

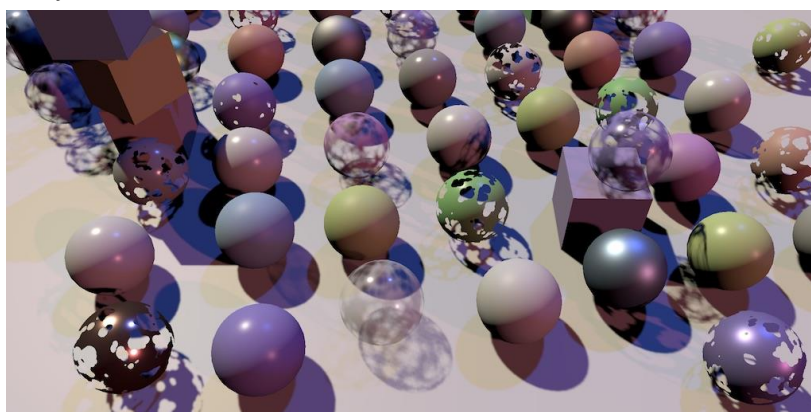
# 定向阴影

## 级联阴影贴图

渲染并采样阴影贴图  
支持多阴影定向光  
使用级联阴影贴图  
混合，淡化和过滤阴影

这是关于创建自定义可编程渲染管线系列教程的第四部分。添加了对级联阴影贴图的支持。

本教程基于 Unity 2019.2.14f1。



避免光线到达不该到达的地方

## 1. 渲染阴影

当绘制对象时，表面和光源数据足够用来计算光照。但在两者之间可能存在某些遮挡光线的东西，在我们绘制的表面投射了阴影。为了实现阴影，我们必须通过某种方式让着色器知道投射阴影的对象。对此有多种技术。最常见的方法是生成一张阴影贴图，用来记录光线从光源发出后在到达表面之前行进了多远。同方向上任何更远的对象都无法被同一束光照亮。Unity 的渲染管线使用了这种方法，我们也用同样的方法。

### 1.1. 阴影设置

在渲染阴影之前，我们首先要确定一些品质，特别是要渲染阴影的距离和阴影贴图的大小。

尽管我们可以对摄像机可见之处都渲染阴影，但这样会需要很多绘制和一张非常大的贴图来充分覆盖整个区域，这几乎是不切实际的。所以我们采用一个阴影的最大距离，最小值为 0 并且默认最大值设为 100。创建一个新的可序列化的 `ShadowSettings` 类来包含这个选项。这个类是单纯的配置选项的容器，所以我们使用 `public maxDistance` 字段。

```

1. using UnityEngine;
2.
3. [System.Serializable]
4. public class ShadowSettings {
5.
6.     [Min(0f)]
7.     public float maxDistance = 100f;
8. }

```

对于贴图尺寸，我们采用一个嵌套在 ShadowSettings 内部的 TextureSize 枚举类。用它来定义允许使用的纹理尺寸，256-8192 之间所有 2 的次方。

```

1. public enum TextureSize {
2.     _256 = 256, _512 = 512, _1024 = 1024,
3.     _2048 = 2048, _4096 = 4096, _8192 = 8192
4. }

```

然后给阴影贴图添加一个尺寸字段，用 1024 作为默认值。我们将使用一张单独的纹理来包含多张阴影贴图，所以将其命名为 atlasSize。因为现在只支持定向光，所以目前我们专门处理定向阴影。但将来我们会支持其他光类型，那时将用它们各自的阴影设置。所以将 atlasSize 放在内部 Directional 结构中。这样我们就在检视面板里自动获得了一个带有层级关系的配置项。

```

1. [System.Serializable]
2. public struct Directional {
3.
4.     public TextureSize atlasSize;
5. }
6.
7. public Directional directional = new Directional {
8.     atlasSize = TextureSize._1024
9. };

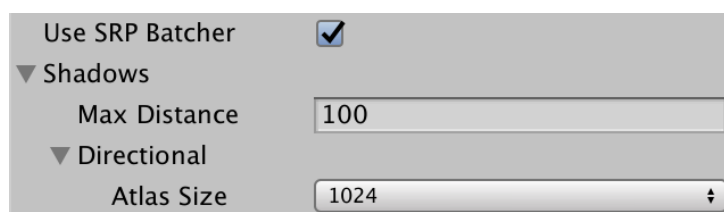
```

在 CustomRenderPipelineAsset 里添加一个阴影设置的字段。

```

1. [SerializeField]
2. ShadowSettings shadows = default;

```



阴影设置

当 CustomRenderPipeline 构造实例时，将这些设置传给它。

```
1. protected override RenderPipeline CreatePipeline () {  
2.     return new CustomRenderPipeline(  
3.         useDynamicBatching, useGPUInstancing, useSRPBatcher, shadows  
4.     );  
5. }
```

然后对它们保持追踪。

```
1. ShadowSettings shadowSettings;  
2.  
3. public CustomRenderPipeline (  
4.     bool useDynamicBatching, bool useGPUInstancing, bool useSRPBatcher,  
5.     ShadowSettings shadowSettings  
6. ) {  
7.     this.shadowSettings = shadowSettings;  
8.     ...  
9. }
```

## 1.2. 传递设置

从现在开始，当我们调用 Render 方法时，需要将这些设置传给摄像机渲染器。在框架里添加对改变阴影设置的支持很容易，但我们在这篇教程里不处理它。

```
1. protected override void Render (  
2.     ScriptableRenderContext context, Camera[] cameras  
3. ) {  
4.     foreach (Camera camera in cameras) {  
5.         renderer.Render(  
6.             context, camera, useDynamicBatching, useGPUInstancing,  
7.             shadowSettings  
8.         );  
9.     }  
10. }
```

然后 CameraRenderer.Render 把它传到 Lighting.Setup 和自己的 Cull 方法里。

```
1. public void Render (  
2.     ScriptableRenderContext context, Camera camera,  
3.     bool useDynamicBatching, bool useGPUInstancing,  
4.     ShadowSettings shadowSettings  
5. ) {
```

```

6.     ...
7.     if (!Cull(shadowSettings.maxDistance)) {
8.         return;
9.     }
10.
11.     Setup();
12.     lighting.Setup(context, cullingResults, shadowSettings);
13.     ...
14. }

```

我们在 Cull 中需要这些设置，因为阴影距离是通过剔除参数设置的。

```

1. bool Cull (float maxShadowDistance) {
2.     if (camera.TryGetCullingParameters(out ScriptableCullingParameters p)) {
3.         p.shadowDistance = maxShadowDistance;
4.         cullingResults = context.Cull(ref p);
5.         return true;
6.     }
7.     return false;
8. }

```

渲染超出摄像机观测范围的阴影是没有意义的，所以设置阴影距离为最大阴影距离和摄像机远裁剪面的最小值。

```

1. p.shadowDistance = Mathf.Min(maxShadowDistance, camera.farClipPlane);

```

为了让代码编译通过，我们还需要在 Lighting.Setup 为阴影设置添加一个参数，但我们暂时还没有对它们进行任何操作。

```

1. public void Setup (
2.     ScriptableRenderContext context, CullingResults cullingResults,
3.     ShadowSettings shadowSettings
4. ) { ... }

```

## 1.3. 阴影类

尽管逻辑上说阴影是光照的一部分，但它们更复杂，所以我们专门创建一个新的 Shadow 类。它以 Lighting 精简后的副本开始，包含自己的缓冲区，上下文、剔除结果和设置字段，用于初始化这些字段的 Setup 方法和一个 ExecuteBuffer 方法。

```

1. using UnityEngine;
2. using UnityEngine.Rendering;
3.

```

```

4. public class Shadows {
5.
6.     const string bufferName = "Shadows";
7.
8.     CommandBuffer buffer = new CommandBuffer {
9.         name = bufferName
10.    };
11.
12.    ScriptableRenderContext context;
13.
14.    CullingResults cullingResults;
15.
16.    ShadowSettings settings;
17.
18.    public void Setup (
19.        ScriptableRenderContext context, CullingResults cullingResults,
20.        ShadowSettings settings
21.    ) {
22.        this.context = context;
23.        this.cullingResults = cullingResults;
24.        this.settings = settings;
25.    }
26.
27.    void ExecuteBuffer () {
28.        context.ExecuteCommandBuffer(buffer);
29.        buffer.Clear();
30.    }
31. }

```

然后所有的 Lighting 都需要记录 Shadows 实例，并在自己的 Setup 方法的 SetupLights 之前调用阴影的 Setup 方法。

```

1. Shadows shadows = new Shadows();
2.
3. public void Setup (...) {
4.     this.cullingResults = cullingResults;
5.     buffer.BeginSample(bufferName);
6.     shadows.Setup(context, cullingResults, shadowSettings);
7.     SetupLights();
8.     ...
9. }

```

## 1.4. 支持阴影的光源

因为渲染阴影需要额外的工作, 它会降低帧率, 所以我们要限制可以支持阴影的定向光数量, 并且独立于定向光数量。在 Shadow 里添加一个常量, 初始设置为 1。

```
1. const int maxShadowedDirectionalLightCount = 1;
```

我们不知道那些可见光源会产生阴影, 所以我们必须进行记录。除此之外, 我们之后还需要对每个产生阴影的光源记录更多的数据, 所以我们定义一个目前包含光源索引的内部 ShadowedDirectionalLight 结构, 并且声明一个数组。

```
1. struct ShadowedDirectionalLight {  
2.     public int visibleLightIndex;  
3. }  
4.  
5. ShadowedDirectionalLight[] ShadowedDirectionalLights =  
6.     new ShadowedDirectionalLight[maxShadowedDirectionalLightCount];
```

为了判断哪个光源会产生阴影, 我们要添加一个带有光源和可见光源索引作为参数的公共 ReserveDirectionalShadows 方法。它的工作是在阴影图集中为光源阴影贴图预留空间, 并存储渲染它们所需的信息。

```
1. public void ReserveDirectionalShadows (Light light, int visibleLightIndex){}
```

因为产生阴影的光源数量有限制, 我们必须记录已经保留了多少。在 Setup 里将数量重置为 0。然后在 ReserveDirectionalShadows 里检查我们是否还没有到达数量上限。如果有剩余空间, 那么就存储光源可见索引, 并累加计数。

```
1. int ShadowedDirectionalLightCount;  
2.  
3. ...  
4.  
5. public void Setup (...) {  
6.     ...  
7.     ShadowedDirectionalLightCount = 0;  
8. }  
9.  
10. public void ReserveDirectionalShadows (Light light, int visibleLightIndex) {  
11.     if (ShadowedDirectionalLightCount < maxShadowedDirectionalLightCount) {  
12.         ShadowedDirectionalLights[ShadowedDirectionalLightCount++] =  
13.             new ShadowedDirectionalLight {  
14.                 visibleLightIndex = visibleLightIndex  
15.             };  
16.     }
```

```
17. }
```

但阴影只保留给能够产生阴影的光源。如果光源的阴影模式是 none 或者阴影强度是 0，那么它将不会产生阴影，就应该被忽略。

```
1. if (  
2.     ShadowedDirectionalLightCount < maxShadowedDirectionalLightCount &&  
3.     light.shadows != LightShadows.None && light.shadowStrength > 0f  
4. ) { ... }
```

除此之外，有可能出现一个可见光源最终没有影响到任何投射阴影的对象，要么因为它们被配置成不产生阴影，要么光源之影响最大阴影距离之外的对象。我们可以通过调用剔除结果的 GetShadowCasterBounds 并传入一个可见光索引来进行检查。它还有表示边界的第二个输出参数（这个我们不需要），并且返回边界是否合理。如果不合理，那么这个光源就没有阴影需要渲染，该光源会被忽略。

```
1. if (  
2.     ShadowedDirectionalLightCount < maxShadowedDirectionalLightCount &&  
3.     light.shadows != LightShadows.None && light.shadowStrength > 0f &&  
4.     cullingResults.GetShadowCasterBounds(visibleLightIndex, out Bounds b)  
5. ) { ... }
```

现在我们可以 Lighting.SetupDirectionalLight 里保留阴影。

```
1. void SetupDirectionalLight (int index, ref VisibleLight visibleLight) {  
2.     dirLightColors[index] = visibleLight.finalColor;  
3.     dirLightDirections[index] = -  
4.         visibleLight.localToWorldMatrix.GetColumn(2);  
5.     shadows.ReserveDirectionalShadows(visibleLight.light, index);  
6. }
```

## 1.5. 创建阴影图集

在保留阴影之后，我们需要进行渲染。我们在 Lighting.Render 的 SetupLights 结束之后调用一个新的 Shadows.Render 方法。

```
1. shadows.Setup(context, cullingResults, shadowSettings);  
2. SetupLights();  
3. shadows.Render();
```

Shadows.Render 方法会将渲染定向阴影委托给另一个 RenderDirectionalShadows 方法，但只有当存在产生阴影的光源时才执行。

```
1. public void Render () {  
2.     if (ShadowedDirectionalLightCount > 0) {
```

```

3.         RenderDirectionalShadows();
4.     }
5. }
6.
7. void RenderDirectionalShadows () {}

```

创建阴影贴图通过将阴影投射对象绘制到一张纹理上来实现。我们使用 `_directionalShadowAtlas` 来引用定向阴影图集。从设置里以整数形式获取图集大小，然后调用命令缓冲区的 `GetTemporaryRT`，传入纹理 ID 作为参数，再加上它的像素宽度和高度。

```

1. static int dirShadowAtlasId = Shader.PropertyToID("_DirectionalShadowAtlas")
   ;
2.
3. ...
4.
5. void RenderDirectionalShadows () {
6.     int atlasSize = (int)settings.directional.atlasSize;
7.     buffer.GetTemporaryRT(dirShadowAtlasId, atlasSize, atlasSize);
8. }

```

这步声明了一个正方形的渲染纹理，但这样默认是个普通的 ARGB 纹理。我们需要对调用方法指定另外 3 个参数来获得一张阴影贴图。第一个是深度缓冲区位数。我们希望它精度尽可能高，所以用 32。第二个是过滤模式，我们使用默认的双线性过滤。第三个是渲染纹理类型，必须是 `RenderTextureFormat.ShadowMap`。这为我们提供了适合渲染阴影贴图的纹理，尽管具体的格式取决于目标平台。

```

1. buffer.GetTemporaryRT(
2.     dirShadowAtlasId, atlasSize, atlasSize,
3.     32, FilterMode.Bilinear, RenderTextureFormat.ShadowMap
4. );

```

当我们获得一张临时渲染纹理，我们还需要在用完之后释放。我们必须将其保留到完成摄像机渲染之后，然后我们可以通过调用缓冲区的 `ReleaseTemporaryRT` 方法传入纹理 ID，然后执行缓冲区来进行纹理释放。如果我们有产生阴影的定向光，就在新的公共 `Cleanup` 方法里进行释放。

```

1. public void Cleanup () {
2.     if (ShadowedDirectionalLightCount > 0) {
3.         buffer.ReleaseTemporaryRT(dirShadowAtlasId);
4.         ExecuteBuffer();
5.     }
6. }

```

同样，给 `Lighting` 添加一个公共 `Cleanup` 方法，将调用传递给 `Shadows`。



```

1. public void Cleanup () {
2.     shadows.Cleanup();
3. }

```

然后 CameraRenderer 可以在提交之前直接请求清理。

```

1. public void Render (...) {
2.     ...
3.     lighting.Cleanup();
4.     Submit();
5. }

```

在请求到渲染纹理之后，Shadows.Render 也必须通知 GPU 渲染到这张纹理，而不是摄像机的目标纹理。这通过调用缓冲区上的 SetRenderTarget，标识渲染纹理和如何加载与存储它的数据来完成。我们不关心初始化状态，因为我们会立刻清理，所以使用 RenderBufferLoadAction.DontCare。并且这张纹理的目的是用来包含阴影数据，所以我们需要使用 RenderBufferStoreAction.Store 作为第三个参数。

```

1. buffer.GetTemporaryRT(...);
2. buffer.SetRenderTarget(
3.     dirShadowAtlasId,
4.     RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
5. );

```

一旦完成，我们就可以像清理摄像机目标纹理一样的方法使用 ClearRenderTarget，在这种情况下只需要关注深度缓冲区。通过执行缓冲区来完成。如果你至少有一个产生阴影的定向光处于活动状态，那么你就可以在帧调试器里看到出现了阴影图集的清理操作。

```

1. buffer.SetRenderTarget(
2.     dirShadowAtlasId,
3.     RenderBufferLoadAction.DontCare, RenderBufferStoreAction.Store
4. );
5. buffer.ClearRenderTarget(true, false, Color.clear);
6. ExecuteBuffer();

```

▼ Main Camera	6
Clear (Z+stencil)	
▼ Shadows	1
Clear (Z+stencil)	
► RenderLoop.Draw	2

清理两个渲染对象

## 1.6. 先处理阴影

因为在设置阴影图集之前设置了常规摄像机, 我们最终在渲染通常几何体之前切换到了阴影图集就, 这不是我们想要的。我们应在 CameraRenderer.Render 调用 CameraRenderer.Setup 之前渲染阴影, 常规渲染就不会被影响。

```
1. //Setup();
2. lighting.Setup(context, cullingResults, shadowSettings);
3. Setup();
4. DrawVisibleGeometry(useDynamicBatching, useGPUInstancing);
```

▼ Shadows	1
Clear (Z+stencil)	
▼ Main Camera	5
Clear (Z+stencil)	
▶ RenderLoop.Draw	2

*先处理阴影*

我们可以通过在设置光照之前开始采样, 并在之后立刻结束采样的方式来让阴影入口在帧调试器中嵌套在摄像机流程内。

```
1. buffer.BeginSample(SampleName);
2. ExecuteBuffer();
3. lighting.Setup(context, cullingResults, shadowSettings);
4. buffer.EndSample(SampleName);
5. Setup();
```

▼ Main Camera	6
▼ Shadows	1
Clear (Z+stencil)	
Clear (Z+stencil)	
▶ RenderLoop.Draw	2

*嵌套阴影*

## 1.7. 渲染

为了渲染单个光源的阴影, 我们要在 Shadow 添加一个带有两个参数的 RenderDirectionalShadows 方法的变体: 第一个是产生阴影的光源索引, 第二个是图集集中的平铺尺寸。然后在另一个 RenderDirectionalShadows 方法里对所有阴影光源调用这个方法, 并用 BeginSample 和 EndSample 进行调用包装。因为目前我们只支持单个阴影光源, 所以平铺尺寸等于图集尺寸。

```

1. void RenderDirectionalShadows () {
2.     ...
3.     buffer.ClearRenderTarget(true, false, Color.clear);
4.     buffer.BeginSample(bufferName);
5.     ExecuteBuffer();
6.
7.     for (int i = 0; i < ShadowedDirectionalLightCount; i++) {
8.         RenderDirectionalShadows(i, atlasSize);
9.     }
10.
11.    buffer.EndSample(bufferName);
12.    ExecuteBuffer();
13. }
14.
15. void RenderDirectionalShadows (int index, int tileSize) {}

```

为了渲染阴影, 我们需要一个 ShadowDrawingSettings 结构值。我们可以通过调用构造函数, 并传入剔除结果和之前储存的适当的可见光索引来创建一个正确配置的对象。

```

1. void RenderDirectionalShadows (int index, int tileSize) {
2.     ShadowedDirectionalLight light = ShadowedDirectionalLights[index];
3.     var shadowSettings =
4.         new ShadowDrawingSettings(cullingResults, light.visibleLightIndex);
5. }

```

阴影贴图的做法是从光源所在点的视角渲染场景, 然后只存储深度信息。这个结果告诉我们光线在接触对象之前行进了多远。

然而, 定向光被假设在无穷远处, 所以没有一个真实的位置。所以我们要做的是处理视图和投影矩阵, 使其匹配光源的朝向, 并且获得一个覆盖摄像机可见区域的裁剪空间立方体, 该区域包含了光源的阴影。与其自己处理, 我们可以使用剔除结果中所带的 ComputeDirectionalShadowMatricesAndCullingPrimitives 方法并传入九个参数来进行处理。

第一个参数是可见光索引。接下来三个是两个整数和一个 Vector3, 用来控制阴影级联。我们稍后再处理级联, 所以现在使用 0, 1 和零向量。之后是纹理尺寸, 这里我们需要使用平铺尺寸。第六个参数是阴影近平面, 现在我们忽略它并设置为 0。

上面这些是输入参数, 剩下的三个是输出参数。第一个是视图矩阵, 然后是投影矩阵, 最后是一个 ShadowSplitData 结构。

```

1. var shadowSettings =
2.     new ShadowDrawingSettings(cullingResults, light.visibleLightIndex);
3. cullingResults.ComputeDirectionalShadowMatricesAndCullingPrimitives(
4.     light.visibleLightIndex, 0, 1, Vector3.zero, tileSize, 0f,

```

```

5.         out Matrix4x4 viewMatrix, out Matrix4x4 projectionMatrix,
6.         out ShadowSplitData splitData
7.     );

```

ShadowSplitData 包含了如何剔除阴影对象的信息，我们必须将其复制到阴影设置里。并且我们要调用缓冲区的 SetViewProjectionMatrices 来应用视图和投影矩阵。

```

1. cullingResults.ComputeDirectionalShadowMatricesAndCullingPrimitives(...);
2. shadowSettings.splitData = splitData;
3. buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

```

最终我们通过执行缓冲区，然后在上下文中调用 DrawShadows 并传入阴影设置的引用，来执行阴影投射的绘制。

```

1. shadowSettings.splitData = splitData;
2. buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
3. ExecuteBuffer();
4. context.DrawShadows(ref shadowSettings);

```

## 1.8. 阴影投射 Pass

现在阴影投射应当被渲染，但图集还是空的。这是因为 DrawShadows 只会渲染使用包含 ShadowCaster Pass 材质的对象。所以在我们的 Lit 着色器里添加第二个 Pass，将其 LightMode 设置成 ShadowCaster。使用相同的目标等级，让它支持实例化，再加上\_CLIPING 着色器功能。然后让它使用特殊的阴影投射函数，我们将其定义在新的 ShadowCasterPass.hlsl 文件里。另外，因为哦我们只需要写入深度值，所以通过在 HLSLPROGRAM 之前添加 ColorMask 0 来禁用写入颜色数据。

```

1. SubShader {
2.     Pass {
3.         Tags {
4.             "LightMode" = "CustomLit"
5.         }
6.
7.         ...
8.     }
9.
10.    Pass {
11.        Tags {
12.            "LightMode" = "ShadowCaster"
13.        }
14.
15.        ColorMask 0
16.

```

```

17.         HLSLPROGRAM
18.         #pragma target 3.5
19.         #pragma shader_feature _CLIPPING
20.         #pragma multi_compile_instancing
21.         #pragma vertex ShadowCasterPassVertex
22.         #pragma fragment ShadowCasterPassFragment
23.         #include "ShadowCasterPass.hlsl"
24.         ENDHLSL
25.     }
26. }

```

复制 LitPass, 创建 ShadowCasterPass, 然后移除所有与阴影投射无关的内容。所以我们只需要裁剪空间位置, 加上裁剪基础颜色。片元函数不返回任何值, 所以变成没有语义的 void。它唯一的作用就是可能需要的裁剪片元。

```

1. #ifndef CUSTOM_SHADOW_CASTER_PASS_INCLUDED
2. #define CUSTOM_SHADOW_CASTER_PASS_INCLUDED
3.
4. #include "../ShaderLibrary/Common.hlsl"
5.
6. TEXTURE2D(_BaseMap);
7. SAMPLER(sampler_BaseMap);
8.
9. UNITY_INSTANCING_BUFFER_START(UnityPerMaterial)
10.     UNITY_DEFINE_INSTANCED_PROP(float4, _BaseMap_ST)
11.     UNITY_DEFINE_INSTANCED_PROP(float4, _BaseColor)
12.     UNITY_DEFINE_INSTANCED_PROP(float, _Cutoff)
13. UNITY_INSTANCING_BUFFER_END(UnityPerMaterial)
14.
15. struct Attributes {
16.     float3 positionOS : POSITION;
17.     float2 baseUV : TEXCOORD0;
18.     UNITY_VERTEX_INPUT_INSTANCE_ID
19. };
20.
21. struct Varyings {
22.     float4 positionCS : SV_POSITION;
23.     float2 baseUV : VAR_BASE_UV;
24.     UNITY_VERTEX_INPUT_INSTANCE_ID
25. };
26.
27. Varyings ShadowCasterPassVertex (Attributes input) {
28.     Varyings output;
29.     UNITY_SETUP_INSTANCE_ID(input);
30.     UNITY_TRANSFER_INSTANCE_ID(input, output);

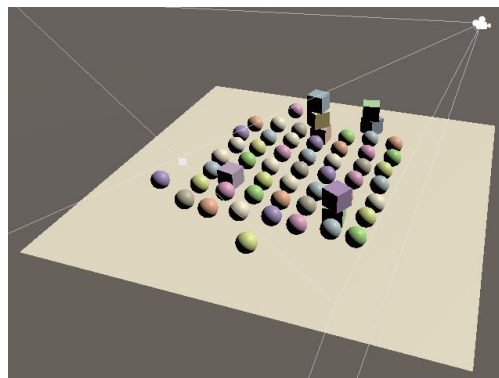
```

```

31.     float3 positionWS = TransformObjectToWorld(input.positionOS);
32.     output.positionCS = TransformWorldToHClip(positionWS);
33.
34.     float4 baseST = UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseMap_S
        T);
35.     output.baseUV = input.baseUV * baseST.xy + baseST.zw;
36.     return output;
37. }
38.
39. void ShadowCasterPassFragment (Varyings input) {
40.     UNITY_SETUP_INSTANCE_ID(input);
41.     float4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, input.baseU
        V);
42.     float4 baseColor = UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _BaseCo
        lor);
43.     float4 base = baseMap * baseColor;
44.     #if defined(_CLIPPING)
45.         clip(base.a - UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Cutoff)
        );
46.     #endif
47. }
48.
49. #endif

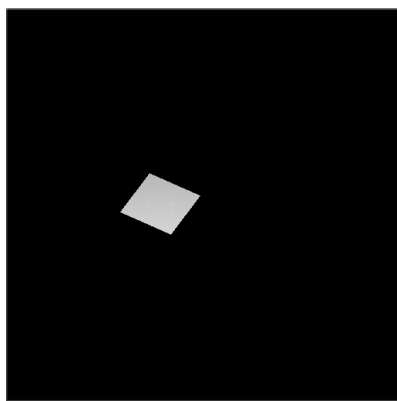
```

现在我们可以渲染阴影投射了。我创建了一个简单的测试场景，在平面上包含一些不透明对象，并带有一个开启了阴影的完整强度的定向光，用来测试阴影投射。光源设置影阴影或者软阴影没有影响。



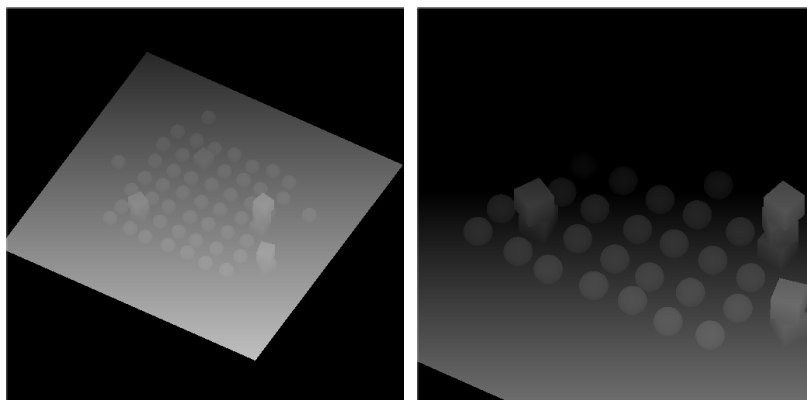
阴影测试场景

阴影还没有影响最后渲染的图片，但是我们已经可以在帧调试器里看到阴影图集上渲染了什么。它通常显示为一张随着距离增加而从白色到黑色的单色纹理，但当使用 OpenGL 时会显示为红色和相反的渐变方式。



512 图集尺寸; 最大距离 100

由于最大距离设置为 100，我们最终只把所有对象渲染到了纹理的一小部分。有效地减少最大距离可以让摄像机前方的纹理贴图放大。



最大距离分别为 20 和 10

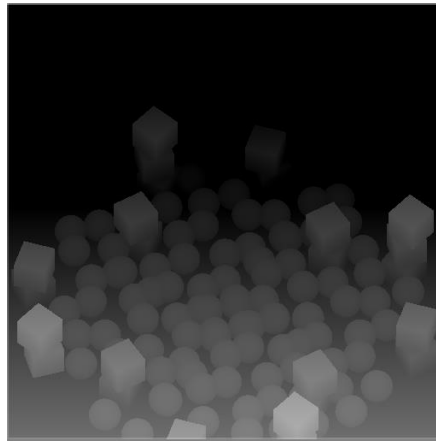
需要注意的是投射阴影是使用正交投影渲染的，因为我们在渲染定向光。

## 1.9. 多光源

我们可以有最多 4 个定向光，所欲 i 让我们也支持四个阴影定向光。

```
1. const int maxShadowedDirectionalLightCount = 4;
```

作为快速测试，我是有了四个相等的定向光，只是将它们的 Y 旋转增加了 90°。



四个光源的阴影投射叠加

尽管我们最终正确渲染了所有光源的阴影投射，但它们叠加在一起，因为我们把每个光源都渲染到了整张图集。我们需要拆分图集，以便为每个光源提供各自的图块用来渲染。

我们最多支持四个阴影光，并在图集中给每个光分配一个方形图块。所以如果我们最终有超过一个阴影光，我们必须减半图块尺寸，将图集拆分成四个图块。在 `Shadows.RenderDirectionalShadows` 里决定拆分数量和尺寸，并将它们传给每个光的另一个方法。

```
1. void RenderDirectionalShadows () {  
2.     ...  
3.  
4.     int split = ShadowedDirectionalLightCount <= 1 ? 1 : 2;  
5.     int tileSize = atlasSize / split;  
6.  
7.     for (int i = 0; i < ShadowedDirectionalLightCount; i++) {  
8.         RenderDirectionalShadows(i, split, tileSize);  
9.     }  
10. }  
11.  
12. void RenderDirectionalShadows (int index, int split, int tileSize) { ... }
```

我们可以通过调整渲染视口将其渲染到单个图块。为此创建一个带有 `index` 和 `split` 作为参数的新方法。首先计算图集偏移，将用 `split` 取模的 `index` 作为 X 偏移，`index` 除以 `split` 作为 Y 偏移。这些事证书操作，但我们最终定义一个矩形，所以将结果存为 `Vector2`。

```
1. void SetTileViewport (int index, int split) {  
2.     Vector2 offset = new Vector2(index % split, index / split);  
3. }
```

然后调用缓冲区的 `SetViewport`，传入一个矩形，其偏移量经过图块尺寸缩放，图块尺寸作为第三个浮点数参数。



```

1. void SetTileViewport (int index, int split, float tileSize) {
2.     Vector2 offset = new Vector2(index % split, index / split);
3.     buffer.SetViewport(new Rect(
4.         offset.x * tileSize, offset.y * tileSize, tileSize, tileSize
5.     ));
6. }

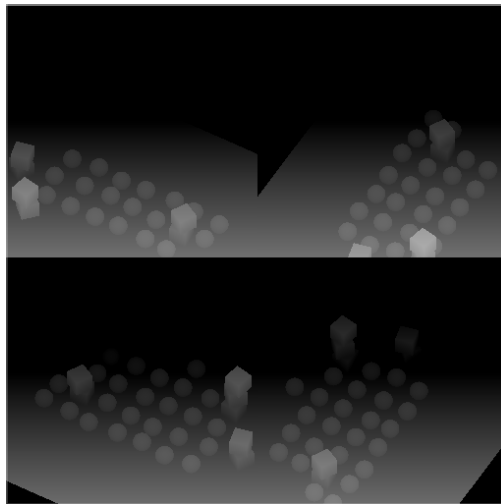
```

当设置矩阵时，在 RenderDirectionalShadows 里调用 SetTileViewport。

```

1. SetTileViewport(index, split, tileSize);
2. buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);

```



四个图块的阴影图集

## 2. 阴影采样

现在我们渲染了阴影投射，但还没影响最后的图片。为了显示阴影，我们必须在 CustomLit Pass 里对阴影贴图采样，以确定一个表面片元是否在阴影中。

### 2.1. 阴影矩阵

我们需要为每一个片元从阴影图集里合适的图块上采样深度信息。所以我们要为一个给定的世界空间位置找到对应的阴影纹理坐标。我们通过为每个阴影定向光创建一个阴影变换矩阵，并将它们传给 GPU 来实现。在 Shadows 添加一个 \_DirectionalShadowMatrices 着色器属性 ID 和静态矩阵数组。

```

1. static int
2.     dirShadowAtlasId = Shader.PropertyToID("_DirectionalShadowAtlas"),

```

```

3.     dirShadowMatricesId = Shader.PropertyToID("_DirectionalShadowMatrices");
4.
5.     static Matrix4x4[]
6.     dirShadowMatrices = new Matrix4x4[maxShadowedDirectionalLightCount];

```

我们可以通过将 `RenderDirectionalShadows` 里的阴影投影矩阵和视图矩阵相乘来获得一个从世界空间转换到光源空间的变换矩阵。

```

1. void RenderDirectionalShadows (int index, int split, int tileSize) {
2.     ...
3.     SetTileViewport(index, split, tileSize);
4.     dirShadowMatrices[index] = projectionMatrix * viewMatrix;
5.     buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
6.     ...
7. }

```

然后一旦所有的阴影光源渲染完成，就调用缓冲区的 `SetGlobalMatrixArray` 将矩阵上传到 GPU。

```

1. void RenderDirectionalShadows () {
2.     ...
3.
4.     buffer.SetGlobalMatrixArray(dirShadowMatricesId, dirShadowMatrices);
5.     buffer.EndSample(bufferName);
6.     ExecuteBuffer();
7. }

```

然而，这忽略了我们正在使用阴影图集。创建一个 `ConvertToAtlasMatrix`，传入光源矩阵，图块偏移，拆分数量，并返回从世界空间变换到阴影图块空间的矩阵。

```

1. Matrix4x4 ConvertToAtlasMatrix (Matrix4x4 m, Vector2 offset, int split) {
2.     return m;
3. }

```

我们已经在 `SetTileViewport` 里计算过图块偏移，所以将它作为返回值。

```

1. Vector2 SetTileViewport (int index, int split, float tileSize) {
2.     ...
3.     return offset;
4. }

```

然后调整 `RenderDirectionalShadows` 以便于调用 `ConvertToAtlasMatrix`。

```

1. //SetTileViewport(index, split, tileSize);

```

```

2. dirShadowMatrices[index] = ConvertToAtlasMatrix(
3.     projectionMatrix * viewMatrix,
4.     SetTileViewport(index, split, tileSize), split
5. );

```

我们在 ConvertToAtlasMatrix 里要做的第一件事是判断如果使用反向 Z Buffer，就应该将 Z 分量取负。我们可以通过 SystemInfo.usesReversedZBuffer 来检查。

```

1. Matrix4x4 ConvertToAtlasMatrix (Matrix4x4 m, Vector2 offset, int split) {
2.     if (SystemInfo.usesReversedZBuffer) {
3.         m.m20 = -m.m20;
4.         m.m21 = -m.m21;
5.         m.m22 = -m.m22;
6.         m.m23 = -m.m23;
7.     }
8.     return m;
9. }

```

其次，裁剪空间定义在一个坐标从-1 到 1 的立方体中，0 在空间中心。但是纹理坐标和深度值是从 0 到 1 的。我们通过将 XYZ 分量缩放和偏移一半来将这个变换烘焙到矩阵里。我们可以通过矩阵乘法来操作，但它会导致许多与 0 的乘法和不必要的加法。所以我们直接调整矩阵。

```

1. m.m00 = 0.5f * (m.m00 + m.m30);
2. m.m01 = 0.5f * (m.m01 + m.m31);
3. m.m02 = 0.5f * (m.m02 + m.m32);
4. m.m03 = 0.5f * (m.m03 + m.m33);
5. m.m10 = 0.5f * (m.m10 + m.m30);
6. m.m11 = 0.5f * (m.m11 + m.m31);
7. m.m12 = 0.5f * (m.m12 + m.m32);
8. m.m13 = 0.5f * (m.m13 + m.m33);
9. m.m20 = 0.5f * (m.m20 + m.m30);
10. m.m21 = 0.5f * (m.m21 + m.m31);
11. m.m22 = 0.5f * (m.m22 + m.m32);
12. m.m23 = 0.5f * (m.m23 + m.m33);
13. return m;

```

最后，我们需要应用图块偏移和缩放。再次直接操作矩阵来避免大量不必要的计算。

```

1. float scale = 1f / split;
2. m.m00 = (0.5f * (m.m00 + m.m30) + offset.x * m.m30) * scale;
3. m.m01 = (0.5f * (m.m01 + m.m31) + offset.x * m.m31) * scale;
4. m.m02 = (0.5f * (m.m02 + m.m32) + offset.x * m.m32) * scale;
5. m.m03 = (0.5f * (m.m03 + m.m33) + offset.x * m.m33) * scale;
6. m.m10 = (0.5f * (m.m10 + m.m30) + offset.y * m.m30) * scale;

```

```

7. m.m11 = (0.5f * (m.m11 + m.m31) + offset.y * m.m31) * scale;
8. m.m12 = (0.5f * (m.m12 + m.m32) + offset.y * m.m32) * scale;
9. m.m13 = (0.5f * (m.m13 + m.m33) + offset.y * m.m33) * scale;

```

## 2.2. 为每个光源存储阴影数据

为了对光源采样阴影，我们需要知道它在阴影图集的图块索引（如果存在）。有些内容必须逐光源存储，所以我们让 `ReserveDirectionalShadows` 返回所需的数据。我们提供两个值：阴影强度和阴影图块偏移，并将它们打包在 `Vector2` 里。如果光源不产生阴影那么结果是 `Vector2.zero`。

```

1. public Vector2 ReserveDirectionalShadows (...) {
2.     if (...) {
3.         ShadowedDirectionalLights[ShadowedDirectionalLightCount] =
4.             new ShadowedDirectionalLight {
5.                 visibleLightIndex = visibleLightIndex
6.             };
7.         return new Vector2(
8.             light.shadowStrength, ShadowedDirectionalLightCount++
9.         );
10.    }
11.    return Vector2.zero;
12. }

```

让 `Lighting` 通过 `_DirectionalLightShadowData` 向量数组把数据提供给着色器。

```

1. static int
2.     dirLightCountId = Shader.PropertyToID("_DirectionalLightCount"),
3.     dirLightColorsId = Shader.PropertyToID("_DirectionalLightColors"),
4.     dirLightDirectionsId = Shader.PropertyToID("_DirectionalLightDirections"
5.     ),
6.     dirLightShadowDataId =
7.         Shader.PropertyToID("_DirectionalLightShadowData");
8. static Vector4[]
9.     dirLightColors = new Vector4[maxDirLightCount],
10.    dirLightDirections = new Vector4[maxDirLightCount],
11.    dirLightShadowData = new Vector4[maxDirLightCount];
12.
13. ...
14.
15. void SetupLights () {

```

```

16.     ...
17.     buffer.SetGlobalVectorArray(dirLightShadowDataId, dirLightShadowData);
18. }
19.
20. void SetupDirectionalLight (int index, ref VisibleLight visibleLight) {
21.     dirLightColors[index] = visibleLight.finalColor;
22.     dirLightDirections[index] = -
        visibleLight.localToWorldMatrix.GetColumn(2);
23.     dirLightShadowData[index] =
24.         shadows.ReserveDirectionalShadows(visibleLight.light, index);
25. }

```

同样把它添加的 Light.hlsl 文件的 \_CustomLight 缓冲区里。

```

1. CBUFFER_START(_CustomLight)
2.     int _DirectionalLightCount;
3.     float4 _DirectionalLightColors[MAX_DIRECTIONAL_LIGHT_COUNT];
4.     float4 _DirectionalLightDirections[MAX_DIRECTIONAL_LIGHT_COUNT];
5.     float4 _DirectionalLightShadowData[MAX_DIRECTIONAL_LIGHT_COUNT];
6. CBUFFER_END

```

## 2.3. Shadows.hlsl 文件

我们同样要创建一个单独的 Shadows.hlsl 文件用于阴影采样。定义相同的最大阴影定向光数量，接下来是 \_DirectionalShadowAtlas 纹理，加上 \_CustomShadows 缓冲区里的 \_DirectionalShadowMatrices 数组。

```

1. #ifndef CUSTOM_SHADOWS_INCLUDED
2. #define CUSTOM_SHADOWS_INCLUDED
3.
4. #define MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT 4
5.
6. TEXTURE2D(_DirectionalShadowAtlas);
7. SAMPLER(sampler_DirectionalShadowAtlas);
8.
9. CBUFFER_START(_CustomShadows)
10.     float4x4 _DirectionalShadowMatrices[MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT
        ];
11. CBUFFER_END
12.
13. #endif

```

因为图集不是常规纹理，我们通过 TEXTURE2D\_SHADOW 宏来定义，尽管在我们支持的平台上没有差别。同时我们使用一个特殊的 SAMPLER\_CMP 宏来定义采样状态，这定义了一种

进行阴影贴图采样的不同方法，因为常规的双线性过滤对深度数据没有意义。

```
1. TEXTURE2D_SHADOW(_DirectionalShadowAtlas);
2. SAMPLER_CMP(sampler_DirectionalShadowAtlas);
```

实际上，只有一种合适的方法用来进行阴影贴图采样，所以我们可以定义一个显式的采样状态来代替 Unity 对我们的渲染纹理所推断的采样状态。采样状态可以通过创建带有关键词的名称来进行内联定义。我们可以使用 `sampler_linear_clamp_compare`。我们同样给它定义一个简写的 `SHADOW_SAMPLER` 宏。

```
1. TEXTURE2D_SHADOW(_DirectionalShadowAtlas);
2. #define SHADOW_SAMPLER sampler_linear_clamp_compare
3. SAMPLER_CMP(SHADOW_SAMPLER);
```

在 LitPass 的 Light 之前引入 Shadows。

```
1. #include "../ShaderLibrary/Surface.hlsl"
2. #include "../ShaderLibrary/Shadows.hlsl"
3. #include "../ShaderLibrary/Light.hlsl"
```

## 2.4. 阴影采样

为了对阴影进行采样，我们需要知道逐光源的阴影数据，所以在 Shadows 里特别为定向光定义一个结构。它包含强度和偏移，但 Shadows 里的代码不知道数据存在哪里。

```
1. struct DirectionalShadowData {
2.     float strength;
3.     int tileOffset;
4. };
```

同样，我们需要表面位置，所以添加到 Surface 结构里。

```
1. struct Surface {
2.     float3 position;
3.     ...
4. };
```

并且在 LitPassFragment 里赋值。

```
1. Surface surface;
2. surface.position = input.positionWS;
3. surface.normal = normalize(input.normalWS);
```

在 Shadows 里添加一个 `SampleDirectionalShadowAtlas` 方法用 `SAMPLE_TEXTURE_SHADOW` 宏进行阴影图集采样，传入图集，阴影采样器和阴影纹理空间的位置（对应参数）。

```

1. float SampleDirectionalShadowAtlas (float3 positionSTS) {
2.     return SAMPLE_TEXTURE2D_SHADOW(
3.         _DirectionalShadowAtlas, SHADOW_SAMPLER, positionSTS
4.     );
5. }

```

然后添加一个返回阴影衰减的 `GetDirectionalShadowAttenuation` 方法，传入定向阴影数据和定义在世界空间的表面。它使用图块偏移来获取正确的矩阵，将表面坐标转换到阴影图块空间，然后进行图集采样。

```

1. float GetDirectionalShadowAttenuation (DirectionalShadowData data, Surface surfaceWS) {
2.     float3 positionSTS = mul(
3.         _DirectionalShadowMatrices[data.tileIndex],
4.         float4(surfaceWS.position, 1.0)
5.     ).xyz;
6.     float shadow = SampleDirectionalShadowAtlas(positionSTS);
7.     return shadow;
8. }

```

采样阴影图集的结果是一个决定多少光线到达表面的系数，该系数只考虑阴影。作为衰减系数，它是个 0 到 1 之间的值。如果片元完全被阴影覆盖，我们将得到 0；如果完全没有被覆盖，我们得到 1。之间的值表示片元被阴影部分覆盖。

除此之外，光源的阴影强度也会因为美术原因或表现半透明表面的阴影而减少。当强度减少到 0，衰减将完全不会影响阴影，所以应当为 1。所以最终的衰减系数通过在 1 和采样衰减之间，基于阴影强度做线性插值。

```

1. return lerp(1.0, shadow, data.strength);

```

但当阴影强度为 0 时，就没有必要去进行阴影采样，因为不会产生影响，甚至不会被渲染。在这种情况下，我们拥有一个不会产生阴影的光源，并且始终返回 1。

```

1. float GetDirectionalShadowAttenuation (DirectionalShadowData data, Surface surfaceWS) {
2.     if (data.strength <= 0.0) {
3.         return 1.0;
4.     }
5.     ...
6. }

```

## 2.5. 衰减光

我们在 `Light` 结构里存储光源衰减。

```

1. struct Light {
2.     float3 color;
3.     float3 direction;
4.     float attenuation;
5. };

```

在 Light 里添加一个获取定向阴影数据的方法。

```

1. DirectionalShadowData GetDirectionalShadowData (int lightIndex) {
2.     DirectionalShadowData data;
3.     data.strength = _DirectionalLightShadowData[lightIndex].x;
4.     data.tileOffset = _DirectionalLightShadowData[lightIndex].y;
5.     return data;
6. }

```

然后向 GetDirectionalLight 添加一个世界空间的 surface 参数，用它和定向光阴影数据传入 GetDirectionalShadowAttenuation 来设置光源衰减。

```

1. Light GetDirectionalLight (int index, Surface surfaceWS) {
2.     Light light;
3.     light.color = _DirectionalLightColors[index].rgb;
4.     light.direction = _DirectionalLightDirections[index].xyz;
5.     DirectionalShadowData shadowData = GetDirectionalShadowData(index);
6.     light.attenuation = GetDirectionalShadowAttenuation(shadowData, surfaceWS);
7.     return light;
8. }

```

现在 Lighting 里的 GetLighting 也需要向 GetDirectionalLight 传入 surface。现在 surface 应该在世界空间内定义，所以相应地重命名参数。只有 BRDF 不关心光源和表面所在空间（只要它们相匹配）。

```

1. float3 GetLighting (Surface surfaceWS, BRDF brdf) {
2.     float3 color = 0.0;
3.     for (int i = 0; i < GetDirectionalLightCount(); i++) {
4.         color += GetLighting(surfaceWS, brdf, GetDirectionalLight(i, surfaceWS));
5.     }
6.     return color;
7. }

```

让阴影生效的最后一步是让光强叠加衰减系数。

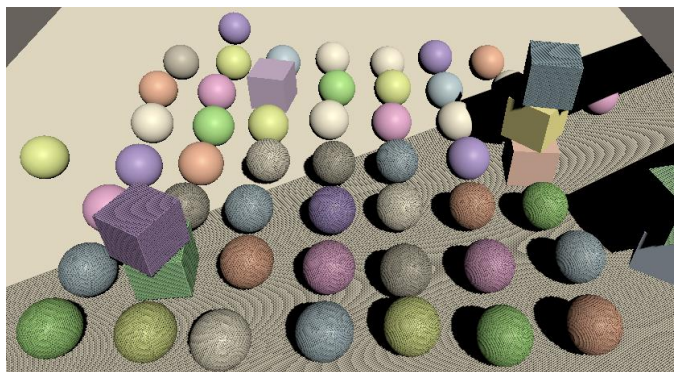
```

1. float3 IncomingLight (Surface surface, Light light) {
2.     return

```



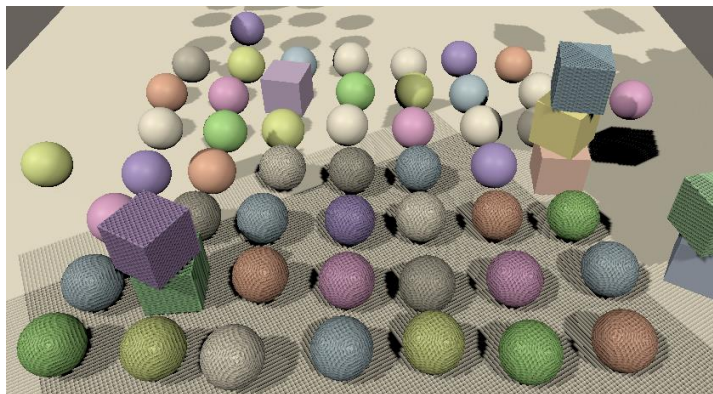
```
3.         saturate(dot(surface.normal, light.direction) * light.attenuation) *  
4.         light.color;  
5.     }
```



一个阴影光源：最大距离 10；图集尺寸 512

现在我们得到了阴影，但看上去很糟糕。不应当获得阴影的表面最终被像素带形成的阴影伪影所覆盖。这些是由自我遮蔽而引起，由阴影贴图的分辨率限制而导致。使用不同的分辨率会改变伪影的样式但不会消除它们。表面最终部分遮盖了自身，但我们稍后再处理这个问题。伪影使得我们易于观察阴影贴图覆盖的区域，所以我们目前保留它们。

例如，我们可以看到阴影贴图只覆盖了部分可见区域，这通过最大阴影距离控制。增加最大距离或者缩小区域。阴影贴图与光源方向对齐，而不是摄像机。有些阴影在最大距离以外依然可见，但有些不见了，并且当阴影采样超出贴图边缘时会变得很奇怪。只有当只激活单个阴影光源时，采样结果才会被限制区间，否则采样就会超出图块边界，同时一个光源最终会使用来自另一个光源的阴影。



两个一半强度的阴影光

我们稍后将正确地在最大距离截断阴影，但现在这些不合理的阴影依然可见。

### 3. 级联阴影贴图

因为定向光影响最大阴影距离内的所有对象，他们的阴影贴图最终覆盖了一大块区域。因为

阴影贴图使用正交投影，所以贴图上的每个纹素拥有固定的世界空间大小。如果这个尺寸过大，那么将会清楚地看到单独的阴影纹素，这导致阴影边缘出现锯齿，并且小阴影会消失。这可以通过增大图集尺寸来缓解，但只能到一定程度。

当使用透视摄像机时越远的对象越小。在某个可见的距离，一个阴影贴图纹素将匹配一个显示像素，意味着阴影分辨率是理论最佳值。离摄像机越近我们需要越高的阴影分辨率，同时越远意味着越低的分辨率就可以满足要求。这表示理想情况下，我们将根据阴影接收器的视距使用动态阴影贴图分辨率。

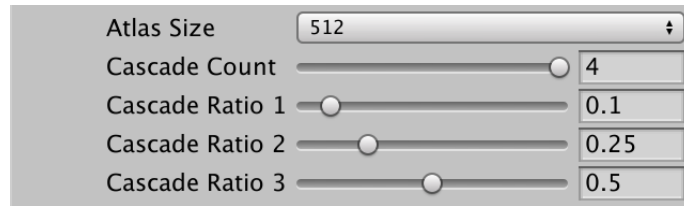
级联阴影贴图（CSM）是一个该问题的解决方案。方法是阴影投射渲染不止一次，所以每一个光源在图集里拥有多个图块，这被称作级联。第一张级联只覆盖靠近摄像机的一小块区域，逐渐缩小连续的级联，用相同的纹素数量覆盖越来越大块的区域。然后着色器为每个片元进行最合适的级联采样。

## 3.1. 设置

Unity 的阴影代码对每个定向光支持最多四层级联。目前我们只用了一张单独的级联用来覆盖最大距离以内的所有对象。为了支持更多级联，我们要在定向阴影设置里添加一个级联数量滑块。相比于对每个定向光使用不同的数量，对所有阴影定向光使用相同的数量更有意义。

每个级联覆盖一部分阴影区域，直到最大阴影距离。我们通过为前三个级联添加比例滑块来使得可配置精确的区域。最后一个级联始终覆盖整个范围，所以不需要滑块。将级联数量默认值设为 4，级联比例分别是 0.1, 0.25 和 0.5。这些比率应该逐级联增加，但我们在 UI 中不做限制。

```
1. public struct Directional {
2.
3.     public MapSize atlasSize;
4.
5.     [Range(1, 4)]
6.     public int cascadeCount;
7.
8.     [Range(0f, 1f)]
9.     public float cascadeRatio1, cascadeRatio2, cascadeRatio3;
10. }
11.
12. public Directional directional = new Directional {
13.     atlasSize = MapSize._1024,
14.     cascadeCount = 4,
15.     cascadeRatio1 = 0.1f,
16.     cascadeRatio2 = 0.25f,
17.     cascadeRatio3 = 0.5f
18. };
```



级联数量和比例

ComputeDirectionalShadowMatricesAndCullingPrimitives 需要我们将比率打包在 Vector3 中提供，所以我们给 settings 添加一个方便的属性用来获取对应格式的数据。

```
1. public Vector3 CascadeRatios =>
2.     new Vector3(cascadeRatio1, cascadeRatio2, cascadeRatio3);
```

## 3.2. 渲染级联图

每张级联图都需要自己的变换矩阵，所以 Shadows 里阴影矩阵数组的长度必须乘以每个光源级联的最大数量 4。

```
1. const int maxShadowedDirectionalLightCount = 4, maxCascades = 4;
2.
3. ...
4.
5. static Matrix4x4[]
6.     dirShadowMatrices = new Matrix4x4[maxShadowedDirectionalLightCount * max
    Cascades];
```

同样在 Shadows.hlsl 里增加数组长度。

```
1. #define MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT 4
2. #define MAX_CASCADE_COUNT 4
3.
4. ...
5.
6. CBUFFER_START(_CustomShadows)
7.     float4x4 _DirectionalShadowMatrices
8.         [MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT * MAX_CASCADE_COUNT];
9. CBUFFER_END
```

做完之后，Unity 会报出着色器数组长度变化，但无法使用新长度的错误。这是因为一旦着色器声明了固定长度的数组，在同一会话期间无法改变 GPU 上的数组大小。我们必须重启 Unity 来重新初始化。

然后将 Shadows.ReserveDirectionalShadows 返回的偏移量乘以配置的级联数量，因为每个定向光将声明多个连续的平铺。

```

1. return new Vector2(
2.     light.shadowStrength,
3.     settings.directional.cascadeCount * ShadowedDirectionalLightCount++
4. );

```

同样地，在 `RenderDirectionalShadow` 里使用的平铺数量也成倍增加，意味着最终有 16 个图块，拆分为 4 部分。

```

1. int tiles = ShadowedDirectionalLightCount * settings.directional.cascadeCount;
2. int split = tiles <= 1 ? 1 : tiles <= 4 ? 2 : 4;
3. int tileSize = atlasSize / split;

```

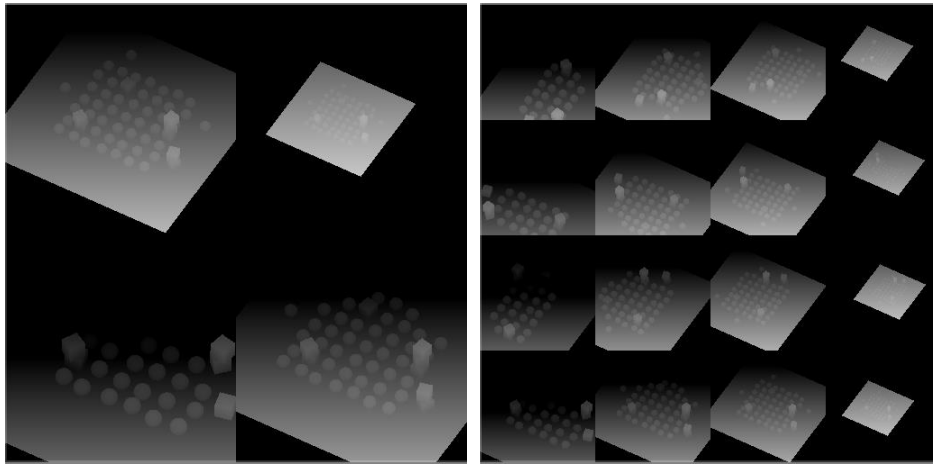
现在 `RenderDirectionalAShadows` 需要为每张级联绘制阴影。针对每个配置的级联，将从 `ComputeDirectionalShadowMatricesAndCullingPrimitives` 开始并包含 `DrawShadows` 的代码放入循环。`ComputeDirectionalShadowMatricesAndCullingPrimitives` 的第二个参数是级联索引，后面跟着级联数量和级联比率。同样调整 `tileIndex`，使其变成光源的 `tileOffset` 加上级联索引。

```

1. void RenderDirectionalShadows (int index, int split, int tileSize) {
2.     ShadowedDirectionalLight light = shadowedDirectionalLights[index];
3.     var shadowSettings =
4.         new ShadowDrawingSettings(cullingResults, light.visibleLightIndex);
5.     int cascadeCount = settings.directional.cascadeCount;
6.     int tileOffset = index * cascadeCount;
7.     Vector3 ratios = settings.directional.CascadeRatios;
8.
9.     for (int i = 0; i < cascadeCount; i++) {
10.         cullingResults.ComputeDirectionalShadowMatricesAndCullingPrimitives(
11.             light.visibleLightIndex, i, cascadeCount, ratios, tileSize, 0f,
12.             out Matrix4x4 viewMatrix, out Matrix4x4 projectionMatrix,
13.             out ShadowSplitData splitData
14.         );
15.         shadowSettings.splitData = splitData;
16.         int tileIndex = tileOffset + i;
17.         dirShadowMatrices[tileIndex] = ConvertToAtlasMatrix(
18.             projectionMatrix * viewMatrix,
19.             SetTileViewport(tileIndex, split, tileSize), split
20.         );
21.         buffer.SetViewProjectionMatrices(viewMatrix, projectionMatrix);
22.         ExecuteBuffer();
23.         context.DrawShadows(ref shadowSettings);

```

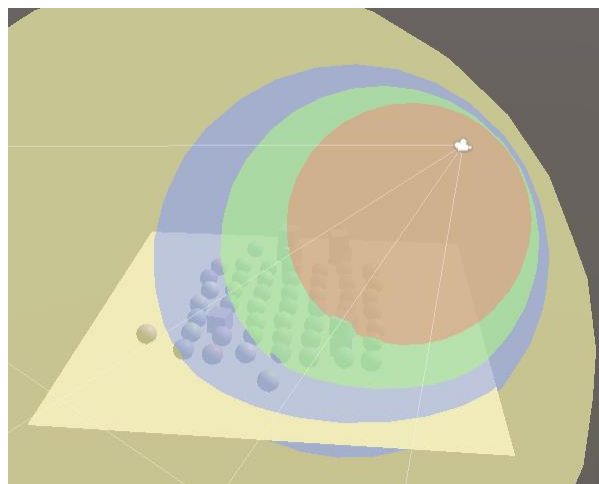
```
24.     }  
25. }
```



带有 4 个级联的单光源和四光源；最大距离 30；比率 0.3, 0.4, 0.5

### 3.3. 剔除球

Unity 通过创建一个剔除球来确定每个级联的覆盖区域。因为阴影投影是正交和方形的，他们最终几乎匹配他们的剔除球，但也覆盖了周围的一些区域。这就是为什么有些阴影会在剔除区域以外出现。同样地，光源方向不影响球体，所以所有的定向光最终使用同样地剔除球。



用透明球将剔除球可视化

这些球体同样需要决定从哪个级联中采样，所以我们必须将他们上传到 GPU。为级联数量和级联剔除球数组个添加一个 ID，再为球体数据添加一个数组。它们由四维向量定义，包括 XYZ 坐标和在 W 分量里的半径。

```
1. static int  
2.     dirShadowAtlasId = Shader.PropertyToID("_DirectionalShadowAtlas"),
```

```

3.     dirShadowMatricesId = Shader.PropertyToID("_DirectionalShadowMatrices"),
4.     cascadeCountId = Shader.PropertyToID("_CascadeCount"),
5.     cascadeCullingSpheresId = Shader.PropertyToID("_CascadeCullingSpheres");
6.
7.     static Vector4[] cascadeCullingSpheres = new Vector4[maxCascades];

```

级联剔除球是 ComputeDirectionalShadowMatricesAndCullingPrimitives 输出的 splitData 的一部分。在 RenderDirectionalShadows 的循环里将其分配给球体数组。但我们只需要对第一个光源进行这种操作，因为所有光源的级联都是相同的。

```

1. for (int i = 0; i < cascadeCount; i++) {
2.     cullingResults.ComputeDirectionalShadowMatricesAndCullingPrimitives(...);
3.     shadowSettings.splitData = splitData;
4.     if (index == 0) {
5.         cascadeCullingSpheres[i] = splitData.cullingSphere;
6.     }
7.     ...
8. }

```

我们需要用着色器里的球体来检测表面片元是否在球体之内，这可以通过比较与球心距离的平方和半径平方的大小来实现。所以我们改为存储半径的平方，以便不需要在着色器里进行计算。

```

1. Vector4 cullingSphere = splitData.cullingSphere;
2. cullingSphere.w *= cullingSphere.w;
3. cascadeCullingSpheres[i] = cullingSphere;

```

在渲染级联之后将级联数量和球体传给 GPU。

```

1. void RenderDirectionalShadows () {
2.     ...
3.
4.     buffer.SetGlobalInt(cascadeCountId, settings.directional.cascadeCount);
5.     buffer.SetGlobalVectorArray(
6.         cascadeCullingSpheresId, cascadeCullingSpheres
7.     );
8.     buffer.SetGlobalMatrixArray(dirShadowMatricesId, dirShadowMatrices);
9.     buffer.EndSample(bufferName);
10.    ExecuteBuffer();
11. }

```

## 3.4. 采样级联图

将级联数量和剔除球数组添加到 Shadows 里。

```
1. CBUFFER_START(_CustomShadows)
2.     int _CascadeCount;
3.     float4 _CascadeCullingSpheres[MAX_CASCADE_COUNT];
4.     float4x4 _DirectionalShadowMatrices
5.         [MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT * MAX_CASCADE_COUNT];
6. CBUFFER_END
```

级联索引是逐片元而非逐光源决定，所以我们采用全局 ShadowData 结构来包含该数据。稍后我们会添加更多的数据。同样添加一个 GetShadowData 方法返回对于世界空间表面的 ShadowData，将级联索引始终初始化为 0。

```
1. struct ShadowData {
2.     int cascadeIndex;
3. };
4.
5. ShadowData GetShadowData (Surface surfaceWS) {
6.     ShadowData data;
7.     data.cascadeIndex = 0;
8.     return data;
9. }
```

将新数据作为参数传给 GetDirectionalShadowData，以便它可以通过将光源的阴影图块偏移加上级联索引来选择正确的图块索引。

```
1. DirectionalShadowData GetDirectionalShadowData (
2.     int lightIndex, ShadowData shadowData
3. ) {
4.     DirectionalShadowData data;
5.     data.strength = _DirectionalLightShadowData[lightIndex].x;
6.     data.tileIndex =
7.         _DirectionalLightShadowData[lightIndex].y + shadowData.cascadeIndex;
8.     return data;
9. }
```

将相同的参数也添加给 GetDirectionalLight，以便科技将数据传给 GetDirecionalShadowData。将定向阴影数据重命名为合适的变量。

```
1. Light GetDirectionalLight (int index, Surface surfaceWS, ShadowData shadowDa
   ta) {
```



```

2.     ...
3.     DirectionalShadowData dirShadowData =
4.         GetDirectionalShadowData(index, shadowData);
5.     light.attenuation = GetDirectionalShadowAttenuation(dirShadowData, surfaceWS);
6.     return light;
7. }

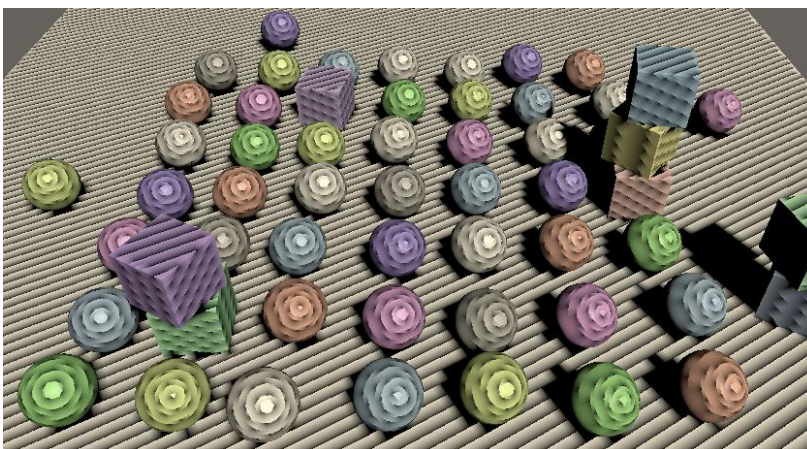
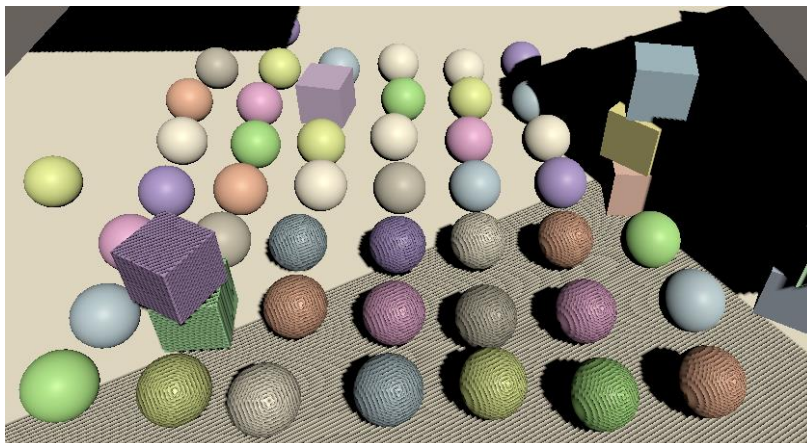
```

在 GetLighting 里获取阴影数据，并将其传入。

```

1. float3 GetLighting (Surface surfaceWS, BRDF brdf) {
2.     ShadowData shadowData = GetShadowData(surfaceWS);
3.     float3 color = 0.0;
4.     for (int i = 0; i < GetDirectionalLightCount(); i++) {
5.         Light light = GetDirectionalLight(i, surfaceWS, shadowData);
6.         color += GetLighting(surfaceWS, brdf, light);
7.     }
8.     return color;
9. }

```



始终使用第一个级联和最后一个级联

选择我们所需要的正确级联来计算两点间的距离平方。我们在 Common 里添加一个方便的

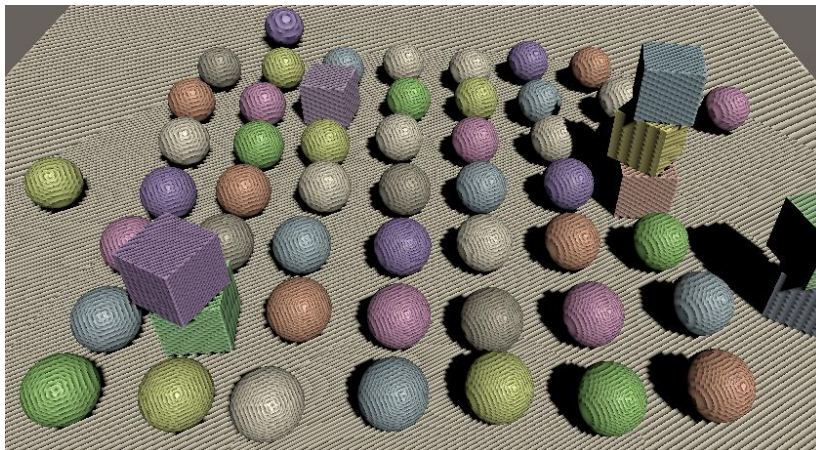


方法来操作。

```
1. float DistanceSquared(float3 pA, float3 pB) {
2.     return dot(pA - pB, pA - pB);
3. }
```

在 GetShadowData 里对所有级联剔除球进行循环，知道找到一个包含当前表面位置的剔除球。一旦找到就跳出循环，然后使用当前的循环索引作为级联索引。这意味着如果片元落在所有球体之外，我们最终会获得一个不合理的索引，但现在我们先忽略。

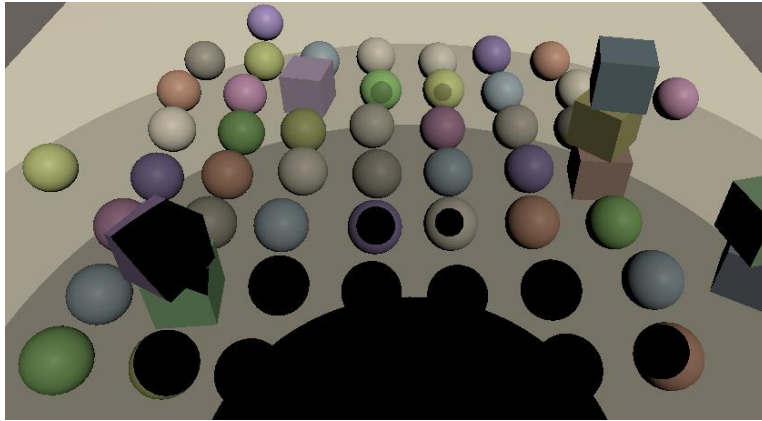
```
1. int i;
2. for (i = 0; i < _CascadeCount; i++) {
3.     float4 sphere = _CascadeCullingSpheres[i];
4.     float distanceSqr = DistanceSquared(surfaceWS.position, sphere.xyz);
5.     if (distanceSqr < sphere.w) {
6.         break;
7.     }
8. }
9. data.cascadeIndex = i;
```



选择最佳的级联单元

现在我们获得了具有更好纹素分布的阴影。在级联之间的曲线过渡边界同样可见，这是由自阴影的伪影导致，尽管我们可以通过用级联索引除以 4 来代替阴影衰减，使其更明显。

```
1. Light GetDirectionalLight (int index, Surface surfaceWS, ShadowData shadowData) {
2.     ...
3.     light.attenuation = GetDirectionalShadowAttenuation(dirShadowData, surfaceWS);
4.     light.attenuation = shadowData.cascadeIndex * 0.25;
5.     return light;
6. }
```



用级联索引来绘制阴影

### 3.5. 剔除阴影采样

如果最终超出了最后一个级联，那么很可能没有有效的阴影数据，所以我们完全不用进行阴影采样。执行这一步的一个简单方法是在 ShadowData 里添加一个 strength 字段，默认设置为 1，如果最终超出最后一个级联，则设为 0。

```
1. struct ShadowData {
2.     int cascadeIndex;
3.     float strength;
4. };
5.
6. ShadowData GetShadowData (Surface surfaceWS) {
7.     ShadowData data;
8.     data.strength = 1.0;
9.     int i;
10.    for (i = 0; i < _CascadeCount; i++) {
11.        ...
12.    }
13.
14.    if (i == _CascadeCount) {
15.        data.strength = 0.0;
16.    }
17.
18.    data.cascadeIndex = i;
19.    return data;
20. }
```

然后把 GetDirectionalShadowData 里的定向阴影强度乘以全局阴影强度系数。这样就剔除了所有超出最后级联的阴影。

```

1. data.strength =
2.     _DirectionalLightShadowData[lightIndex].x * shadowData.strength;

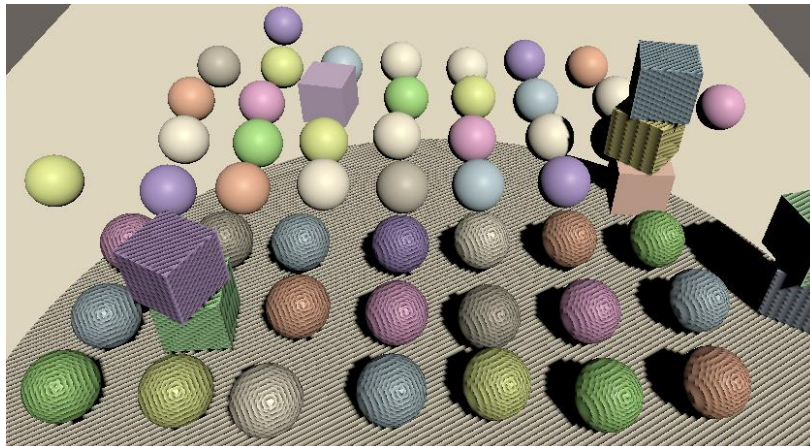
```

同样，在 `GetDirectionalLight` 里重新记录正确的衰减。

```

1. light.attenuation = GetDirectionalShadowAttenuation(dirShadowData, surfaceWS
    );
2. //light.attenuation = shadowData.cascadeIndex * 0.25;

```



剔除阴影；最大距离 12

### 3.6. 最大距离

一些关于最大阴影距离的实验表明，部分阴影投射仍然在最后一个级联的剔除球内时会突然消失。出现这种情况的原因是最外层的剔除球不是正好在配置的最大距离处结束，而是略微超出一点。当最大距离较小时这个差异更加明显。

我们可以通过在最大距离处同样停止采样来修正这部分突出的阴影。为了实现这个功能，我们需要在 `Shadows` 里将最大距离上传到 GPU。

```

1. static int
2.     ...
3.     cascadeCullingSpheresId = Shader.PropertyToID("_CascadeCullingSpheres"),
4.     shadowDistanceId = Shader.PropertyToID("_ShadowDistance");
5.
6. ...
7.
8. void RenderDirectionalShadows () {
9.     ...
10.    buffer.SetGlobalFloat(shadowDistanceId, settings.maxDistance);
11.    buffer.EndSample(bufferName);

```

```
12.     ExecuteBuffer();
13. }
```

最大距离基于视图空间深度，而不是到摄像机位置的距离。所以为了进行剔除，我们需要知道表面的深度。在 Surface 里添加一个字段。

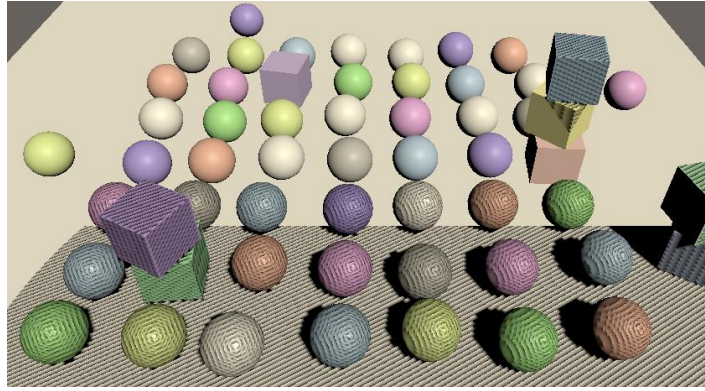
```
1. struct Surface {
2.     float3 position;
3.     float3 normal;
4.     float3 viewDirection;
5.     float depth;
6.     ...
7. };
```

深度值可以在 LitPassFragment 里通过 TransformWorldToView 从世界空间变换到视图空间，并将 Z 坐标取负来得到。因为这个变换只包括相对世界空间的旋转和偏移，所以深度值在世界空间和视图空间中是相同的。

```
1. surface.viewDirection = normalize(_WorldSpaceCameraPos - input.positionWS);
2. surface.depth = -TransformWorldToView(input.positionWS).z;
```

现在将 GetShadowData 里的强度改为只在表面深度小于最大距离时才设置为 1，否则设置为 0。

```
1. CBUFFER_START(_CustomShadows)
2.     ...
3.     float _ShadowDistance;
4. CBUFFER_END
5.
6. ...
7.
8. ShadowData GetShadowData (Surface surfaceWS) {
9.     ShadowData data;
10.    data.strength = surfaceWS.depth < _ShadowDistance ? 1.0 : 0.0;
11.    ...
12. }
```



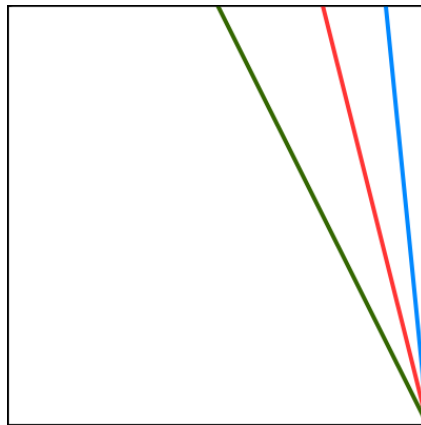
同样基于深度剔除

### 3.7. 阴影衰减

在最大距离处突然切断阴影会非常明显，所以我们通过线性衰减来使过渡更平滑。衰减从最

大距离之前的某个距离开始，知道在最大距离处达到强度为 0。我们可以使用  $1 - \frac{d}{m}$  并将其

限制在 0-1 的范围内来进行过渡，其中  $d$  是表面深度， $m$  是最大阴影距离， $f$  是衰减范围，表现为最大距离的分数。



$f$  为 0.1, 0.2 和 0.5

在阴影设置里为距离衰减添加一个滑块。因为衰减和最大距离都用做分母，不能为 0，所以把它们最小值设为 0.001。

1. `[Min(0.001f)]`
2. `public float maxDistance = 100f;`
- 3.
4. `[Range(0.001f, 1f)]`

```
5. public float distanceFade = 0.1f;
```

将 Shadows 里的阴影距离 ID 对应的数据同时包含距离和衰减值。

```
1. //shadowDistanceId = Shader.PropertyToID("_ShadowDistance");
2. shadowDistanceFadeId = Shader.PropertyToID("_ShadowDistanceFade");
```

当将它们作为向量的 XY 分量上传给 GPU 时, 使用这些值的倒数以便在着色器里避免使用除法, 因为乘法速度更快。

```
1. buffer.SetGlobalFloat(shadowDistanceId, settings.maxDistance);
2. buffer.SetGlobalVector(
3.     shadowDistanceFadeId,
4.     new Vector4(1f / settings.maxDistance, 1f / settings.distanceFade)
5. );
```

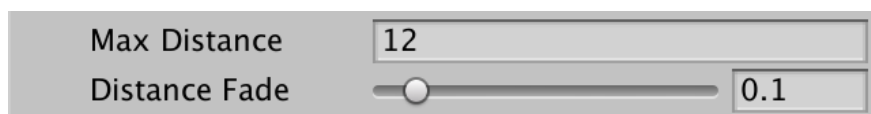
调整 Shadows 里的 \_CustomShadows 使其匹配。

```
1. //float _ShadowDistance;
2. float4 _ShadowDistanceFade;
```

现在我们可以使用  $(1 - ds)f$  进行 saturate 来计算衰减阴影强度,  $\frac{1}{m}$  为缩放  $s$ ,  $\frac{1}{f}$  为新衰

减乘数  $f$ 。创建 FadedShadowStrength 函数并在 GetShadowData 里调用。

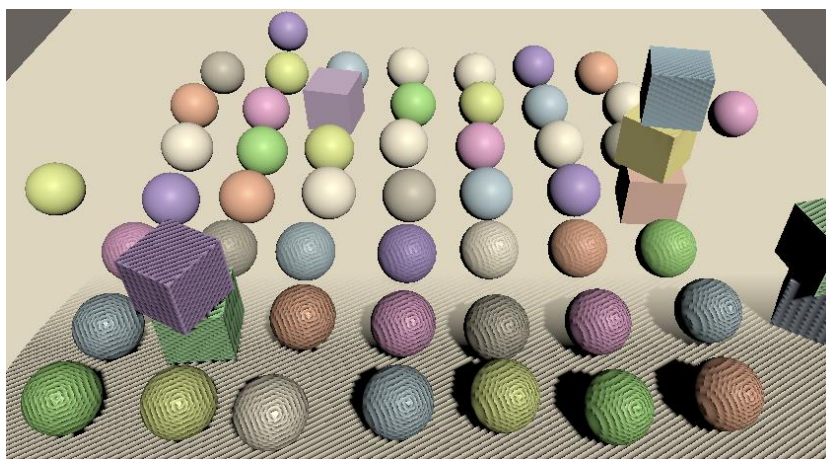
```
1. float FadedShadowStrength (float distance, float scale, float fade) {
2.     return saturate((1.0 - distance * scale) * fade);
3. }
4.
5. ShadowData GetShadowData (Surface surfaceWS) {
6.     ShadowData data;
7.     data.strength = FadedShadowStrength(
8.         surfaceWS.depth, _ShadowDistanceFade.x, _ShadowDistanceFade.y
9.     );
10.    ...
11. }
```



Max Distance 12

Distance Fade 0.1





距离衰减

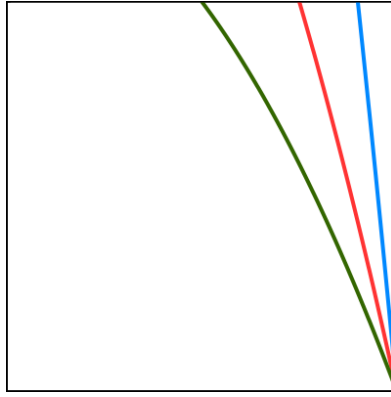
### 3.8. 级联衰减

我们也可以使用相同的方法在最后一个级联边缘进行阴影衰减，而不是切断阴影。为其添加一个级联衰减阴影设置的滑块。

```
1. public struct Directional {  
2.  
3.     ...  
4.  
5.     [Range(0.001f, 1f)]  
6.     public float cascadeFade;  
7. }  
8.  
9. public Directional directional = new Directional {  
10.     ...  
11.     cascadeRatio3 = 0.5f,  
12.     cascadeFade = 0.1f  
13. };
```

唯一的不同是对于级联我们使用距离和半径的平方，而不是线性深度和最大值。这意味着过渡不是线性的：

$\frac{1 - \frac{d^2}{r^2}}{f}$ ， $r$  是剔除球的半径。这个差别不明显，但为了保持配置的衰减比例相同，我们需要将  $f$  改成  $1 - (1 - f)^2$ 。然后将其存在 ShadowDistanceFade 向量的 Z 分量里，同样使用倒数。



$f = 0.1, 0.2, \text{ 和 } 0.5$

```

1. float f = 1f - settings.directional.cascadeFade;
2. buffer.SetGlobalVector(
3.     shadowDistanceFadeId, new Vector4(
4.         1f / settings.maxDistance, 1f / settings.distanceFade,
5.         1f / (1f - f * f)
6.     )
7. );

```

为了执行级联衰减，当在 GetShadowData 的循环里时需要检查是否是最后一个级联。如果是，则对当前级联计算衰减阴影强度并将其作为最终强度的系数。

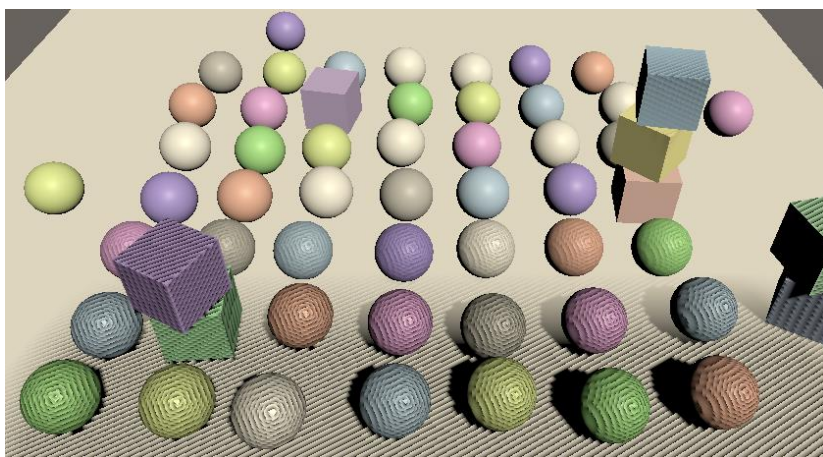
```

1. for (i = 0; i < _CascadeCount; i++) {
2.     float4 sphere = _CascadeCullingSpheres[i];
3.     float distanceSqr = DistanceSquared(surfaceWS.position, sphere.xyz);
4.     if (distanceSqr < sphere.w) {
5.         if (i == _CascadeCount - 1) {
6.             data.strength *= FadedShadowStrength(
7.                 distanceSqr, 1.0 / sphere.w, _ShadowDistanceFade.z
8.             );
9.         }
10.        break;
11.    }
12. }

```

Cascade Ratio 3	<input type="range" value="0.5"/>	0.5
Cascade Fade	<input type="range" value="0.1"/>	0.1

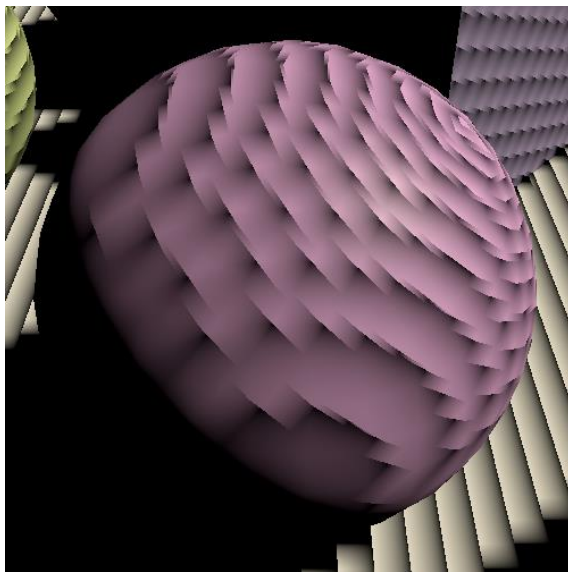




同时包含级联衰减和距离衰减

## 4. 阴影质量

现在我们实现了级联阴影贴图的功能，接下来让我们专注于提升阴影质量。一直以来，我们观察到的这些伪影被称作 shadow acne，这是由于没有完全对其光源方向的表面产生的错误自阴影而造成的。当表面更贴近平行于光源方向时，acne 效果就会更糟。



Shadow acne

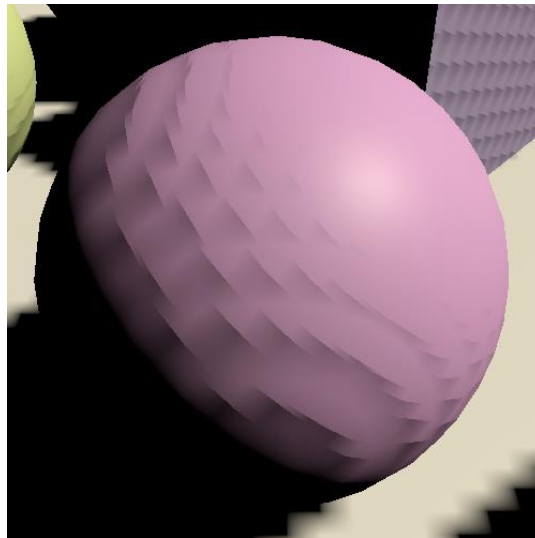
增加图集尺寸会减小纹素在世界空间中的尺寸，所以 acne 会变小。但是，伪影的数量同样变多了，所以这个问题不能通过增加图集大小来解决。

### 4.1. 深度偏移

有多种方法来减少 shadow acne。最简单的是对阴影投射的深度值添加一个偏移常量，将他

们像远离光源的方向推，以至于不再产生错误的自阴影。添加这项技术的最快方法是在渲染使应用一个全局深度偏移，在 DrawShadows 之前调用缓冲区的 SetGlobalDepthBias，并在之后将其设回 0。这是一个应用在裁剪空间的深度便宜，并且是一个非常小的值的倍数，具体取决于阴影贴图使用的确切格式。我们可以通过使用一个大数值（比如 50000）来搞清楚它如何起效。还有第二个参数用于斜率偏移，但我们先使用 0。

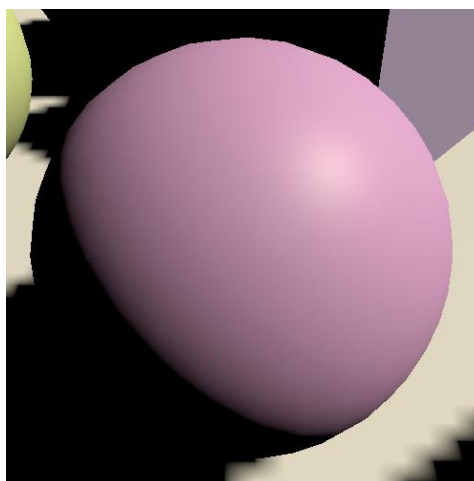
```
1. buffer.SetGlobalDepthBias(50000f, 0f);  
2. ExecuteBuffer();  
3. context.DrawShadows(ref shadowSettings);  
4. buffer.SetGlobalDepthBias(0f, 0f);
```



常量深度偏移

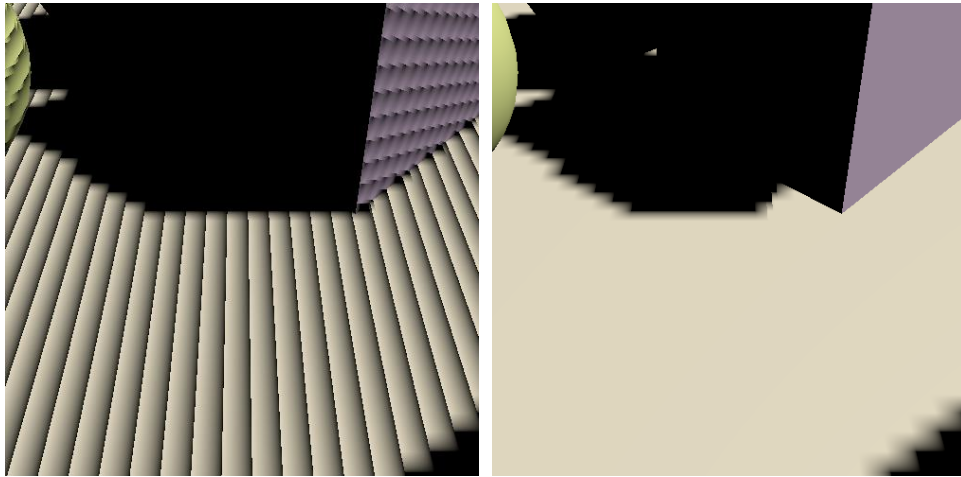
常量偏移很简单，但只能去除光线正面照射表面的伪影。移除所有的 acne 需要一个大得多的偏移，比如增大一个数量级。

```
1. buffer.SetGlobalDepthBias(500000f, 0f);
```



更大的深度偏移

然而, 因为深度便宜将阴影投射想原理光源的方向推挤, 采样的阴影同样向相同方向移动了。这些用来移除大多数 acne 的足够大的定量偏移同样将阴影移动很远, 以至于它们与投射物相分离, 引起了可见的伪影, 这被称作 Peter-Panning。



偏移引起 peter-panning

另一个方法是使用斜率比例偏移, 这通过在 SetGlobalDepthBias 里对第二个参数使用一个非零值来实现。这个值用于对绝对裁剪空间深度值沿 X 和 Y 方向的偏导最大值进行缩放。所以对于光线直射的表面来说为 0, 当光线至少在其中一个方向上以  $45^\circ$  照射时为 1, 当表面法线和光方向点乘为 0 时达到无穷大。所以当需要的时候, 偏移会自动增大, 但没有上限。于是需要一个小得多的系数来消除 acne, 例如用 3 代替 500000。

```
1. buffer.SetGlobalDepthBias(0f, 3f);
```



斜率缩放偏移

斜率缩放偏移更有效但不直观。需要通过实验来获得一个用伪影代替 Peter-Panning 的可接受的结果。所以我们先禁用它, 并寻找一个更直观和可预测的方法。

```
1. //buffer.SetGlobalDepthBias(0f, 3f);  
2. ExecuteBuffer();
```

```

3. context.DrawShadows(ref shadowSettings);
4. //buffer.SetGlobalDepthBias(0f, 0f);

```

## 4.2. 级联数据

因为 acne 的大小取决于世界空间内纹素的大小，所以需要在所有情况下都兼容的方法必须考虑到这一点。由于对于每个级联来说，纹素大小不同，这就意味着我们需要向 GPU 发送更多的级联数据。在 Shadows 里为其添加一个通用的级联数据向量数组。

```

1. static int
2.     ...
3.     cascadeCullingSpheresId = Shader.PropertyToID("_CascadeCullingSpheres"),
4.     cascadeDataId = Shader.PropertyToID("_CascadeData"),
5.     shadowDistanceFadeId = Shader.PropertyToID("_ShadowDistanceFade");
6.
7. static Vector4[]
8.     cascadeCullingSpheres = new Vector4[maxCascades],
9.     cascadeData = new Vector4[maxCascades];

```

像其他内容一样上传给 GPU。

```

1. buffer.SetGlobalVectorArray(
2.     cascadeCullingSpheresId, cascadeCullingSpheres
3. );
4. buffer.SetGlobalVectorArray(cascadeDataId, cascadeData);

```

我们现在可以做的事情是将级联半径平方的倒数放在这些向量的 X 分量里。这样的话我们就不需要在着色器里执行除法。在新的 SetCascadeData 方法里执行，并且存储剔除球，然后在 RenderDirectionalShadows 里进行调用。传入级联索引，剔除球和图块尺寸的浮点数。

```

1. void RenderDirectionalShadows (int index, int split, int tileSize) {
2.     ...
3.
4.     for (int i = 0; i < cascadeCount; i++) {
5.         ...
6.         if (index == 0) {
7.             SetCascadeData(i, splitData.cullingSphere, tileSize);
8.         }
9.         ...
10.    }
11. }
12.
13. void SetCascadeData (int index, Vector4 cullingSphere, float tileSize) {

```

```

14.     cascadeData[index].x = 1f / cullingSphere.w;
15.     cullingSphere.w *= cullingSphere.w;
16.     cascadeCullingSpheres[index] = cullingSphere;
17. }

```

在 Shadows 里把级联数据添加到 \_CustomShadows。

```

1. CBUFFER_START(_CustomShadows)
2.     int _CascadeCount;
3.     float4 _CascadeCullingSpheres[MAX_CASCADE_COUNT];
4.     float4 _CascadeData[MAX_CASCADE_COUNT];
5.     ...
6. CBUFFER_END

```

然后在 GetShadowData 里使用预计算的倒数。

```

1. data.strength *= FadedShadowStrength(
2.     distanceSqr, _CascadeData[i].x, _ShadowDistanceFade.z
3. );

```

## 4.3. 法线偏移

产生不正确的自阴影的原因是阴影投射深度的纹素覆盖了多于一个片元, 导致投射单元的体积超出了表面。所以如果我们将投射收缩到一定程度就不会发生这种情况。然而, 缩小阴影投射是的阴影变得比原本药效, 并且会产生不应存在的孔洞。

我们也可以反向操作: 当采样阴影时将表面向外膨胀。我们远离表面一些的地方进行采样, 这个距离刚好足够避免不正确的自阴影。这将略微调整阴影的位置, 可能会引起边缘的错位和产生假阴影, 但伪影远比 Peter-Panning 来的不明显。

我们可以为了采样阴影而将表面位置沿法线方向移动一小段距离。如果我们只考虑一维, 那么一个等于世界空间内纹素尺寸的偏移就足够了。我们可以在 SetCascadeData 里通过用剔除球的直径除以图块尺寸来获得纹素大小。将它存在级联数据向量的 Y 分量。

```

1. float texelSize = 2f * cullingSphere.w / tileSize;
2. cullingSphere.w *= cullingSphere.w;
3. cascadeCullingSpheres[index] = cullingSphere;
4. //cascadeData[index].x = 1f / cullingSphere.w;
5. cascadeData[index] = new Vector4(
6.     1f / cullingSphere.w,
7.     texelSize
8. );

```

然而, 这并不满足所有情况, 因为纹素是正方形。最差情况下, 我们最终需要沿着正方形的

对角线进行偏移，所以把它乘以 $\sqrt{2}$ 。

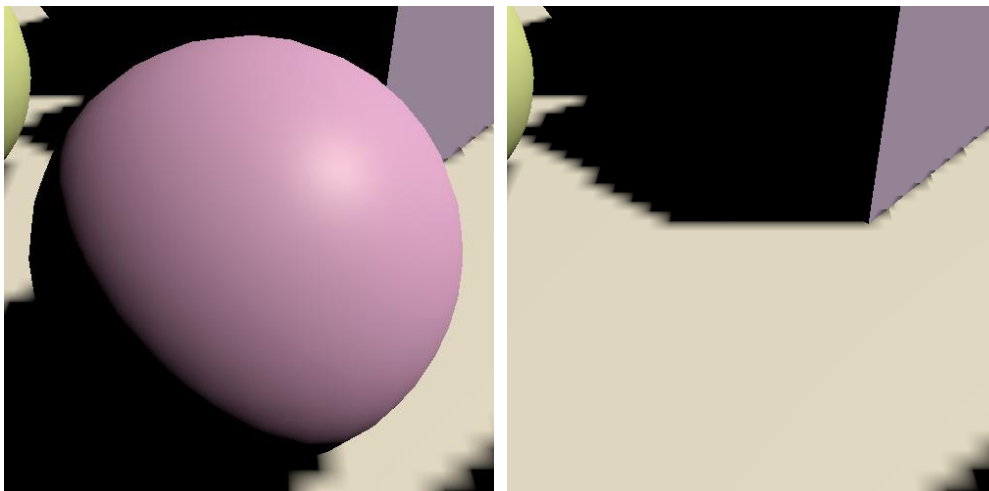
```
1. texelSize * 1.4142136f
```

在着色器这边，添加一个全局阴影数据作为 GetDirectionalShadowAttenuation 的参数。用表面法线乘以偏移来获取 normalBias，并将其在计算阴影图块空间的坐标前加给世界坐标。

```
1. float GetDirectionalShadowAttenuation (  
2.     DirectionalShadowData directional, ShadowData global, Surface surfaceWS  
3. ) {  
4.     if (directional.strength <= 0.0) {  
5.         return 1.0;  
6.     }  
7.     float3 normalBias = surfaceWS.normal * _CascadeData[global.cascadeIndex]  
8.         .y;  
9.     float3 positionSTS = mul(  
10.         _DirectionalShadowMatrices[directional.tileIndex],  
11.         float4(surfaceWS.position + normalBias, 1.0)  
12.     ).xyz;  
13.     float shadow = SampleDirectionalShadowAtlas(positionSTS);  
14.     return lerp(1.0, shadow, directional.strength);  
15. }
```

在 GetDirectionalLight 里将额外数据传进去。

```
1. light.attenuation =  
2.     GetDirectionalShadowAttenuation(dirShadowData, shadowData, surfaceWS);
```



法线偏移等于纹素大小

## 4.4. 可配置偏移

法线偏移可以在不产生新的明显伪影的情况下避免 shadow acne，但它不能解决所有的阴影问题。比如，在靠近墙面的地面上会有本不该出现的阴影线条。这不是自阴影，而是墙面向外膨胀的阴影影响到了下面的地面。添加一个很小的斜率缩放偏移可以解决这个问题，但并不存在一个完美值。所以我们将基于光源用他们已有的 Bias 滑块来配置。在 Shadows 里的 ShadowedDirectionalLight 结构里为其添加一个字段。

```
1. struct ShadowedDirectionalLight {
2.     public int visibleLightIndex;
3.     public float slopeScaleBias;
4. }
```

光源的 bias 可以通过 shadowBias 属性来获得。将其添加到 ReserveDirectionalShadows 的数据里。

```
1. shadowedDirectionalLights[ShadowedDirectionalLightCount] =
2.     new ShadowedDirectionalLight {
3.         visibleLightIndex = visibleLightIndex,
4.         slopeScaleBias = light.shadowBias
5.     };
```

并且使用它在 RenderDirectionalShadows 里配置斜率缩放偏移。

```
1. buffer.SetGlobalDepthBias(0f, light.slopeScaleBias);
2. ExecuteBuffer();
3. context.DrawShadows(ref shadowSettings);
4. buffer.SetGlobalDepthBias(0f, 0f);
```

让我们同样使用光源已有的 Normal Bias 滑块来调制我们使用的法线偏移。让 ReserveDirectionalShadows 返回一个 Vector3，并且使用光源的 shadowNormalBias 作为新的 Z 分量。

```
1. public Vector3 ReserveDirectionalShadows (
2.     Light light, int visibleLightIndex
3. ) {
4.     if (...) {
5.         ...
6.         return new Vector3(
7.             light.shadowStrength,
8.             settings.directional.cascadeCount * ShadowedDirectionalLightCount++,
9.             light.shadowNormalBias
10.        );
11.    }
```

```

11.     }
12.     return Vector3.zero;
13. }

```

将新的 normalBias 添加到 DirectionalShadowData，并在 GetDirectionalShadowAttenuation 中使用。

```

1. struct DirectionalShadowData {
2.     float strength;
3.     int tileIndex;
4.     float normalBias;
5. };
6.
7. ...
8.
9. float GetDirectionalShadowAttenuation (...) {
10.     ...
11.     float3 normalBias = surfaceWS.normal *
12.         (directional.normalBias * _CascadeData[global.cascadeIndex].y);
13.     ...
14. }

```

并在 Light 的 GetDirectionalShadowData 里进行配置。

```

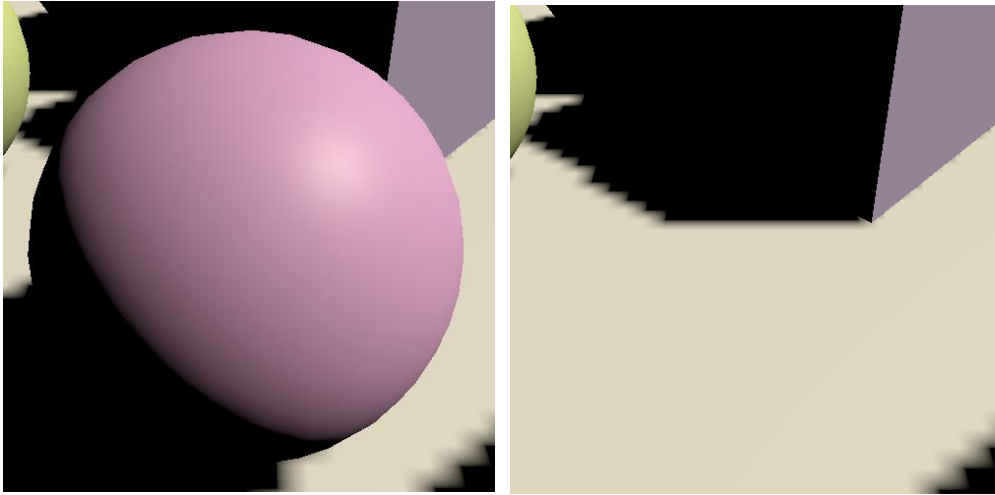
1. data.tileIndex =
2.     _DirectionalLightShadowData[lightIndex].y + shadowData.cascadeIndex;
3. data.normalBias = _DirectionalLightShadowData[lightIndex].z;

```

现在我们可以基于光源调整两种偏移。斜率缩放偏移用 0，法线偏移用 1 是一个很好的默认设置。如果你增大了第一个就需要减小第二个。但需要注意的是，我们对这些灯光设置的解释与原始解释不同。它们原本是裁剪空间的深度偏移和世界空间的收缩法线偏移。所以当你创建一个新的灯光时，除非对这些偏移进行调整，否则会得到明显的 Peter-Panning。



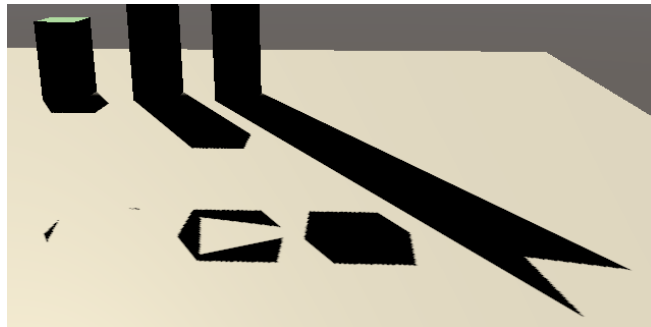




两个偏移都设为 0.6

## 4.5. Shadow Pancaking (阴影薄片? 不太确定中文翻译)

另一个可能会引起伪影的问题是 Unity 使用了阴影着陆。这功能是为定向光渲染阴影投射时，近平面尽可能地向前移动。这提高了深度精度，但也意味着不在摄像机视图范围内的阴影投射可能最终会落在近平面之前，这会引起原本不该有的裁剪。



被裁剪的阴影

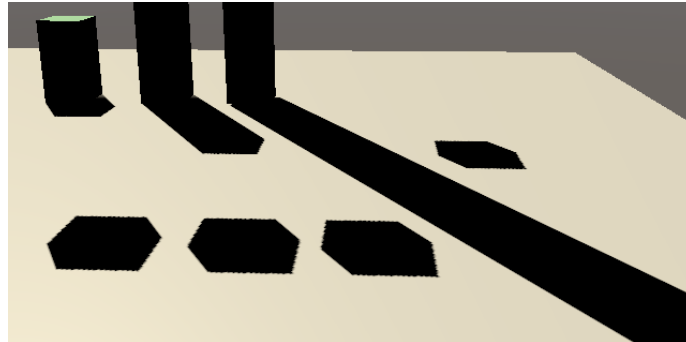
这通过在 ShadowCasterPassVertex 里将顶点位置限制在近平面来解决，这可以有效地将落在近平面前方的阴影投射铺平，将它们转化成紧贴近平面的薄片。我们通过获取裁剪空间内 Z 和 W 坐标的最大值（当 UNITY\_REVERSED\_Z 定义时使用最小值）。为了使用正确的 W 坐标符号，需要将其乘以 UNITY\_NEAR\_CLIP\_VALUE。

```
1. output.positionCS = TransformWorldToHClip(positionWS);
2.
3. #if UNITY_REVERSED_Z
4.     output.positionCS.z =
5.         min(output.positionCS.z, output.positionCS.w * UNITY_NEAR_CLIP_VALUE
6.         );
7. #else
8.     output.positionCS.z =
```

```

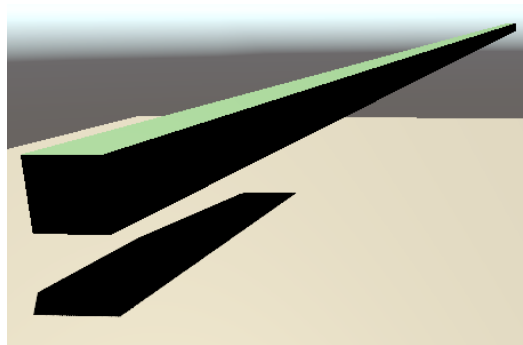
8.         max(output.positionCS.z, output.positionCS.w * UNITY_NEAR_CLIP_VALUE
    );
9. #endif

```



限制范围的阴影

这个方法对完全处在近平面任意一边的阴影投射对象都有效, 但穿越平面的阴影投射对象会发生变形, 因为只有部分顶点收到了影响。对于小三角面来说并不明显, 但大型三角面最终会明显变形, 弯曲并且经常会造成陷入表面以内。



长立方体的阴影变形

这个问题可以通过将近平面向回反推一些来缓解。这对应光源里 Near Plane 滑块的作用。在 ShadowedDirectionalLight 里为近平面偏移添加一个字段。

```

1. struct ShadowedDirectionalLight {
2.     public int visibleLightIndex;
3.     public float slopeScaleBias;
4.     public float nearPlaneOffset;
5. }

```

将光源的 shadowNearPlane 属性赋值给它。

```

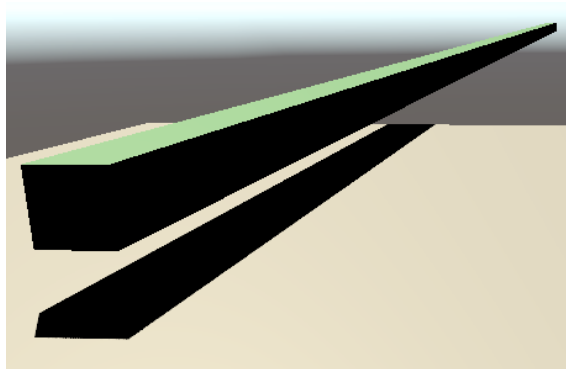
1. shadowedDirectionalLights[ShadowedDirectionalLightCount] =
2.     new ShadowedDirectionalLight {
3.         visibleLightIndex = visibleLightIndex,
4.         slopeScaleBias = light.shadowBias,
5.         nearPlaneOffset = light.shadowNearPlane

```

```
6.     };
```

我们将这个值赋给 `ComputeDirectionalShadowMatricesAndCullingPrimitives` 中我们还在使用固定值 0 的最后一个参数。

```
1. cullingResults.ComputeDirectionalShadowMatricesAndCullingPrimitives(  
2.     light.visibleLightIndex, i, cascadeCount, ratios, tileSize,  
3.     light.nearPlaneOffset, out Matrix4x4 viewMatrix,  
4.     out Matrix4x4 projectionMatrix, out ShadowSplitData splitData  
5. );
```



带有近平面偏移

## 4.6. PCF 滤波

到目前为止，我们只通过逐片元采样了一次阴影贴图来使用了硬阴影。阴影比较采样器使用了一种特殊的双线性插值格式，在插值之前进行了深度比较。这被称作百分比邻近滤波（简称 PCF），因为其中包含四个纹素，所以特别被称作 2x2PCF 滤波。

但这不是唯一可以进行纹理贴图滤波的方法。我们也可以使用更大的滤镜使阴影更柔和并更不容易混叠，尽管精确度也较低。让我们支持 2x2, 3x3, 5x5 和 7x7 的滤镜。我们将不使用基于单个光源已存在的软阴影模式来控制，而让所有定向光使用相同的滤镜。在 `ShadowSettings` 添加一个 `FilterMode` 枚举，同样在 `Directional` 里添加滤镜选项，默认值设为 2x2。

```
1. public enum FilterMode {  
2.     PCF2x2, PCF3x3, PCF5x5, PCF7x7  
3. }  
4.  
5. ...  
6.  
7. [System.Serializable]  
8. public struct Directional {  
9.
```

```

10.     public MapSize atlasSize;
11.
12.     public FilterMode filter;
13.
14.     ...
15. }
16.
17. public Directional directional = new Directional {
18.     atlasSize = MapSize._1024,
19.     filter = FilterMode.PCF2x2,
20.     ...
21. };

```

Atlas Size	512
Filter	PCF 2x2

滤镜设置为 PCF 2x2

我们为新的滤镜模式创建着色器变量。在 Shadows 里添加一个带有三个关键字的静态数组。

```

1. static string[] directionalFilterKeywords = {
2.     "_DIRECTIONAL_PCF3",
3.     "_DIRECTIONAL_PCF5",
4.     "_DIRECTIONAL_PCF7",
5. };

```

创建 SetKeywords 方法来启用或禁用合适的关键字。在 RenderDirectionalShadows 执行缓冲区之前调用。

```

1. void RenderDirectionalShadows () {
2.     ...
3.     SetDirectionalKeywords();
4.     buffer.EndSample(bufferName);
5.     ExecuteBuffer();
6. }
7.
8. void SetKeywords () {
9.     int enabledIndex = (int)settings.directional.filter - 1;
10.    for (int i = 0; i < directionalFilterKeywords.Length; i++) {
11.        if (i == enabledIndex) {
12.            buffer.EnableShaderKeyword(directionalFilterKeywords[i]);
13.        }
14.        else {
15.            buffer.DisableShaderKeyword(directionalFilterKeywords[i]);
16.        }

```

```

17.     }
18. }

```

更大的滤镜需要更多的纹理采样。我们需要在着色器中同时知道图集大小和纹素大小来进行采样。为这些数据添加一个着色器 ID。

```

1. cascadeDataId = Shader.PropertyToID("_CascadeData"),
2. shadowAtlasSizeId = Shader.PropertyToID("_ShadowAtlasSize"),
3. shadowDistanceFadeId = Shader.PropertyToID("_ShadowDistanceFade");

```

将图集大小存入 X 分量，纹素大小存入 Y 分量。

```

1. SetKeywords();
2. buffer.SetGlobalVector(
3.     shadowAtlasSizeId, new Vector4(atlasSize, 1f / atlasSize)
4. );

```

在 CustomLit 通道里对这三个关键字添加 `#pragma mulit_compile` 指令，并用下划线代表无关键字选项，它对应 2x2 滤镜。

```

1. #pragma shader_feature _PREMULTIPLY_ALPHA
2. #pragma multi_compile _ _DIRECTIONAL_PCF3 _DIRECTIONAL_PCF5 _DIRECTIONAL_PCF7
3. #pragma multi_compile_instancing

```

我们将使用在 Core RP 库的 Shadow/ShadowSamplingTent.hlsl 文件里定义的方法，所以在 Shadows 的开头引入它。如果定义了 3x3 关键词，我们一共需要四次滤镜采样，使用 `SampleShadow_ComputeSamples_Tent_3x3` 方法进行设置。我们只需要进行四次采样是因为每一次都使用了 2x2 的双线性过滤。在各个方向上偏移半个纹素构成的正方形使用 tent filter 覆盖了 3x3 的纹素范围，中心权重高于边缘。

```

1. #include "Packages/com.unity.render-
   pipelines/core/ShaderLibrary/Shadow/ShadowSamplingTent.hlsl"
2.
3. #if defined(_DIRECTIONAL_PCF3)
4.     #define DIRECTIONAL_FILTER_SAMPLES 4
5.     #define DIRECTIONAL_FILTER_SETUP SampleShadow_ComputeSamples_Tent_3x3
6. #endif
7.
8. #define MAX_SHADOWED_DIRECTIONAL_LIGHT_COUNT 4
9. #define MAX_CASCADE_COUNT 4

```

基于同样的原因，我们可以对 5x5 过滤进行 9 次采样，并且对于 7x7 进行 16 次采样，再加上对应名称的方法。

```

1. #if defined(_DIRECTIONAL_PCF3)
2.     #define DIRECTIONAL_FILTER_SAMPLES 4
3.     #define DIRECTIONAL_FILTER_SETUP SampleShadow_ComputeSamples_Tent_3x3
4. #elif defined(_DIRECTIONAL_PCF5)
5.     #define DIRECTIONAL_FILTER_SAMPLES 9
6.     #define DIRECTIONAL_FILTER_SETUP SampleShadow_ComputeSamples_Tent_5x5
7. #elif defined(_DIRECTIONAL_PCF7)
8.     #define DIRECTIONAL_FILTER_SAMPLES 16
9.     #define DIRECTIONAL_FILTER_SETUP SampleShadow_ComputeSamples_Tent_7x7
10. #endif

```

对阴影图块空间坐标创建一个新的 FilterDirectionalShadow 方法。当 DIRECTIONAL\_FILTER\_SETUP 被定义，那么需要进行多次采样，否则只需调用一次 SampleDirectionalShadowAtlas 即可满足需求。

```

1. float FilterDirectionalShadow (float3 positionSTS) {
2.     #if defined(DIRECTIONAL_FILTER_SETUP)
3.         float shadow = 0;
4.         return shadow;
5.     #else
6.         return SampleDirectionalShadowAtlas(positionSTS);
7.     #endif
8. }

```

Filter 设置函数包含四个参数。第一个是 float4，其中前两个分量代表 XY 方向的纹素大小，Z 和 W 代表纹理总尺寸。然后是原始采样点，接着是关于每次采样的权重和位置的输出参数。它们通过 float 和 float2 数组进行定义。之后我们可以循环所有采样，叠加权重后进行累加。

```

1. #if defined(DIRECTIONAL_FILTER_SETUP)
2.     float weights[DIRECTIONAL_FILTER_SAMPLES];
3.     float2 positions[DIRECTIONAL_FILTER_SAMPLES];
4.     float4 size = _ShadowAtlasSize.yyxx;
5.     DIRECTIONAL_FILTER_SETUP(size, positionSTS.xy, weights, positions);
6.     float shadow = 0;
7.     for (int i = 0; i < DIRECTIONAL_FILTER_SAMPLES; i++) {
8.         shadow += weights[i] * SampleDirectionalShadowAtlas(
9.             float3(positions[i].xy, positionSTS.z)
10.        );
11.    }
12.    return shadow;
13. #else

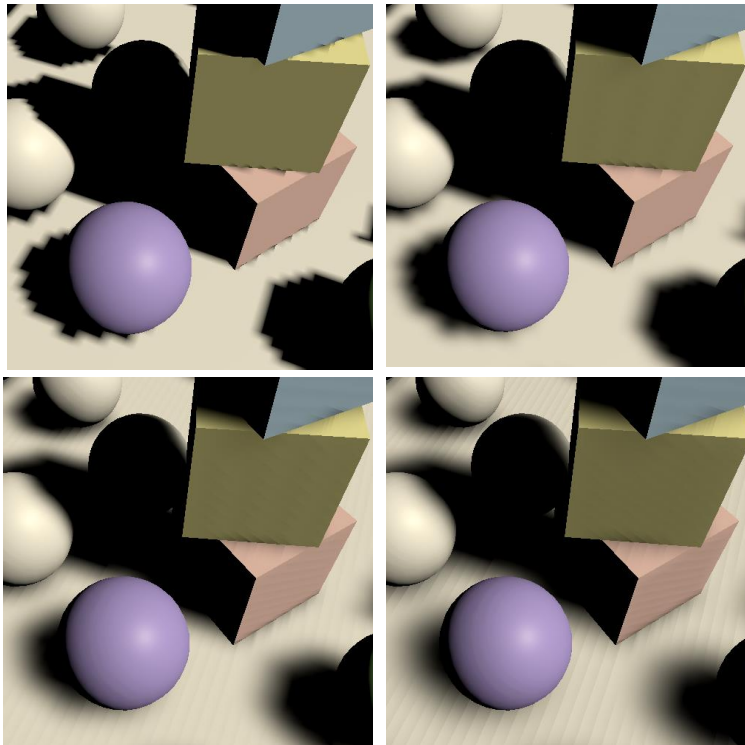
```

在 GetDirectionalShadowAttenuation 里调用新方法来代替 SampleDirectionalShadowAtlas。

```

1. float shadow = FilterDirectionalShadow(positionSTS);
2. return lerp(1.0, shadow, directional.strength);

```



*PCF 2x2, 3x3, 5x5, 7x7*

提高 filter 尺寸使阴影更平滑，但同样会使 acne 再次出现。我们需要增大 normal bias 来匹配 filter 尺寸。我们可以在 SetCascadeData 里用纹素尺寸乘以 1+filter 模式来自动匹配。

```

1. void SetCascadeData (int index, Vector4 cullingSphere, float tileSize) {
2.     float texelSize = 2f * cullingSphere.w / tileSize;
3.     float filterSize = texelSize * ((float)settings.directional.filter + 1f)
4.     ;
5.     ...
6.     1f / cullingSphere.w,
7.     filterSize * 1.4142136f
8.     );
9. }

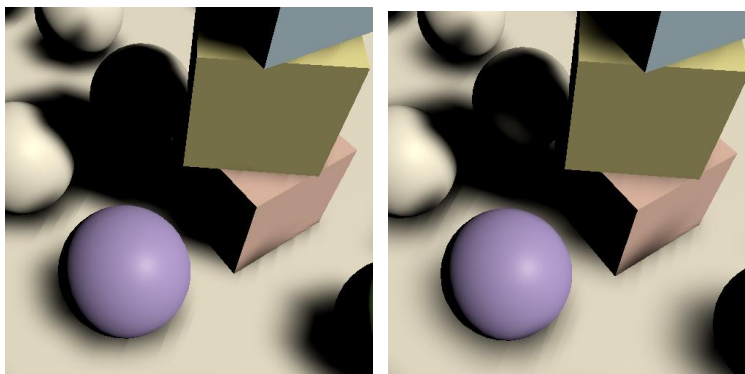
```

除此之外，增大采样区域意味着我们最终会采样到级联剔除球之外。我们可以在平方之前将球体半径减少对应的 filter 尺寸来避免这种情况发生。

```

1. cullingSphere.w -= filterSize;
2. cullingSphere.w *= cullingSphere.w;

```

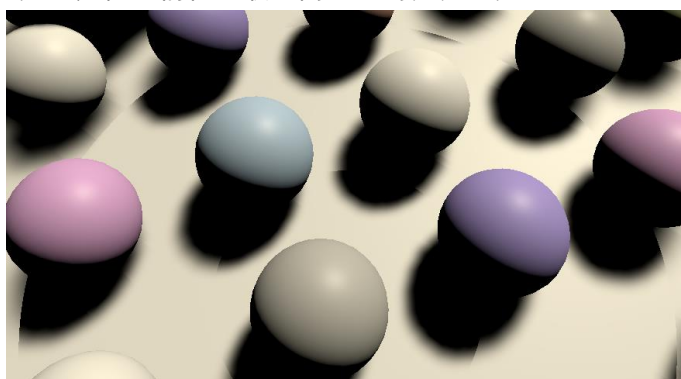


PCF 5x5 和 7x7, 包含缩放偏移

这样再次解决的 shadow acne 的问题, 但增大的 filter 尺寸加重了使用法线偏移的负面影响, 并也使得我们之前看到的墙壁阴影更糟。需要调整斜率缩放偏移或更大的图集尺寸来减轻这些伪影。

## 4.7. 级联混合

软阴影看上去效果更好, 但它们在级联之间产生的突变也更明显了。



硬级联过渡; PCF 7x7

我们可以通过在级联之间添加一个同时混合两个级联的过渡区域让过渡不那么明显 (尽管不能完全消除)。我们已经有了一个级联衰减系数可以用来过渡。

首先, 在 Shadows 的 ShadowData 里添加一个级联混合值, 我们用这个值在相邻级联间做线性插值。

```
1. struct ShadowData {  
2.     int cascadeIndex;  
3.     float cascadeBlend;  
4.     float strength;  
5. };
```

在 GetShadowData 里先将 blend 初始化为 1, 意味着选中的级联是完全的强度。然后在循环中对符合条件的级联始终进行衰减系数的计算。如果我们处在最后一个级联, 就像之前一样将系数乘以强度, 否则就将它用作 blend。



```

1. data.cascadeBlend = 1.0;
2. data.strength = FadedShadowStrength(
3.     surfaceWS.depth, _ShadowDistanceFade.x, _ShadowDistanceFade.y
4. );
5. int i;
6. for (i = 0; i < _CascadeCount; i++) {
7.     float4 sphere = _CascadeCullingSpheres[i];
8.     float distanceSqr = DistanceSquared(surfaceWS.position, sphere.xyz);
9.     if (distanceSqr < sphere.w) {
10.         float fade = FadedShadowStrength(
11.             distanceSqr, _CascadeData[i].x, _ShadowDistanceFade.z
12.         );
13.         if (i == _CascadeCount - 1) {
14.             data.strength *= fade;
15.         }
16.         else {
17.             data.cascadeBlend = fade;
18.         }
19.         break;
20.     }
21. }

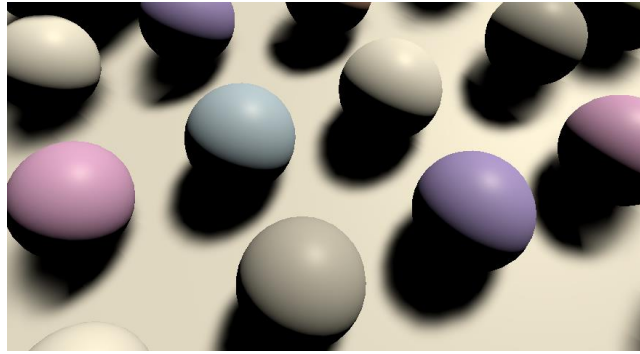
```

现在在 GetDirectionalShadowAttenuation 里获取第一个阴影值之后检查 cascadeBlend 是否小于 1。如果小于 1，说明我们处在过渡区域内，需要对下一个级联进行采样，并将两个值做线性插值。

```

1. float shadow = FilterDirectionalShadow(positionSTS);
2. if (global.cascadeBlend < 1.0) {
3.     normalBias = surfaceWS.normal *
4.         (directional.normalBias * _CascadeData[global.cascadeIndex + 1].y);
5.     positionSTS = mul(
6.         _DirectionalShadowMatrices[directional.tileIndex + 1],
7.         float4(surfaceWS.position + normalBias, 1.0)
8.     ).xyz;
9.     shadow = lerp(
10.         FilterDirectionalShadow(positionSTS), shadow, global.cascadeBlend
11.     );
12. }
13. return lerp(1.0, shadow, directional.strength);

```



软级联过渡

需要注意的是级联衰减比例应用于每个级联的整个半径，而不仅仅是可见部分。所以需要确认该比率不会影响到较低级别的级联。通常情况下这不是个问题，因为总是希望过渡区域较小。

## 4.8. 抖动过渡

尽管级联间的混合看上去效果更好，但在混合区域内我们也花费了两倍的时间用于阴影贴图采样。另一个替代方法是基于抖动模式始终在一个级联内采样。这种方法的效果看上去不像以前那么好，但消耗低得多，特别在使用大范围滤镜的时候。

给 Directional 添加一个级联混合选项，支持 hard, soft, dither。

```
1. public enum CascadeBlendMode {  
2.     Hard, Soft, Dither  
3. }  
4.  
5. public CascadeBlendMode cascadeBlend;  
6. }  
7.  
8. public Directional directional = new Directional {  
9.     ...  
10.     cascadeFade = 0.1f,  
11.     cascadeBlend = Directional.CascadeBlendMode.Hard  
12. };
```



级联混合模式

在 Shadows 里为 soft 和 dither 级联混合关键字添加一个静态数组。

```
1. static string[] cascadeBlendKeywords = {
```

```

2.     "_CASCADE_BLEND_SOFT",
3.     "_CASCADE_BLEND_DITHER"
4. };

```

调整 SetKeywords，以便能用于任意的关键字数组和索引，然后同样设置级联混合关键字。

```

1. void RenderDirectionalShadows () {
2.     SetKeywords(
3.         directionalFilterKeywords, (int)settings.directional.filter - 1
4.     );
5.     SetKeywords(
6.         cascadeBlendKeywords, (int)settings.directional.cascadeBlend - 1
7.     );
8.     buffer.SetGlobalVector(
9.         shadowAtlastSizeId, new Vector4(atlasSize, 1f / atlasSize)
10.    );
11.    buffer.EndSample(bufferName);
12.    ExecuteBuffer();
13. }
14.
15. void SetKeywords (string[] keywords, int enabledIndex) {
16.     //int enabledIndex = (int)settings.directional.filter - 1;
17.     for (int i = 0; i < keywords.Length; i++) {
18.         if (i == enabledIndex) {
19.             buffer.EnableShaderKeyword(keywords[i]);
20.         }
21.         else {
22.             buffer.DisableShaderKeyword(keywords[i]);
23.         }
24.     }
25. }

```

在 CustomLit Pass 里添加所需的多编译指令。

```

1. #pragma multi_compile _ _CASCADE_BLEND_SOFT _CASCADE_BLEND_DITHER
2. #pragma multi_compile_instancing

```

为了执行抖动，我们需要一个 float dither，将其添加到 Surface。

```

1. struct Surface {
2.     ...
3.     float dither;
4. };

```

在 LitPassFragment 里有多种方式来生成 dither 值。最简单的方法是使用来自 Core RP 库中

的 InterleavedGradientNoise 方法，该函数在给定的屏幕空间 XY 位置下生成旋转的平铺抖动图案。在片元函数里它等同于裁剪空间 XY 坐标。它还需要第二个参数用来进行动画处理，我们不需要，将其设为 0。

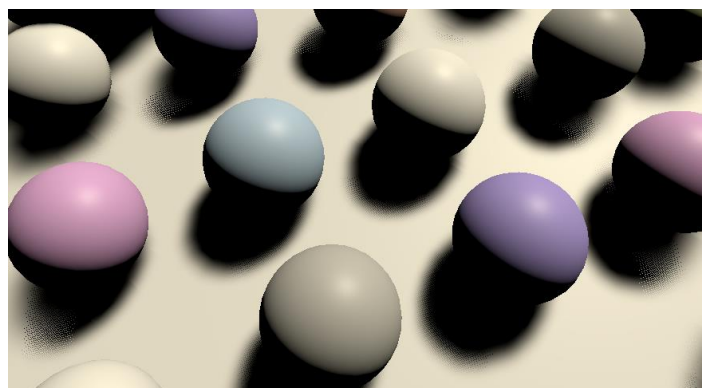
```
1. surface.smoothness =  
2.     UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Smoothness);  
3. surface.dither = InterleavedGradientNoise(input.positionCS.xy, 0);
```

在 GetShadowData 设置级联索引之前，如果没有使用 soft 混合，则将 cascadeBlend 设为 0。这样一来，整个分支将从着色器变体中移除。

```
1. if (i == _CascadeCount) {  
2.     data.strength = 0.0;  
3. }  
4. #if !defined(_CASCADE_BLEND_SOFT)  
5.     data.cascadeBlend = 1.0;  
6. #endif  
7. data.cascadeIndex = i;
```

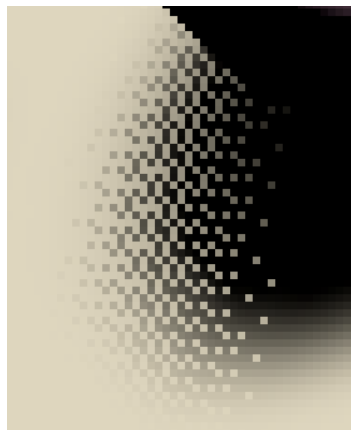
当使用 dither 混合时，如果我们不在最后一个级联，并且混合值小于 dither 值时，就跳转到下一个级联。

```
1. if (i == _CascadeCount) {  
2.     data.strength = 0.0;  
3. }  
4. #if defined(_CASCADE_BLEND_DITHER)  
5.     else if (data.cascadeBlend < surfaceWS.dither) {  
6.         i += 1;  
7.     }  
8. #endif  
9. #if !defined(_CASCADE_BLEND_SOFT)  
10.     data.cascadeBlend = 1.0;  
11. #endif
```



抖动级联

抖动混合的可接受程度取决于渲染帧的分辨率。如果对最终结果使用了后处理，那么它会变的很有效，例如同时使用了时间抗锯齿和动态抖动模式。

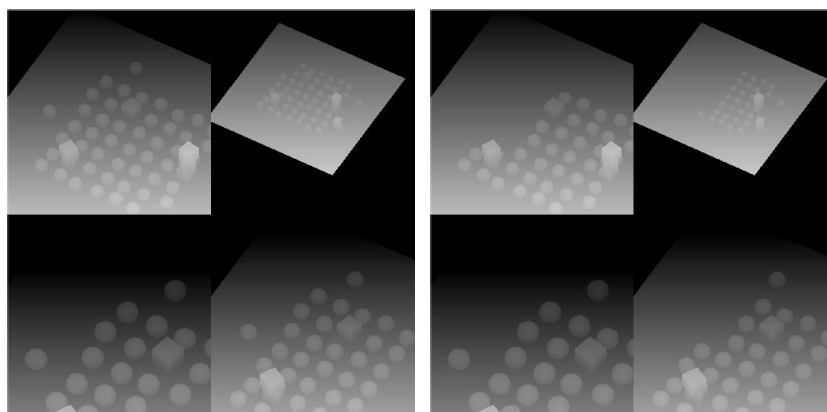


放大的抖动模式

## 4.9. 剔除偏移

一个使用阴影级联贴图的缺点在于我们最终基于光源对相同的阴影投射对象渲染了超过一次。如果可以确保一些阴影投射对象总是被较小的级联覆盖，那么就可以尝试从较大的级联中剔除这些对象。Unity 通过设置 `splitData` 的 `shadowCascadeBlendCullingFactor` 来实现这一功能。在 `RenderDirectionalShadows` 应用阴影设置之前执行这步操作。

```
1. splitData.shadowCascadeBlendCullingFactor = 1f;  
2. shadowSettings.splitData = splitData;
```



剔除偏移分别为 0 和 1

该值是一个用来调整前一个级联用来执行剔除的级联半径的系数。在进行剔除时 Unity 相当保守，但我们应该通过级联淡入淡出比率来使其降低，并额外增加一点以确保过渡区域中的阴影投射器永远不会被剔除。所以我们用 0.8 减去淡入淡出范围，最小值为 0。如果你在级联过渡区的阴影种看到了孔洞，那么它需要进一步减小。

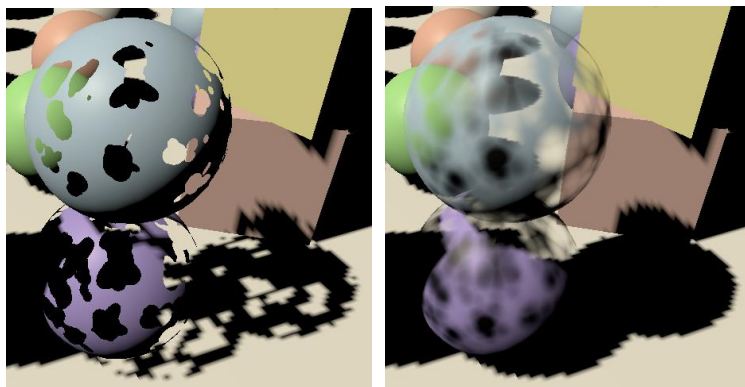
```

1. float cullingFactor =
2.     Mathf.Max(0f, 0.8f - settings.directional.cascadeFade);
3.
4. for (int i = 0; i < cascadeCount; i++) {
5.     ...
6.     splitData.shadowCascadeBlendCullingFactor = cullingFactor;
7.     ...
8. }

```

## 5. 透明

我们将用考虑透明阴影投射器来结束本篇教程。镂空，淡入淡出和透明材质都可以像不透明材质一样获取阴影，但目前只有镂空材质正确地投射了阴影。透明物体表现得与实心阴影投射器一样。



镂空和透明材质的阴影

### 5.1. 阴影模式

有好几种方法来修改阴影投射器。因为我们的阴影写入深度缓冲区，所以是二进制的，要么存在要么不存在，但依然给我们一些灵活性。它们可以打开并表现为完全实心，镂空，抖动或者完全关闭。为了提供最大的灵活性，我们可以独立于其他的材质属性来进行操作。所以我们新建一个单独的\_Shadows 着色器属性。我们可以使用 KeywordEnum 属性来创建一个关键字下拉菜单，其中 On 为默认值。

```

1. [KeywordEnum(On, Clip, Dither, Off)] _Shadows ("Shadows", Float) = 0

```



开启阴影

为这些模式添加一个新的着色器功能，替代已经存在的\_CLIPPPING 功能。我们只需要三个变体，无关键字对应 on 和 off，\_SHADOWS\_CLIP 和 \_SHADOWS\_DITHER。

```
1. //pragma shader_feature _CLIPPING
2. #pragma shader_feature _ _SHADOWS_CLIP _SHADOWS_DITHER
```

在 CustomShaderGUI 里创建一个 shadows 的 setter 属性。

```
1. enum ShadowMode {
2.     On, Clipped, Dithered, Off
3. }
4.
5. ShadowMode Shadows {
6.     set {
7.         if (SetProperty("_Shadows", (float)value)) {
8.             SetKeyword("_SHADOWS_CLIP", value == ShadowMode.Clip);
9.             SetKeyword("_SHADOWS_DITHER", value == ShadowMode.Dither);
10.        }
11.    }
12. }
```

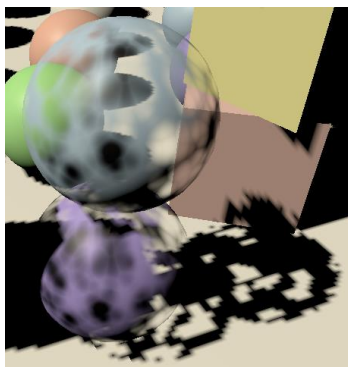
然后在预设方法里设置合适的阴影模式。On 对应 Opaque，clip 对应 clip，然后使用 dither 对应淡入淡出和透明。

## 5.2. 镂空阴影

在 ShadowCasterPassFragment 里将\_CLIPPPED 的检测替换为\_SHADOWS\_CLIP。

```
1. #if defined(_SHADOWS_CLIP)
2.     clip(base.a - UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Cutoff));
3. #endif
```

现在让透明材质显示镂空阴影，这可能适用于大部分不透明或者透明带需要 alpha 混合的表面。



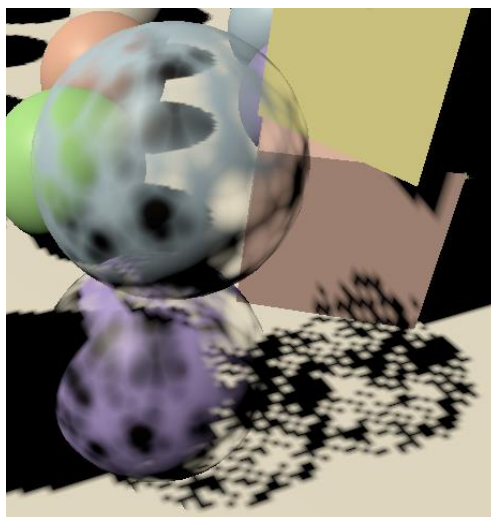
带有镂空阴影的透明对象

需要注意的是，镂空阴影并不像实心阴影那样稳定，因为当视角移动时，阴影矩阵会发生变化，这会导致片元略微变动。这会导致阴影贴图的纹素突然从镂空过渡到非镂空。

### 5.3. 抖动阴影

抖动阴影的原理与镂空阴影很想，除了标准不同。这种情况下我们从表面 alpha 里减去一个 dither 值，再以它为基础进行裁剪。我们可以再次使用 InterleavedGradientNoise 方法。

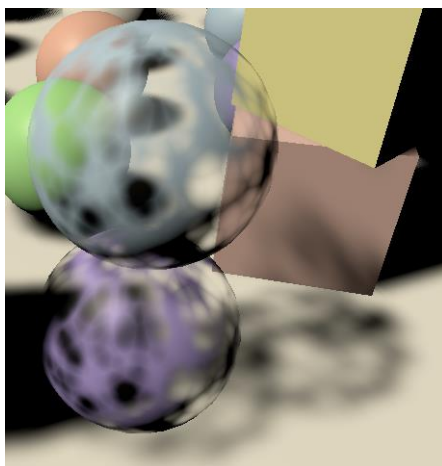
```
1. #if defined(_SHADOWS_CLIP)
2.     clip(base.a - UNITY_ACCESS_INSTANCED_PROP(UnityPerMaterial, _Cutoff));
3. #elif defined(_SHADOWS_DITHER)
4.     float dither = InterleavedGradientNoise(input.positionCS.xy, 0);
5.     clip(base.a - dither);
6. #endif
```



抖动阴影

抖动可用于近似半透明的阴影投射器，但这是一种很粗糙的方法。坚硬的抖动阴影看起来很糟，但是当使用较大的 PCF 滤镜时，它看起来似乎可以接受。





PCF7x7 下的抖动阴影

因为抖动模式是基于纹素固定，所以充电的半透明阴影投射器不会投射出更深的组合阴影。该效果与完全不透明的阴影投射器一样强。同样地，由于生成的图案有噪声，因此当阴影矩阵发生变化时，它会受到很多暂时的伪影影响，这会使阴影看起来在抖动。只要对象不移动，这个方法对于其他使用固定投影的光源类型表现更好，通常更使用的做法是对所有半透明物体都使用镂空阴影或者完全不使用阴影。

## 5.4. 无阴影

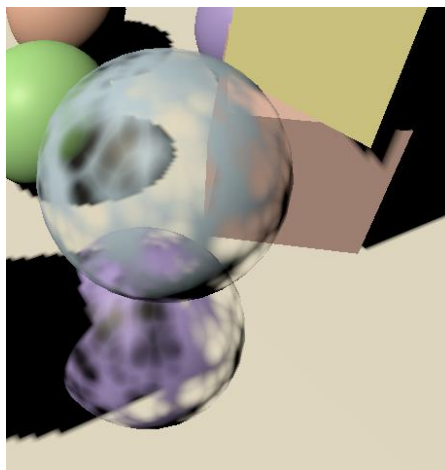
可以通过调整每个对象 MeshRenderer 组件的 Cast Shadows 设置来逐对象关闭阴影投射。然而如果你想禁用使用相同材质的所有对象的阴影，这就不太可行。所以我们将支持逐材质禁用阴影。我们通过禁用材质的 ShadowCaster Pass 来实现。

在 CustomShaderGUI 里添加一个 SetShadowCasterPass 方法，首先检查着色器属性 \_Shadows 是否存在。如果存在，再通过 hasMixedValue 属性检查是否所有选中的材质都使用了相同的模式。如果没有该模式或者是混合模式，那么就终止。否则，调用 SetShaderPassEnabled 并传入 pass 名称和启用状态作为参数，将所有材质的 ShadowCaster Pass 启用或禁用。

```
1. void SetShadowCasterPass () {  
2.     MaterialProperty shadows = FindProperty("_Shadows", properties, false);  
3.     if (shadows == null || shadows.hasMixedValue) {  
4.         return;  
5.     }  
6.     bool enabled = shadows.floatValue < (float)ShadowMode.Off;  
7.     foreach (Material m in materials) {  
8.         m.SetShaderPassEnabled("ShadowCaster", enabled);  
9.     }  
10. }
```

确保 pass 被正确设置的最简单方法是当材质在 GUI 里被改变时，始终调用 SetShadowCasterPass。我们可以在 OnGUI 开头调用 EditorGUI.BeginChangeCheck 并在结尾调用 EditorGUI.EndChangeCheck。后一个方法返回了从开始检查以来是否有内容被改变。如果有，那么设置 shadowCaster pass。

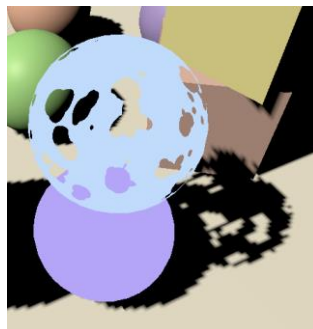
```
1. public override void OnGUI (  
2.     MaterialEditor materialEditor, MaterialProperty[] properties  
3. ) {  
4.     EditorGUI.BeginChangeCheck();  
5.     ...  
6.     if (EditorGUI.EndChangeCheck()) {  
7.         SetShadowCasterPass();  
8.     }  
9. }
```



不投射阴影

## 5.5. 不受光阴影投射器

尽管不受光材质不会被光照影响，但你可能想让它们投射阴影。我们可以通过简单地将 ShadowCaster Pass 从 Lit 复制到 UnLit 着色器里来支持阴影。



不受光但投射阴影

## 5.6. 接收阴影

最后，我们也可以设置受光表面忽略阴影，这可能对于类似全息图像的功能有用，或者仅仅为了美术目的。在 Lit 里为此添加一个\_RECEIVE\_SHADOWS 关键字的开关属性。

```
1. [Toggle(_RECEIVE_SHADOWS)] _ReceiveShadows ("Receive Shadows", Float) = 1
```

在 CustomLit pass 里添加对应的着色器功能。

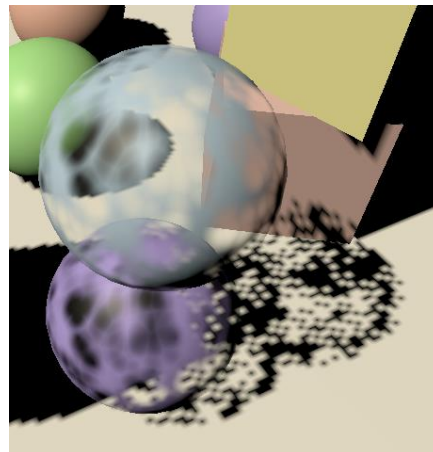
```
1. #pragma shader_feature _RECEIVE_SHADOWS
```



接收阴影

我们需要做的只是当该关键字定义时，强制将 GetDirectionalShadowAttenuation 里的阴影衰减设置为 1。

```
1. float GetDirectionalShadowAttenuation (...) {  
2.     #if !defined(_RECEIVE_SHADOWS)  
3.         return 1.0;  
4.     #endif  
5.     ...  
6. }
```



投射但不接收阴影

下一篇教程是烘焙光。