShadowBias");

在RenderShadows函数中设置观察和投影矩阵的地方设置阴影偏差。VisibleLight有一个light字段，该字段中有阴影偏差值。

shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

shadowBuffer.SetGlobalFloat(

shadowBiasId, cull.visibleLights[0].light.shadowBias

);

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

将相应的变量添加到ShadowCaster.hlsl中的阴影映射缓冲区中。在修正之前，使用它来偏移裁剪空间坐标的Z分量。如果Z反转，则应减去偏差，否则添加偏差。

CBUFFER\_START(\_ShadowCasterBuffer)

float \_ShadowBias;

CBUFFER\_END

…

VertexOutput ShadowCasterPassVertex (VertexInput input) {

…

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

#if UNITY\_REVERSED\_Z

output.clipPos.z -= \_ShadowBias;

output.clipPos.z =

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

#else

output.clipPos.z += \_ShadowBias;

output.clipPos.z =

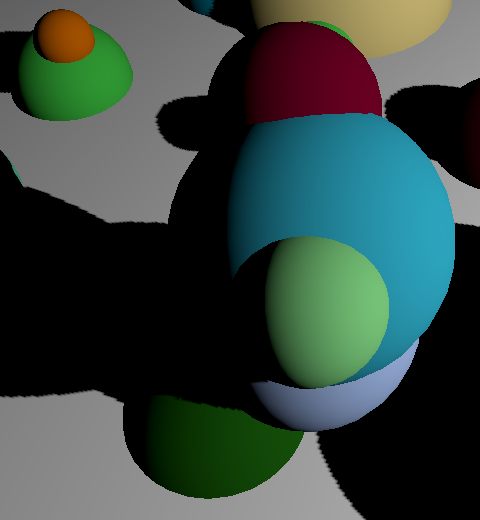
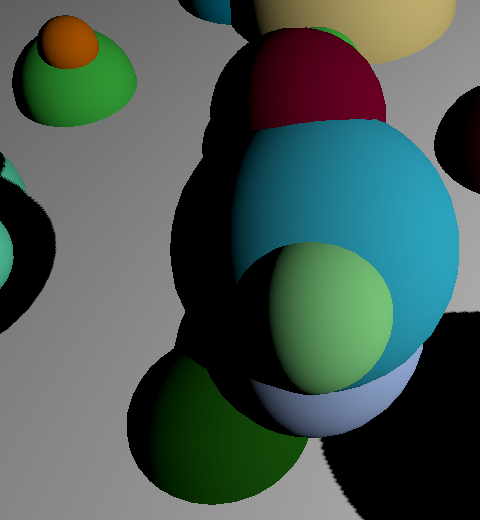
max(output.clipPos.z, output.clipPos.w \* UNITY\_NEAR\_CLIP\_VALUE);

#endif

return output;

}

阴影深度偏差应该尽可能小，以防止阴影偏移太远而导致部分阴影消失（peter-panning？彼得潘？）。



阴影深度偏移为0.05和0.01

**4.3阴影强度**

由于我们使用单一的光源并且没有任何环境光，因此我们的阴影完全是黑色的。但我们可以降低阴影的强度来使阴影淡化。这就像所有的阴影映射物体都是半透明的。我们将通过\_ShadowStrength属性将阴影强度传给着色器，因此需要设置属性ID。

static int shadowStrengthId = Shader.PropertyToID("\_ShadowStrength");

在对阴影映射纹理进行采样时需要使用阴影强度，因此将其与世界-阴影矩阵和阴影映射纹理本身一起设置。像深度偏差一样，我们可以从Light组件中获取它。

shadowBuffer.SetGlobalMatrix(worldToShadowMatrixId, worldToShadowMatrix);

shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);

shadowBuffer.SetGlobalFloat(

shadowStrengthId, cull.visibleLights[0].light.shadowStrength

);

shadowBuffer.EndSample("Render Shadows");

将阴影强度添加到阴影缓冲区，然后在ShadowAttenuation函数中使用它在1和采样衰减因子之间进行插值。

CBUFFER\_START(\_ShadowBuffer)

float4x4 \_WorldToShadowMatrix;

float \_ShadowStrength;

CBUFFER\_END

TEXTURE2D\_SHADOW(\_ShadowMap);

SAMPLER\_CMP(sampler\_ShadowMap);

float ShadowAttenuation (float3 worldPos) {

float4 shadowPos = mul(\_WorldToShadowMatrix, float4(worldPos, 1.0));

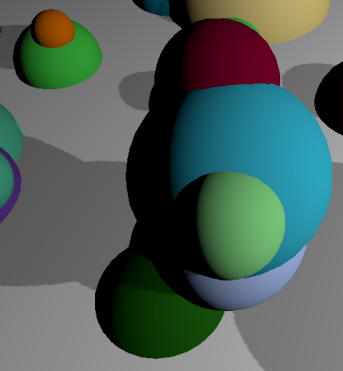
shadowPos.xyz /= shadowPos.w;

float attenuation =

SAMPLE\_TEXTURE2D\_SHADOW(\_ShadowMap, sampler\_ShadowMap, shadowPos.xyz);

return lerp(1, attenuation, \_ShadowStrength);

}



**4.4 soft阴影**

我们最后需要支持的设置是hard阴影和soft阴影之间的切换。我们目前正在使用hard阴影，这意味着阴影边缘的唯一平滑是由采样阴影映射纹理时的双线性插值引起的。当启用平滑阴影时，阴影过渡变得模糊，会出现较大的半影区域（阴影边缘）。然而，这与现实中的半影不同，现实中半影是均匀的，而不是取决于光源、阴影映射物体和阴影接收物体之间的空间关系。

通过多次对阴影映射纹理进行采样来产生soft阴影，距离原始样本位置越远对最终值的贡献越小。我们将使用5×5的tent filter，这需要9个纹理样本。我们可以使用核心库包含文件Shadow / ShadowSamplingTent.hlsl中定义的函数，在Lit.hlsl中包含该文件。

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

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

tent filter需要知道阴影映射纹理的尺寸。该方法要求一个特定的向量，四个分量分别为宽的倒数、高的倒数、宽度、高度。我们将其添加到阴影缓存区。

CBUFFER\_START(\_ShadowBuffer)

float4x4 \_WorldToShadowMatrix;

float \_ShadowStrength;

float4 \_ShadowMapSize;

CBUFFER\_END

在MyPipeline获取对应的属性ID。

static int shadowMapSizeId = Shader.PropertyToID("\_ShadowMapSize");

在RenderShadows函数结尾设置该向量。

float invShadowMapSize = 1f / shadowMapSize;

shadowBuffer.SetGlobalVector(

shadowMapSizeId, new Vector4(

invShadowMapSize, invShadowMapSize, shadowMapSize, shadowMapSize

)

);

shadowBuffer.EndSample("Render Shadows");

当\_SHADOWS\_SOFT 关键字被定义时，在ShadowAttenuation方法中我们用tent filter替换常规的阴影映射纹理采样。

float attenuation =

SAMPLE\_TEXTURE2D\_SHADOW(\_ShadowMap, sampler\_ShadowMap, shadowPos.xyz);

#if defined(\_SHADOWS\_SOFT)

#endif

return lerp(1, attenuation, \_ShadowStrength);

不同于单次采样，我们需要叠加9次采样来创建5×5的tent filter。把阴影映射纹理的尺寸和xy坐标作为参数传给SampleShadow\_ComputeSamples\_Tent\_5x5函数，可以获得权重和UV坐标。权重和UV坐标通过两个输出参数传出，一个float数组和一个float2数组，它们长度都是9。

#if defined(\_SHADOWS\_SOFT)

float tentWeights[9];

float2 tentUVs[9];

SampleShadow\_ComputeSamples\_Tent\_5x5(

\_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs

);

#endif

然而，该函数的输出参数不是float类型而是real类型。这不是一个真正的数值类型，而是一个宏，它会视情况创建一个float或者half变量。通常情况下我们可以无视，但是为了避免在某些平台报错，最好使用real作为输出参数的类型。

real tentWeights[9];

real2 tentUVs[9];

我们用权重和UV坐标在循环中对阴影映射纹理采样9次由于这是一个固定长度的循环体，所以shader编译器会把它展开。我们需要用阴影坐标的z分量和UV坐标一起构建一个float3类型来进行采样。

#if defined(\_SHADOWS\_SOFT)

real tentWeights[9];

real2 tentUVs[9];

SampleShadow\_ComputeSamples\_Tent\_5x5(

\_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs

);

attenuation = 0;

for (int i = 0; i < 9; i++) {

attenuation += tentWeights[i] \* SAMPLE\_TEXTURE2D\_SHADOW(

\_ShadowMap, sampler\_ShadowMap, float3(tentUVs[i].xy, shadowPos.z)

);

}

#endif

为了启用soft阴影，当\_SHADOWS\_SOFT关键字被定义时，我们必须在Lit着色器的默认pass中添加一个多重编译指令来创建一个着色器pass变量。我们希望根据\_SHADOWS\_SOFT关键字是否被定义来创建对应的两个变量。我们可以在通过一个下划线表示没有定义关键字，后面跟着\_SHADOWS\_SOFT关键字。

#pragma multi\_compile\_instancing

#pragma instancing\_options assumeuniformscaling

#pragma multi\_compile \_ \_SHADOWS\_SOFT

最后在RenderShadows函数的末尾，要根据光源的shadows属性切换关键字的开关。当设置成LightShadows.Soft时，在阴影命令缓冲区中调用EnableShaderKeyword函数，否则调用DisableShaderKeyword函数。unity会在渲染时根据关键字状态去使用不同的pass变量。

const string shadowsSoftKeyword = "\_SHADOWS\_SOFT";

…

void RenderShadows (ScriptableRenderContext context) {

…

if (cull.visibleLights[0].light.shadows == LightShadows.Soft) {

shadowBuffer.EnableShaderKeyword(shadowsSoftKeyword);

}

else {

shadowBuffer.DisableShaderKeyword(shadowsSoftKeyword);

}

shadowBuffer.EndSample("Render Shadows");

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

}

由于根据bool值切换关键字很常见，我们也可以使用CoreUtils.SetKeyword函数来完成相同的工作。

//if (cull.visibleLights[0].light.shadows == LightShadows.Soft) {

// shadowBuffer.EnableShaderKeyword(shadowsSoftKeyword);

//}

//else {

// shadowBuffer.DisableShaderKeyword(shadowsSoftKeyword);

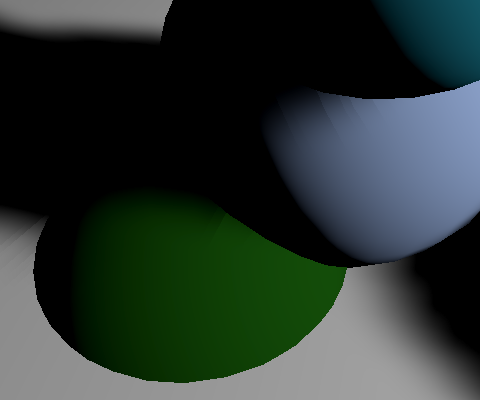
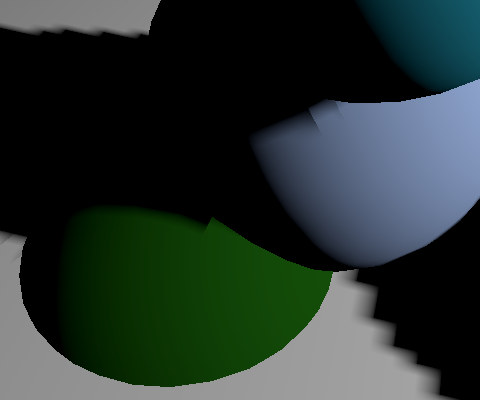
//}

CoreUtils.SetKeyword(

shadowBuffer, shadowsSoftKeyword,

cull.visibleLights[0].light.shadows == LightShadows.Soft

);



hard阴影和soft阴影

**五、多光源阴影**

目前我们只支持单光源的阴影，但是我们的渲染管线最多支持16个光源。所以，我们需要同时支持16个聚光灯阴影。

**5.1每个光源的阴影数据**

如果我们支持多阴影光源，同时仍然在一次pass中完成所有光照，那么所有的阴影数据，比如阴影强度，必须同时传给GPU。我们在ConfigureLights函数中采集这些数据，就像我们设置其它光源相关的数据一样。所以我们要在调用RenderShadows之前调用ConfigureLights，并且只在有可见光时调用RenderShadows。

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

if (cull.visibleLights.Count > 0) {

ConfigureLights();

RenderShadows(context);

}

else {

cameraBuffer.SetGlobalVector(

lightIndicesOffsetAndCountID, Vector4.zero

);

}

ConfigureLights();

context.SetupCameraProperties(camera);

CameraClearFlags clearFlags = camera.clearFlags;

cameraBuffer.ClearRenderTarget(

(clearFlags & CameraClearFlags.Depth) != 0,

(clearFlags & CameraClearFlags.Color) != 0,

camera.backgroundColor

);

//if (cull.visibleLights.Count > 0) {

// ConfigureLights();

//}

//else {

// cameraBuffer.SetGlobalVector(

// lightIndicesOffsetAndCountID, Vector4.zero

// );

//}

我们使用vector4数组存储阴影数据，每个光源对应一个元素。在ConfigureLights函数中循环处理光源时把每个元素都初始化为0，就像处理衰减数据一样。

Vector4[] shadowData = new Vector4[maxVisibleLights];

…

void ConfigureLights () {

for (int i = 0; i < cull.visibleLights.Count; i++) {

if (i == maxVisibleLights) {

break;

}

VisibleLight light = cull.visibleLights[i];

visibleLightColors[i] = light.finalColor;

Vector4 attenuation = Vector4.zero;

attenuation.w = 1f;

Vector4 shadow = Vector4.zero;

…

visibleLightAttenuations[i] = attenuation;

shadowData[i] = shadow;

}

…

}

当光源类型时聚光灯时，获得光源的light组件的引用。如果它的shadows属性不是LightShadows.None，就它把的阴影强度存到向量的x分量上。

if (light.lightType == LightType.Spot) {

…

Light shadowLight = light.light;

if (shadowLight.shadows != LightShadows.None) {

shadow.x = shadowLight.shadowStrength;

}

}

每个光源都可能使用hard或者soft阴影，我们把该属性存到向量的y分量上。1表示soft阴影，0表示hard阴影。

if (shadowLight.shadows != LightShadows.None) {

shadow.x = shadowLight.shadowStrength;

shadow.y =

shadowLight.shadows == LightShadows.Soft ? 1f : 0f;

}

**5.2排除光源**

一个光源可见并且开启了阴影并不能保证一定需要阴影映射纹理。如果在光源的视角中并没有任何阴影的投射物或接受物，自然不需要阴影贴图。我们可以调用剔除结果的 GetShadowCasterBounds函数，传入一个光源索引来检查该光源是否需要阴影贴图。该函数会检查光源的shadow volume是否在一个有效的范围内。如果不在，我们就跳过设置阴影数据。尽管不需要，但是我们还是得提供一个shadow bound作为输出参数。

Light shadowLight = light.light;

Bounds shadowBounds;

if (

shadowLight.shadows != LightShadows.None &&

cull.GetShadowCasterBounds(i, out shadowBounds)

) {

shadow.x = shadowLight.shadowStrength;

shadow.y =

shadowLight.shadows == LightShadows.Soft ? 1f : 0f;

}

**5.3渲染所有的阴影映射纹理**

接下来回到RenderShadows函数中，我们把首次执行阴影缓冲区和设置阴影映射纹理之间的代码用一个循环包起来。在循环中我们要放弃超过支持最大光源数量的光源。并把其中所有原本固定的索引0替换为迭代值变量i。

shadowBuffer.BeginSample("Render Shadows");

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

for (int i = 0; i < cull.visibleLights.Count; i++) {

if (i == maxVisibleLights) {

break;

}

Matrix4x4 viewMatrix, projectionMatrix;

ShadowSplitData splitData;

cull.ComputeSpotShadowMatricesAndCullingPrimitives(

i, out viewMatrix, out projectionMatrix, out splitData

);

shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

shadowBuffer.SetGlobalFloat(

shadowBiasId, cull.visibleLights[i].light.shadowBias

);

context.ExecuteCommandBuffer(shadowBuffer);

shadowBuffer.Clear();

var shadowSettings = new DrawShadowsSettings(cull, i);

context.DrawShadows(ref shadowSettings);

…

shadowBuffer.SetGlobalMatrix(

worldToShadowMatrixId, worldToShadowMatrix

);

}

shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);

我们跳过不需要阴影映射纹理的光源，我们用阴影数据中的阴影强度来判断。小于等于0（有可能原本的强度就是0，也有可能是我们之前设0来跳过）就直接用continue跳到下个迭代。

if (i == maxVisibleLights) {

break;

}

if (shadowData[i].x <= 0f) {

continue;

}

ComputeSpotShadowMatricesAndCullingPrimitives函数返回是否能够生成可用的矩阵，它的结果应该和GetShadowCasterBounds函数一致，但是还要确保返回失败时把阴影强度设为0并跳过。

Matrix4x4 viewMatrix, projectionMatrix;

ShadowSplitData splitData;

if (!cull.ComputeSpotShadowMatricesAndCullingPrimitives(

i, out viewMatrix, out projectionMatrix, out splitData

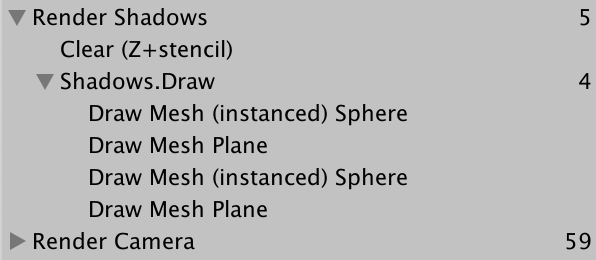
)) {

shadowData[i].x = 0f;

continue;

}

当有多个带阴影的光源时（只要它们的位置能够产生可见的阴影），帧调试器会显示我们实际上渲染了多次阴影映射纹理。



然而，由此产生的阴影一团糟，我们还有一些工作要做。

**5.4使用正确的阴影数据**

现在需要把一个阴影数据数组传给GPU，而不只是一个阴影强度数。所以，需要修改MyPipeline。

//static int shadowStrengthId = Shader.PropertyToID("\_ShadowStrength");

static int shadowDataId = Shader.PropertyToID("\_ShadowData");

…

void RenderShadows (ScriptableRenderContext context) {

…

//shadowBuffer.SetGlobalFloat(

// shadowStrengthId, cull.visibleLights[0].light.shadowStrength

//);

shadowBuffer.SetGlobalVectorArray(shadowDataId, shadowData);

…

}

同样的，我们需要世界-阴影矩阵的数组，而不是单独一个矩阵。在RenderShadows函数的循环中填充数组，接着将数组传给GPU。

//static int worldToShadowMatrixId =

// Shader.PropertyToID("\_WorldToShadowMatrix");

static int worldToShadowMatricesId =

Shader.PropertyToID("\_WorldToShadowMatrices");

…

void RenderShadows (ScriptableRenderContext context) {

…

for (int i = 0; i < cull.visibleLights.Count; i++) {

…

//Matrix4x4 worldToShadowMatrix =

// scaleOffset \* (projectionMatrix \* viewMatrix);

//shadowBuffer.SetGlobalMatrix(

// worldToShadowMatrixId, worldToShadowMatrix

//);

worldToShadowMatrices[i] =

scaleOffset \* (projectionMatrix \* viewMatrix);

}

shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);

shadowBuffer.SetGlobalMatrixArray(

worldToShadowMatricesId, worldToShadowMatrices

);

shadowBuffer.SetGlobalVectorArray(shadowDataId, shadowData);

…

}

在着色器里，对阴影缓冲区的内容做相应的调整。

CBUFFER\_START(\_ShadowBuffer)

//float4x4 \_WorldToShadowMatrix;

//float \_ShadowStrength;

float4x4 \_WorldToShadowMatrices[MAX\_VISIBLE\_LIGHTS];

float4 \_ShadowData[MAX\_VISIBLE\_LIGHTS];

float4 \_ShadowMapSize;

CBUFFER\_END

ShadowAttenuation 方法需要新增一个光源索引参数以便取得正确的数组元素。当然，首先我们要检查阴影强度是否为正数，如果不是，直接将1作为衰减因子返回。我们基于阴影数据的y分量来进行条件分支，而不是依赖\_SHADOWS\_SOFT关键字。

float ShadowAttenuation (int index, float3 worldPos) {

if (\_ShadowData[index].x <= 0) {

return 1.0;

}

float4 shadowPos = mul(\_WorldToShadowMatrices[index], float4(worldPos, 1.0));

shadowPos.xyz /= shadowPos.w;

float attenuation;

if (\_ShadowData[index].y == 0) {

attenuation =

SAMPLE\_TEXTURE2D\_SHADOW(\_ShadowMap, sampler\_ShadowMap, shadowPos.xyz);

}

//#if defined(\_SHADOWS\_SOFT)

else {

real tentWeights[9];

real2 tentUVs[9];

SampleShadow\_ComputeSamples\_Tent\_5x5(

\_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs

);

attenuation = 0;

for (int i = 0; i < 9; i++) {

attenuation += tentWeights[i] \* SAMPLE\_TEXTURE2D\_SHADOW(

\_ShadowMap, sampler\_ShadowMap, float3(tentUVs[i].xy, shadowPos.z)

);

}

}

//#endif

return lerp(1, attenuation, \_ShadowData[index].x);

}

最后在片元着色器中把光源索引传给ShadowAttenuation函数。

float shadowAttenuation = ShadowAttenuation(lightIndex, input.worldPos);

**5.5阴影映射纹理图集**

虽然我们有了正确的阴影数据和矩阵，但是在多个阴影光源时，最终产生的仍然是错误的阴影。这是因为所有的阴影映射纹理都渲染进了同一张纹理之中，产生来一个合并的无意义的结果。Unity轻量级渲染管线通过阴影映射纹理图集来解决这一问题。将渲染纹理分割为多个平铺块，每个光源使用一个。我们也使用这种方法。

我们最多支持16个光源，所以要把单张阴影映射纹理分成4x4的平铺块网格。每个平铺块尺寸应该是阴影映射纹理的尺寸除以4。我们要将渲染时的视口约束在这个尺寸，所以在RenderShadows函数开头创建一个Rect结构体，并填充合适的值。

void RenderShadows (ScriptableRenderContext context) {

float tileSize = shadowMapSize / 4;

Rect tileViewport = new Rect(0f, 0f, tileSize, tileSize);

…

}

在我们设置视口和投影矩阵的地方，用命令缓冲区的SetViewport函数通知GPU使用合适的视口尺寸。

shadowBuffer.SetViewport(tileViewport);

shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);



现在所有的阴影映射纹理都出现在渲染纹理一角的单个平铺块中。下一步就是偏移每个光源的视口。我们可以依据每个平铺块的xy索引得到视口位置。Y轴偏移索引通过光源索引整除4得到，x轴偏移索引通过整数取余得到。最终视口的xy坐标等于xy索引乘以平铺块大小。

float tileOffsetX = i % 4;

float tileOffsetY = i / 4;

tileViewport.x = tileOffsetX \* tileSize;

tileViewport.y = tileOffsetY \* tileSize;

shadowBuffer.SetViewport(tileViewport);



这样的图集有一个缺点，在一个平铺块边缘采样时，可能会在两个平铺块之间插值，从而导致错误的结果。当使用soft阴影时情况会更加的糟糕，因为tent filter可能会在离原始采样点偏移最多4个纹素的地方采样。相比混合相邻的平铺块，能够淡出阴影肯定更好。所以通过限制GPU在比视口小一点的区域内写入数据，来使每个平铺块周围添加一圈空值边缘，这被称为裁剪（scissoring）。我们可以传递一个比视口略小的矩形给shadowBuffer.EnableScissorRect方法来实现裁剪。我们需要四个纹素的边缘，所以这个矩形位置应该是视口位置加4，尺寸为视口大小减8。

shadowBuffer.SetViewport(tileViewport);

shadowBuffer.EnableScissorRect(new Rect(

tileViewport.x + 4f, tileViewport.y + 4f,

tileSize - 8f, tileSize - 8f

));



在完成阴影渲染之后，我们需要通过调用DisableScissorRect方法来禁用裁剪矩形，否则正常的渲染也会受影响。

shadowBuffer.DisableScissorRect();

shadowBuffer.SetGlobalTexture(shadowMapId, shadowMap);

最后要做的就是调整世界-阴影矩阵，让它能在正确的平铺块采样。我们可以乘以一个有适当的缩放偏移矩阵。着色器不需要关心我们是否使用了图集。

var scaleOffset = Matrix4x4.identity;

scaleOffset.m00 = scaleOffset.m11 = scaleOffset.m22 = 0.5f;

scaleOffset.m03 = scaleOffset.m13 = scaleOffset.m23 = 0.5f;

worldToShadowMatrices[i] =

scaleOffset \* (projectionMatrix \* viewMatrix);

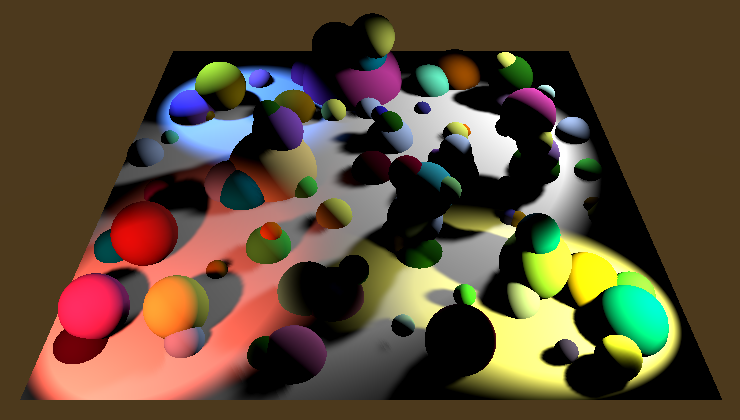
var tileMatrix = Matrix4x4.identity;

tileMatrix.m00 = tileMatrix.m11 = 0.25f;

tileMatrix.m03 = tileOffsetX \* 0.25f;

tileMatrix.m13 = tileOffsetY \* 0.25f;

worldToShadowMatrices[i] = tileMatrix \* worldToShadowMatrices[i];



要记住我们现在每个物体最多支持4个像素光，所以你让第5个聚光源照射到平面时，其中一个光源会变为顶点光源，进而无法接受该光源的阴影。

**六、动态平铺**

使用阴影映射纹理图集的优点是无论有多少阴影映射纹理，我们用的都是同一张渲染纹理，所以纹理占用的内存是固定的。缺点则是每个光源对应阴影映射纹理的特定部分，所以最终的阴影贴图分辨率会比我们想象的要低，并且可能大部分的纹理都没有被利用。



要充分利用我们的纹理就不能总是把它分成16个平铺块。我们可以更加实际需要多少平铺块来动态的确定平铺块的尺寸。这种方式可以确保至少能用到一半的纹理。

**6.1计算阴影平铺块的数量**

首先，我们需要明确我们需要多少个平铺块。我们可以在ConfigureLights中记录我们有多少带阴影的聚光灯。并用一个字段记录总数以便在之后使用。

int shadowTileCount;

…

void ConfigureLights () {

shadowTileCount = 0;

for (int i = 0; i < cull.visibleLights.Count; i++) {

…

else {

…

if (light.lightType == LightType.Spot) {

…

if (

shadowLight.shadows != LightShadows.None &&

cull.GetShadowCasterBounds(i, out shadowBounds)

) {

shadowTileCount += 1;

shadow.x = shadowLight.shadowStrength;

shadow.y =

shadowLight.shadows == LightShadows.Soft ? 1f : 0f;

}

}

}

…

}

…

}

**6.2分割阴影映射纹理**

接着，在RenderShadows函数开头就要算好如何分割阴影映射纹理。我用一个整数变量（split）来记录，如果我们只要一个平铺块，就不需要分割，所以split值为1；否则如果平铺块数量小于等于4，split值为2；小于等于9，值为3；大于9，值才为4。

void RenderShadows (ScriptableRenderContext context) {

int split;

if (shadowTileCount <= 1) {

split = 1;

}

else if (shadowTileCount <= 4) {

split = 2;

}

else if (shadowTileCount <= 9) {

split = 3;

}

else {

split = 4;

}

…

}

平铺块的尺寸可以通过阴影映射纹理的尺寸整除split值得到。这意味着在除以3的时候我们会舍弃部分纹素。平铺块的缩放值应该改为1f/split（浮点除法）。然后，在调整世界-阴影矩阵的缩放和平移时也要使用split值。

float tileSize = shadowMapSize / split;

float tileScale = 1f / split;

…

for (int i = 0; i < cull.visibleLights.Count; i++) {

…

float tileOffsetX = i % split;

float tileOffsetY = i / split;

…

tileMatrix.m00 = tileMatrix.m11 = tileScale;

tileMatrix.m03 = tileOffsetX \* tileScale;

tileMatrix.m13 = tileOffsetY \* tileScale;

…

}

为了在正确的位置打包所有的阴影映射纹理，必须只在设置好一个平铺块后才递增其索引。因此我们使用独立的一个变量记录而不是直接使用光源索引。在没我们没有跳过的迭代的末尾自递增索引。

int tileIndex = 0;

for (int i = 0; i < cull.visibleLights.Count; i++) {

…

float tileOffsetX = tileIndex % split;

float tileOffsetY = tileIndex / split;

…

tileIndex += 1;

}



**6.3一个平铺块就是没有平铺**

最后，如果只有一个平铺块就不需要设置视口和裁剪（scissor）矩阵，因此只在需要多个平铺块时设置这些。

for (int i = 0; i < cull.visibleLights.Count; i++) {

…

if (split > 1) {

shadowBuffer.SetViewport(tileViewport);

shadowBuffer.EnableScissorRect(new Rect(

tileViewport.x + 4f, tileViewport.y + 4f,

tileSize - 8f, tileSize - 8f

));

}

shadowBuffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

…

if (split > 1) {

var tileMatrix = Matrix4x4.identity;

tileMatrix.m00 = tileMatrix.m11 = tileScale;

tileMatrix.m03 = tileOffsetX \* tileScale;

tileMatrix.m13 = tileOffsetY \* tileScale;

worldToShadowMatrices[i] = tileMatrix \* worldToShadowMatrices[i];

}

tileIndex += 1;

}

if (split > 1) {

shadowBuffer.DisableScissorRect();

}

**6.4着色器关键字**

目前，我们每个片元最多采样阴影4次，它们可以是soft或者hard阴影的组合。最麻烦的情况就是4个软阴影，一共需要采样36次。在着色器中有多个分支可以根据不同情况进行阴影采样，由于同一物体的片元最终使用的是同一条分支，所以它非常有效。当然，我们可以根据不同的阴影组合切换到更简单的备选着色器。

一共有四种情况，第一种是完全没有阴影，第二种只有hard阴影，第三种只有soft阴影，最复杂的一种就是soft/hard阴影的组合。通过使用独立的\_SHADOWS\_HARD和 \_SHADOWS\_SOFT关键字，我们可以得到着色器变体来处理所有可能的情况。

在RenderShadows函数中，根据阴影信息的Y分量来使用两个布尔变量单独记录是否使用了soft/hard阴影。在循环之后使用这些布尔值切换着色器关键字。

const string shadowsHardKeyword = "\_SHADOWS\_HARD";

…

void RenderShadows (ScriptableRenderContext context) {

…

int tileIndex = 0;

bool hardShadows = false;

bool softShadows = false;

for (int i = 0; i < cull.visibleLights.Count; i++) {

…

if (shadowData[i].y <= 0f) {

hardShadows = true;

}

else {

softShadows = true;

}

}

…

CoreUtils.SetKeyword(shadowBuffer, shadowsHardKeyword, hardShadows);

CoreUtils.SetKeyword(shadowBuffer, shadowsSoftKeyword, softShadows);

…

}

在着色器中添加另一个多重编译指令，\_SHADOWS\_HARD。

#pragma multi\_compile \_ \_SHADOWS\_HARD

#pragma multi\_compile \_ \_SHADOWS\_SOFT

在ShadowAttenuation函数的开头，如果没有关键字被定义直接返回1。这样会函数的后面部分不会执行，完全消除阴影。

float ShadowAttenuation (int index, float3 worldPos) {

#if !defined(\_SHADOWS\_HARD) && !defined(\_SHADOWS\_SOFT)

return 1.0;

#endif

if (\_ShadowData[index].x <= 0) {

return 1.0;

}

…

}

为了让代码更清晰，把soft/hard阴影的采样代码移到单独的函数中去。

float HardShadowAttenuation (float4 shadowPos) {

return SAMPLE\_TEXTURE2D\_SHADOW(\_ShadowMap, sampler\_ShadowMap, shadowPos.xyz);

}

float SoftShadowAttenuation (float4 shadowPos) {

real tentWeights[9];

real2 tentUVs[9];

SampleShadow\_ComputeSamples\_Tent\_5x5(

\_ShadowMapSize, shadowPos.xy, tentWeights, tentUVs

);

float attenuation = 0;

for (int i = 0; i < 9; i++) {

attenuation += tentWeights[i] \* SAMPLE\_TEXTURE2D\_SHADOW(

\_ShadowMap, sampler\_ShadowMap, float3(tentUVs[i].xy, shadowPos.z)

);

}

return attenuation;

}

float ShadowAttenuation (int index, float3 worldPos) {

…

float attenuation;

if (\_ShadowData[index].y == 0) {

attenuation = HardShadowAttenuation(shadowPos);

}

else {

attenuation = SoftShadowAttenuation(shadowPos);

}

return lerp(1, attenuation, \_ShadowData[index].x);

}

现在根据关键字去完成另外三种情况下的代码。原始的分支时两个关键字都被定义来。

#if defined(\_SHADOWS\_HARD)

#if defined(\_SHADOWS\_SOFT)

if (\_ShadowData[index].y == 0) {

attenuation = HardShadowAttenuation(shadowPos);

}

else {

attenuation = SoftShadowAttenuation(shadowPos);

}

#else

attenuation = HardShadowAttenuation(shadowPos);

#endif

#else

attenuation = SoftShadowAttenuation(shadowPos);

#endif

最后，如果不需要阴影平铺块 ，在MyPipeline.Render里直接跳过RenderShadows方法。我们甚至都不需要阴影映射纹理。如果我们跳过了，我们要确保两个阴影的关键字都禁用了。没有可见光时我们也要把两者禁用掉。

if (cull.visibleLights.Count > 0) {

ConfigureLights();

if (shadowTileCount > 0) {

RenderShadows(context);

}

else {

cameraBuffer.DisableShaderKeyword(shadowsHardKeyword);

cameraBuffer.DisableShaderKeyword(shadowsSoftKeyword);

}

}

else {

cameraBuffer.SetGlobalVector(

lightIndicesOffsetAndCountID, Vector4.zero

);

cameraBuffer.DisableShaderKeyword(shadowsHardKeyword);

cameraBuffer.DisableShaderKeyword(shadowsSoftKeyword);

下一章时平行光阴影