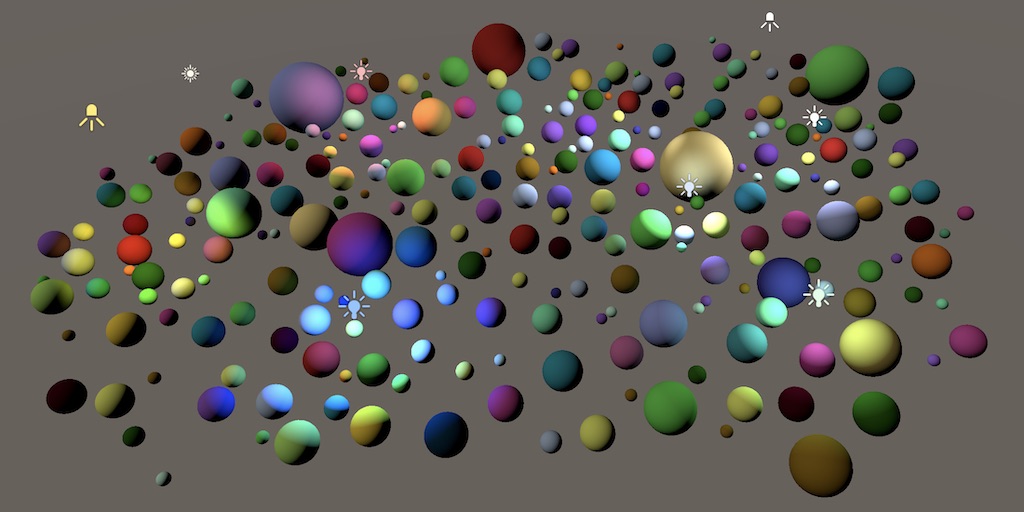
**光照-单pass前向渲染**

* **漫反射着色**
* **支持平行光、点光源、聚光灯**
* **最多支持16个可见的光源**
* **每个物体最多支持4个逐像素计算光源和4个逐顶点计算光源**

这是Unity可编程渲染管线系列教程的第三章。这章我们将添加对漫反射光照的支持，实现在一个draw call中使每个物体着色最多支持八个光源。

本教程使用Unity 2018.3.0f2完成。



**一、着色器中添加光照**

为了支持光照，我们必须为我们的管线添加一个光照着色器。照明复杂性可以从非常简单的 仅包括漫射光到非常复杂的基于物理的阴影光照。同时支持非真实感渲染，比如卡通渲染。我们将从最简单的光照着色器开始，该着色器仅计算漫反射方向光照，没有阴影。

**1.1光照着色器**

复制"Unlit.hlsl"文件并将其重命名为"Lit.hlsl"，将新文件中所有的"unlit"替换为"lit"，特别是包含定义以及顶点和片元函数名。

#ifndef MYRP\_LIT\_INCLUDED

#define MYRP\_LIT\_INCLUDED

…

VertexOutput LitPassVertex (VertexInput input) {

…

}

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

…

}

#endif // MYRP\_LIT\_INCLUDED

然后复制"Unlit.shader"文件并将其重命名为"Lit.shader"，再次将新文件中所有的"unlit"替换为"lit"。

Shader "My Pipeline/Lit" {

Properties {

\_Color ("Color", Color) = (1, 1, 1, 1)

}

SubShader {

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

}

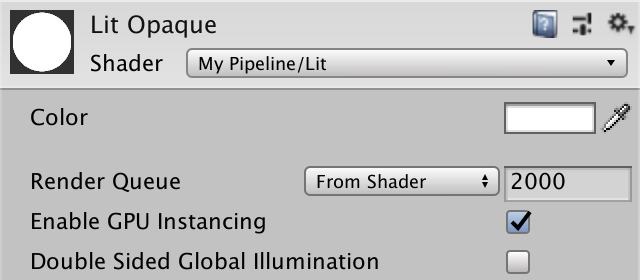
}

}

我们不应该显式使用lit pass吗？

因为我们的管线依然非常的基础，我们还不用为专门使用pass而操心。

现在我们要为新的光照着色器创建一个非透明体材质球，即使它仍然和非光照的着色器没什么区别。



**1.2 法向量**

为了计算平行光的影响，我们需要知道表面法线。所以我们必须在顶点输入和输出结构中添加法向量。有关如何计算光照的详细说明，请参见 [Rendering 4, The First Light](https://catlikecoding.com/unity/tutorials/rendering/part-4/)。

struct VertexInput {

float4 pos : POSITION;

float3 normal : NORMAL;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

struct VertexOutput {

float4 clipPos : SV\_POSITION;

float3 normal : TEXCOORD0;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

在LitPassVertex中将法线从对象空间转换到世界空间。因为我们假设只使用均匀缩放，因此可以简单地使用模型矩阵3x3的部分，接着在LitPassFragment的每个片元中归一化。对支持非均匀缩放要求我们使用世界到对象矩阵的转置矩阵。

VertexOutput LitPassVertex (VertexInput input) {

…

output.normal = mul((float3x3)UNITY\_MATRIX\_M, input.normal);

return output;

}

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

UNITY\_SETUP\_INSTANCE\_ID(input);

input.normal = normalize(input.normal);

…

}

为了验证我们最终是正确的法向量，将它们用到最后的颜色上。但还是保持追踪材质的颜色，我们之后将用于反照度。

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

UNITY\_SETUP\_INSTANCE\_ID(input);

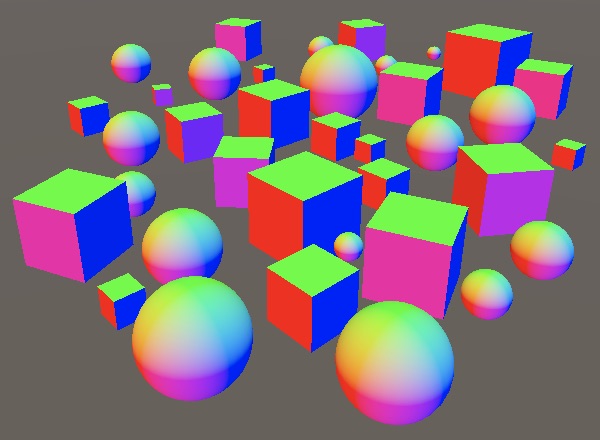
input.normal = normalize(input.normal);

float3 albedo = UNITY\_ACCESS\_INSTANCED\_PROP(PerInstance, \_Color).rgb;

float3 color = input.normal;

return float4(color, 1);

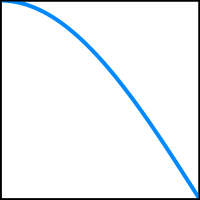
}



显示原生世界空间法向量。

1.3 漫反射光照

漫反射光照的贡献取决于光击中表面时的角度，这可以通过表面法线和光线过来的方向的点积运算得到，丢弃负值结果。在这个案例中的方向光的向量是常数。现在让我们硬编码方向，让它直指向上。乘上漫反射光照和反照度获得最终的颜色。



漫反射光照从0°到90°入射角度衰减。

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

UNITY\_SETUP\_INSTANCE\_ID(input);

input.normal = normalize(input.normal);

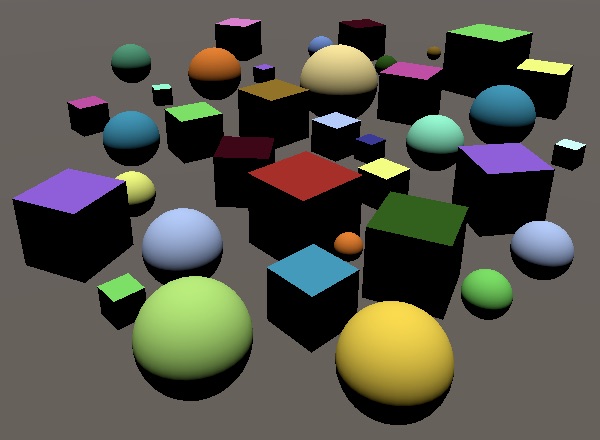
float3 albedo = UNITY\_ACCESS\_INSTANCED\_PROP(PerInstance, \_Color).rgb;

float3 diffuseLight = saturate(dot(input.normal, float3(0, 1, 0)));

float3 color = diffuseLight \* albedo;

return float4(color, 1);

}



来自上面的漫反射光照。

1. 可见的光源

为了使用场景中定义的光源，我们的管线需要发送光源数据到GPU。一个场景中可能有多个光源，所以我们也应支持多光源。这有很多种方法去做。Unity默认的管线是对每个对象的独立pass中渲染每一个光源。轻量级管线对每个对象的一个pass中渲染所有的光照。而高清管线使用延迟渲染，就是先渲染所有对象的表面数据，接着对每个光源用一个pass渲染。

我们现在使用和轻量级管线相同的方法，因此每个对象都渲染一次，并考虑所有的光源。我们通过发送所有的可见的光源的数据到GPU来完成。场景中的光源不影响任何东西的话，在渲染中就会被忽略。

* 1. 光照缓存

在一个pass中渲染所有的光源意味着在同一时刻所有的光照数据都要可获得。只是现在限制我们仅使用平行光，这也就是说我们需要的就是每个光源的颜色和方向。为了支持任意数量的光源，我们使用数组去存储这些数据，我们将把它们放在一个名为\_LightBuffer的独立缓存中。数组在shader中的定义像是在C#中，除了括号是在变量名后而不是在类型后。

CBUFFER\_START(UnityPerDraw)

float4x4 unity\_ObjectToWorld;

CBUFFER\_END

CBUFFER\_START(\_LightBuffer)

float4 \_VisibleLightColors[];

float4 \_VisibleLightDirections[];

CBUFFER\_END

然而，我们不可以定义任意长度的数组。数组的定义必须立即声明他的长度。让我们使用一个长度为4的数组。这就意味着我们可以支持最多一次计算四个可见光源。使用宏来定义限制更方便引用。

#define MAX\_VISIBLE\_LIGHTS 4

CBUFFER\_START(\_LightBuffer)

float4 \_VisibleLightColors[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightDirections[MAX\_VISIBLE\_LIGHTS];

CBUFFER\_END

在光照缓存的后面，添加一个DiffuseLight函数，它使用光源数据来处理光照计算。它需要一个光源索引和法向量作为参数，从数组中提取相关的数据，然后运行漫反射光照计算并返回它与光源颜色的调制结果。

CBUFFER\_START(\_LightBuffer)

float4 \_VisibleLightColors[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightDirections[MAX\_VISIBLE\_LIGHTS];

CBUFFER\_END

float3 DiffuseLight (int index, float3 normal) {

float3 lightColor = \_VisibleLightColors[index].rgb;

float3 lightDirection = \_VisibleLightDirections[index].xyz;

float diffuse = saturate(dot(normal, lightDirection));

return diffuse \* lightColor;

}

在LitPassFragment中，使用一个for循环对每个光源调用一次新建的函数，累计影响片元的总漫反射光照。

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

…

float3 diffuseLight = 0;

for (int i = 0; i < MAX\_VISIBLE\_LIGHTS; i++) {

diffuseLight += DiffuseLight(i, input.normal);

}

float3 color = diffuseLight \* albedo;

return float4(color, 1);

}

注意即使我们使用循环，shader编译器也可能会展开它。随着我们shader变得更复杂，有时编译器会切换到使用真正的循环。

* 1. 填充缓存

目前我们最后得到的是全黑的形状，因为我们还没有传递任何光源数据到GPU。我们需要添加一些数组到MyPipeline，使用相同的长度。并且，使用静态的Shader.PropertyToID方法找到shader属性相应的标识。shader标识在每个session中是常量，因此可以被存储为静态变量。

const int maxVisibleLights = 4;

static int visibleLightColorsId =

Shader.PropertyToID("\_VisibleLightColors");

static int visibleLightDirectionsId =

Shader.PropertyToID("\_VisibleLightDirections");

Vector4[] visibleLightColors = new Vector4[maxVisibleLights];

Vector4[] visibleLightDirections = new Vector4[maxVisibleLights];

为什么不使用Color数组？

这里没有方法直接传递color数组到GPU。Vector4数组是最好的选择并且匹配shader的数据格式。我们可以直接分配颜色到数组中，因为存在隐式的转换从Color到Vector4。

数组可以在命令缓存中通过调用SetGlobalVectorArray方法复制到GPU中，然后执行它。由于我们早已有了cameraBuffer，让我们在开始Render Camera采样时使用它。

cameraBuffer.BeginSample("Render Camera");

cameraBuffer.SetGlobalVectorArray(

visibleLightColorsId, visibleLightColors

);

cameraBuffer.SetGlobalVectorArray(

visibleLightDirectionsId, visibleLightDirections

);

context.ExecuteCommandBuffer(cameraBuffer);

cameraBuffer.Clear();

* 1. 配置光源

我们现在每一帧都要传递光源数据到GPU中，但是它依然是默认的数据，因此对象还是黑色。我们必须要在复制向量之前配置光源。让我们委托它到一个新的ConfigureLights方法中。

cameraBuffer.ClearRenderTarget(

(clearFlags & CameraClearFlags.Depth) != 0,

(clearFlags & CameraClearFlags.Color) != 0,

camera.backgroundColor

);

ConfigureLights();

cameraBuffer.BeginSample("Render Camera");

在剔除过程中，Unity也会计算出哪些光源是可见的。这个信息经visibleLights列表可获得，它是剔除结果的一部分。这个列表的元素是VisibleLight结构体，它包含了我们所需的所有数据。创建需要的ConfigureLights方法并让它循环遍历列表。

void ConfigureLights () {

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

VisibleLight light = cull.visibleLights[i];

}

}

VisibleLight.finalColor字段保存了光源的颜色。光源的颜色乘上了他的强度，并且也转换到了正确的色彩空间中。因此我们可以直接复制它到visibleLightColors中，使用相同的索引。

VisibleLight light = cull.visibleLights[i];

visibleLightColors[i] = light.finalColor;

当然，由于默认的Unity认为光源的强度是定义在伽马空间中的，即使我们工作在线性空间中也是如此。这是Unity默认渲染管线的遗留问题。新的管线认为它是一个线性值。这个行为通过布尔值GraphicsSettings.lightsUseLinearIntensity属性来控制。它是个项目设置，但是只能通过代码来调整。我们只需要设置一次，因此在MyPipeline的构造方法中完成。

public MyPipeline (bool dynamicBatching, bool instancing) {

GraphicsSettings.lightsUseLinearIntensity = true;

…

}

改变这个设置只会影响编辑器因为它不会自动重新应用图形设置。进入并退出运行模式才会应用它。

除此之外，平行光的方向由旋转值决定。光源沿着它的局部Z轴发光。我们可以通过VisibleLight.localtoWorld矩阵字段在世界空间中找到这个向量。矩阵的第三列定义了变换后的局部Z方向向量，这个我们可以通过Matrix4x4.GetColumn方法来获得，使用索引2作为参数。

上面给了我们光源发光的方向，但是在shader中我们使用的方向是从表面指向光源。因此我们在分配向量到visibleLightDirections之前必须要对它取反。由于方向向量的第四个分量总是为零，我们只需要对X、Y和Z取反。

VisibleLight light = cull.visibleLights[i];

visibleLightColors[i] = light.finalColor;

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

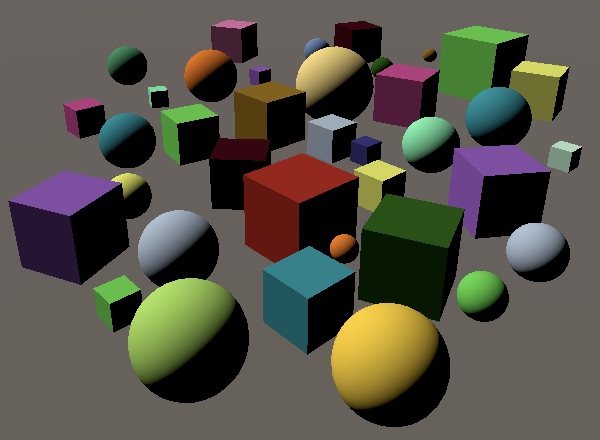
v.x = -v.x;

v.y = -v.y;

v.z = -v.z;

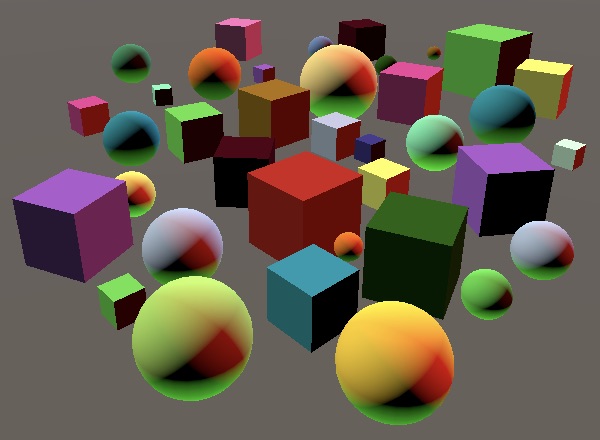
visibleLightDirections[i] = v;

我们的对象现在由颜色以及主要的平行光方向着色，假设你在场景中没有其他的光源。如果场景中没有光源的话，仅添加一个平行光就行了。



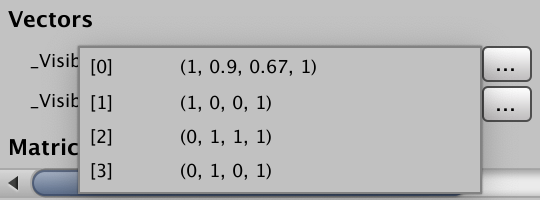
单个平行光的漫反射着色。

但是我们的shader总是计算四个光源的贡献值，即使场景中只有一个光源。因此你可以添加三个平行光源，而且它也不会拖慢GPU。



四个平行光源。

你可以通过frame debugger检查发送到GPU的光源数据。选择使用我们的shader的其中一个draw call，然后展开向量数组看它们的内容。



通过frame debugger找到光源颜色。

* 1. 可变的光源数量

当使用明确的四个平行光源一切都按着预期在运行。我们甚至可以使用更多的光源，只要在同一时间只有四个光源是可见的。但是当超过四个可见光源时我们的管线将会出现数组越界的异常。我们仅支持最多四个可见光源，但是Unity在剔除时不会考虑这些。因此visibleLights最后可以获得比我们数组更多的元素。当我们超出最大限度时必须终止循环。这意味着简单地忽略了一些可见光源。

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

if (i == maxVisibleLights) {

break;

}

VisibleLight light = cull.visibleLights[i];

…

}

哪些光源被省略了呢？

我们简单的跳过了visibleLights列表最后的光源。这个光源的顺序基于多个方面的条件，包括光源类型、强度和它们是否开启阴影。你可以假设光源由重要至不重要排序。例如，带有最大强度并开启阴影的平行光会使第一个元素。

当可见光源数量减少时会发生另一个奇怪的事情。它们依然时可见的，因为我们没有重设它们的数据。我们可以通过在结束可见光源之后继续循环遍历数组，清除那些不再用的光源的颜色来解决这个问题。

int i = 0;

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

…

}

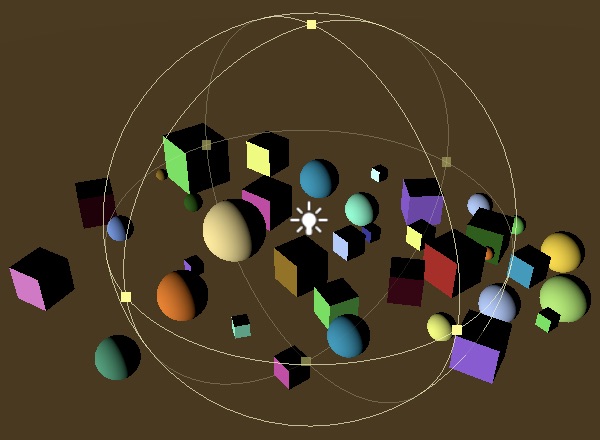
for (; i < maxVisibleLights; i++) {

visibleLightColors[i] = Color.clear;

}

3 点光源

我们当前仅支持了平行光，但是通常来说一个场景只会有一个平行光加上额外的点光源。虽然我们可以添加点光源到场景中，但是它们现在被当成是平行光。我们现在就去修复它。



点光源被当作平行光。

[Rendering 5, Multiple Lights](https://catlikecoding.com/unity/tutorials/rendering/part-5/) 描述了点光源和聚光灯，但是使用的是Unity默认管线的旧方法。我们现在要使用和轻量级管线一样的方法。

3.1 光源位置

不像平行光，点光源的位置很重要。我们要将方向和位置数据存储在同一个数组中，而不是为位置添加独立的数组，每个元素包含一个方向或是位置。相应地在MyPipeline中重命名变量。

static int visibleLightColorsId =

Shader.PropertyToID("\_VisibleLightColors");

static int visibleLightDirectionsOrPositionsId =

Shader.PropertyToID("\_VisibleLightDirectionsOrPositions");

Vector4[] visibleLightColors = new Vector4[maxVisibleLights];

Vector4[] visibleLightDirectionsOrPositions = new Vector4[maxVisibleLights];

ConfigureLights可以使用VisibleLight.lightType去检查每个光源的类型。在方向光的情况下，存储方向是正确的。另一种情况就是存储光源的世界位置，这可以从局部到世界矩阵的第四列提取出来。

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;

}

else {

visibleLightDirectionsOrPositions[i] =

light.localToWorld.GetColumn(3);

}

在shader中也要重命名数组。在DiffuseLight，首先假设我们依然处理的是平行光。

CBUFFER\_START(\_LightBuffer)

float4 \_VisibleLightColors[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightDirectionsOrPositions[MAX\_VISIBLE\_LIGHTS];

CBUFFER\_END

float3 DiffuseLight (int index, float3 normal) {

float3 lightColor = \_VisibleLightColors[index].rgb;

float4 lightPositionOrDirection = \_VisibleLightDirectionsOrPositions[index];

float3 lightDirection = lightPositionOrDirection.xyz;

float diffuse = saturate(dot(normal, lightDirection));

return diffuse \* lightColor;

}

但是如果我们处理点光源，就必须要自己计算光源方向。首先，光源位置减去表面位置，这要求我们添加一个额外的参数到函数中。这为我们提供了一个世界空间的光源向量，可以通过归一化它来获得方向。

float3 DiffuseLight (int index, float3 normal, float3 worldPos) {

float3 lightColor = \_VisibleLightColors[index].rgb;

float4 lightPositionOrDirection = \_VisibleLightDirectionsOrPositions[index];

float3 lightVector =

lightPositionOrDirection.xyz - worldPos;

float3 lightDirection = normalize(lightVector);

float diffuse = saturate(dot(normal, lightDirection));

return diffuse \* lightColor;

}

这些是为点光源做的，对方向光来说没有意义。我们可以用相同的计算来支持两者，通过世界位置乘上光源的方向或位置的W分量。如果它是一个位置向量，那么W就是1即不会改变计算。但是如果它是方向向量，那么W就是0，这样就消除可减法。因此我们最后归一化原生的方向向量的话，它不会有什么不同。这确实对平行光来说引入了不必要的归一化，但是不值得为避免这种情况而分支。

lightPositionOrDirection.xyz - worldPos \* lightPositionOrDirection.w;

为实现这些，在LitPassFragment中需要知道片元的世界空间位置。我们在LitPassVertex中就有了位置，因此将它添加到输出中并一同发送出去。

struct VertexOutput {

float4 clipPos : SV\_POSITION;

float3 normal : TEXCOORD0;

float3 worldPos : TEXCOORD1;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

VertexOutput LitPassVertex (VertexInput input) {

…

output.worldPos = worldPos.xyz;

return output;

}

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

…

float3 diffuseLight = 0;

for (int i = 0; i < MAX\_VISIBLE\_LIGHTS; i++) {

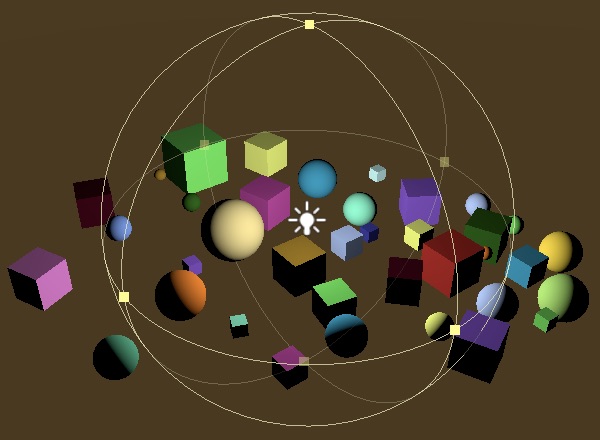
diffuseLight += DiffuseLight(i, input.normal, input.worldPos);

}

float3 color = diffuseLight \* albedo;

return float4(color, 1);

}



正确的光源方向。

3.2 距离衰减

除了被认为是无限远的方向光外，光源的强度会随着距离减弱。这个关系为i/d2，其中i是光源规定的强度，d是光源与表面之间的距离。这被称之为平方反比定律。因此我们需要让最终的漫反射贡献度除以光源向量的平方。为了避免被零除，我们强制设置一个最小的平方距离。

float diffuse = saturate(dot(normal, lightDirection));

float distanceSqr = max(dot(lightVector, lightVector), 0.00001);

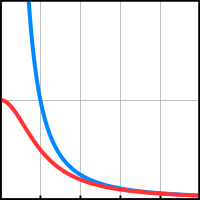
diffuse /= distanceSqr;

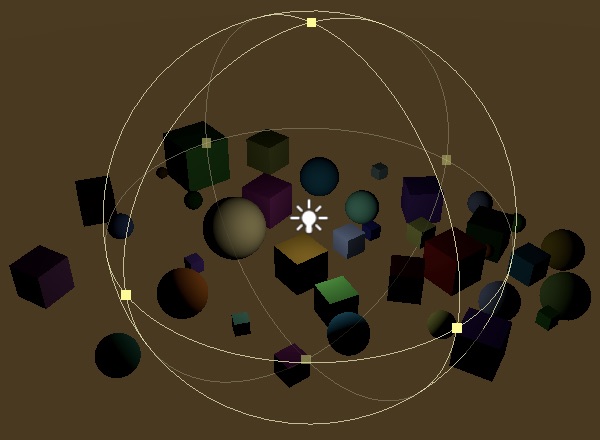
return diffuse \* lightColor;

这不会增加非常接近点光源时的强度吗？

的确，当d小于1时光源的强度会增加。当d接近最小值时强度会变得非常巨大。

Unity的默认管线使用i/(1+d2)来避免增加亮度，但是这不真实并且靠近光源又太暗了。轻量级管线最初使用相同的衰减方式，但是从3.3.0版本开始它使用正确的平方衰减方式。

1/d2 和 1/（1+d2）



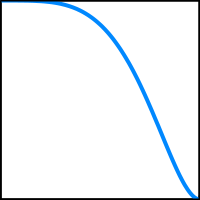
光线随着距离暗淡。

由于对平行光来说光源向量和方向向量相同，最后距离平方就为1。这意味着平行光不会受距离衰减影响，这是对的。

3.3 光照范围

点光源还有个配置范围，这限制了它的影响范围。超过范围就不会受光源影响，即使它依然可以照亮对象。这并不真实，但是这样允许对光源有更好的控制且限制了被光源所影响的对象数量。没有这个范围限制的话，每个光源都会被视为可见的。

这个范围限制也不是突然被截断。相反，光源的强度会基于平方距离平滑地淡出。轻量级管线和光照贴图使用（1-（d2/r2）2）2公式，其中r是光源的半径。我们会使用一样的的淡出曲线。



范围淡出曲线

光源范围是场景数据的一部分，因此需要传递每个光源的范围到GPU中。我们将为衰减数据使用另一个数组。虽然浮点数组就满足所需，但是我们再一次使用向量数组，因为我们之后将包含更多的数据。

static int visibleLightColorsId =

Shader.PropertyToID("\_VisibleLightColors");

static int visibleLightDirectionsOrPositionsId =

Shader.PropertyToID("\_VisibleLightDirectionsOrPositions");

static int visibleLightAttenuationsId =

Shader.PropertyToID("\_VisibleLightAttenuations");

Vector4[] visibleLightColors = new Vector4[maxVisibleLights];

Vector4[] visibleLightDirectionsOrPositions = new Vector4[maxVisibleLights];

Vector4[] visibleLightAttenuations = new Vector4[maxVisibleLights];

还要在Render中复制新数组到GPU。

cameraBuffer.SetGlobalVectorArray(

visibleLightDirectionsOrPositionsId, visibleLightDirectionsOrPositions

);

cameraBuffer.SetGlobalVectorArray(

visibleLightAttenuationsId, visibleLightAttenuations

);

并将它填充到ConfigureLights中。平行光没有范围限制，因此它们可以使用零向量。对于点光源的情况，我们将它的范围放到向量的X分量中。但不是直接存储范围，我们通过存储1/r2来减少shader必须做的工作，并且避免了被零除。

Vector4 attenuation = Vector4.zero;

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

…

}

else {

visibleLightDirectionsOrPositions[i] =

light.localToWorld.GetColumn(3);

attenuation.x = 1f /

Mathf.Max(light.range \* light.range, 0.00001f);

}

visibleLightAttenuations[i] = attenuation;

添加新数组到shader，计算除由范围造成的淡出并将其作为最终漫反射贡献度的因子。

CBUFFER\_START(\_LightBuffer)

float4 \_VisibleLightColors[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightDirectionsOrPositions[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightAttenuations[MAX\_VISIBLE\_LIGHTS];

CBUFFER\_END

float3 DiffuseLight (int index, float3 normal, float3 worldPos) {

float3 lightColor = \_VisibleLightColors[index].rgb;

float4 lightPositionOrDirection = \_VisibleLightDirectionsOrPositions[index];

float4 lightAttenuation = \_VisibleLightAttenuations[index];

float3 lightVector =

lightPositionOrDirection.xyz - worldPos \* lightPositionOrDirection.w;

float3 lightDirection = normalize(lightVector);

float diffuse = saturate(dot(normal, lightDirection));

float rangeFade = dot(lightVector, lightVector) \* lightAttenuation.x;

rangeFade = saturate(1.0 - rangeFade \* rangeFade);

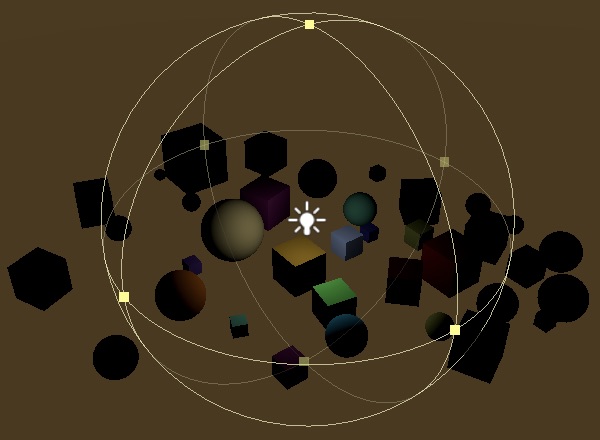
rangeFade \*= rangeFade;

float distanceSqr = max(dot(lightVector, lightVector), 0.00001);

diffuse \*= rangeFade / distanceSqr;

return diffuse \* lightColor;

}



光源基于范围的淡出。

平行光再次不受影响，因为它们的lightAttenuation.x总是0，所以rangeFade就为1。

4 聚光灯

轻量级管线也支持聚光灯，因此我们也添加它。聚光灯工作方式像点光源，但是被限制在锥形里而不是向各个方向发光。

4.1 聚光方向

就像方向光一样，聚光灯沿着它的局部Z轴发光，但是在一个锥体里。并且它也有一个位置，这意味着我们必须同时提供位置和方向给聚光灯。因此添加一个额外的数组给聚光灯方向在MyPipeline中。

static int visibleLightAttenuationsId =

Shader.PropertyToID("\_VisibleLightAttenuations");

static int visibleLightSpotDirectionsId =

Shader.PropertyToID("\_VisibleLightSpotDirections");

Vector4[] visibleLightColors = new Vector4[maxVisibleLights];

Vector4[] visibleLightDirectionsOrPositions = new Vector4[maxVisibleLights];

Vector4[] visibleLightAttenuations = new Vector4[maxVisibleLights];

Vector4[] visibleLightSpotDirections = new Vector4[maxVisibleLights];

…

void Render (ScriptableRenderContext context, Camera camera) {

…

cameraBuffer.SetGlobalVectorArray(

visibleLightAttenuationsId, visibleLightAttenuations

);

cameraBuffer.SetGlobalVectorArray(

visibleLightSpotDirectionsId, visibleLightSpotDirections

);

…

}

在ConfigureLights中，当没有处理平行光时，还要检查光源是否为聚光灯。如果是，设置方向向量，就像是平行光，但是分配它到visibleLightSpotDirection中。

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;

}

else {

visibleLightDirectionsOrPositions[i] =

light.localToWorld.GetColumn(3);

attenuation.x = 1f /

Mathf.Max(light.range \* light.range, 0.00001f);

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

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

v.x = -v.x;

v.y = -v.y;

v.z = -v.z;

visibleLightSpotDirections[i] = v;

}

}

还要添加新数据到shader。

CBUFFER\_START(\_LightBuffer)

float4 \_VisibleLightColors[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightDirectionsOrPositions[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightAttenuations[MAX\_VISIBLE\_LIGHTS];

float4 \_VisibleLightSpotDirections[MAX\_VISIBLE\_LIGHTS];

CBUFFER\_END

float3 DiffuseLight (int index, float3 normal, float3 worldPos) {

float3 lightColor = \_VisibleLightColors[index].rgb;

float4 lightPositionOrDirection = \_VisibleLightDirectionsOrPositions[index];

float4 lightAttenuation = \_VisibleLightAttenuations[index];

float3 spotDirection = \_VisibleLightSpotDirections[index].xyz;

…

}

4.2 角度衰减

聚光灯的锥体是特殊的，带有小于180°的正角度。我们可以通过点乘聚光灯方向和光线方向来确定表面点是否在锥体内部。如果结果最大是配置的聚光灯角度的一半的余弦，那么这个片元就受光的影响。

锥体的边缘没有被突然截断。相反，这有光线淡出的传输范围。这个范围可以通过光线开始淡出的内部聚光角度和光线强度接近于零的外部聚光角度来定义。然而Unity的聚光灯只允许我们设置外部聚光角度。Unity的默认管线使用一个light cookie来明确衰减，然而轻量级管线使用一个假设内角和外角关系为固定的平滑函数来计算衰减。

为了明确衰减，首先，将聚光角的一半由角度转为弧度，然后计算它的余弦。通过VisibleLight.spotAngle可获得配置角度。

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

…

float outerRad = Mathf.Deg2Rad \* 0.5f \* light.spotAngle;

float outerCos = Mathf.Cos(outerRad);

}

轻量级管线和光照贴图使用tan(ri)=46/64tan(ro)关系定义内角，其中ri和ro是是内部聚光角度和外部聚光角度的弧度制半角。我们需要使用内部角度的余弦因此最后的关系是cos（ri）=cos（arctan（46/64tan(ro)））

float outerRad = Mathf.Deg2Rad \* 0.5f \* light.spotAngle;

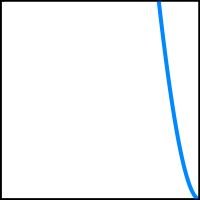
float outerCos = Mathf.Cos(outerRad);

float outerTan = Mathf.Tan(outerRad);

float innerCos =

Mathf.Cos(Mathf.Atan(((46f / 64f) \* outerTan));

基于角度的衰减定义为限制到0-1然后平方，其中Ds\*Dl是聚光灯方向和光线方向点乘的结果。

对90°聚光灯来说衰减从0°到45°。

这个表达式可以简化为(Ds\*Dl)a+b，对于a=1/(cos(ri)-cos(ro))且b=-cos(ro)a。这运行我们计算a和b在ConfigureLights中并且存储它们到衰减数据向量的后两位分量中。

float outerRad = Mathf.Deg2Rad \* 0.5f \* light.spotAngle;

float outerCos = Mathf.Cos(outerRad);

float outerTan = Mathf.Tan(outerRad);

float innerCos =

Mathf.Cos(Mathf.Atan(((64f - 18f) / 64f) \* outerTan));

float angleRange = Mathf.Max(innerCos - outerCos, 0.001f);

attenuation.z = 1f / angleRange;

attenuation.w = -outerCos \* attenuation.z;

在shader中，聚光灯淡出因子可以在之后被点乘、相乘、相加、satruation和最终的平方计算。然后使用结果去和漫反射光照调制。

float rangeFade = dot(lightVector, lightVector) \* lightAttenuation.x;

rangeFade = saturate(1.0 - rangeFade \* rangeFade);

rangeFade \*= rangeFade;

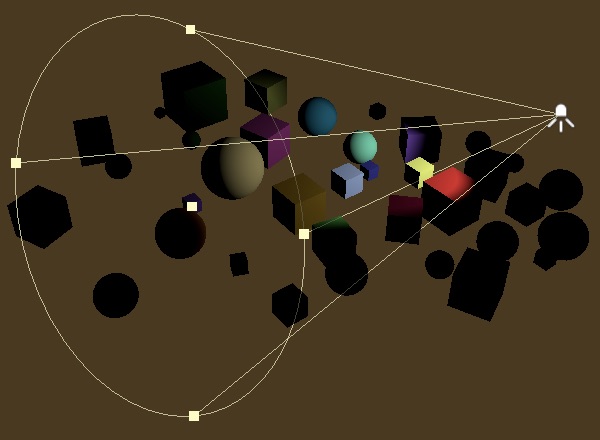
float spotFade = dot(spotDirection, lightDirection);

spotFade = saturate(spotFade \* lightAttenuation.z + lightAttenuation.w);

spotFade \*= spotFade;

float distanceSqr = max(dot(lightVector, lightVector), 0.00001);

diffuse \*= spotFade \* rangeFade / distanceSqr;



聚光灯，强度为4

为保证聚光灯淡出计算不受其他光源类型影响，设置它们的衰减向量W分量为1。

Vector4 attenuation = Vector4.zero;

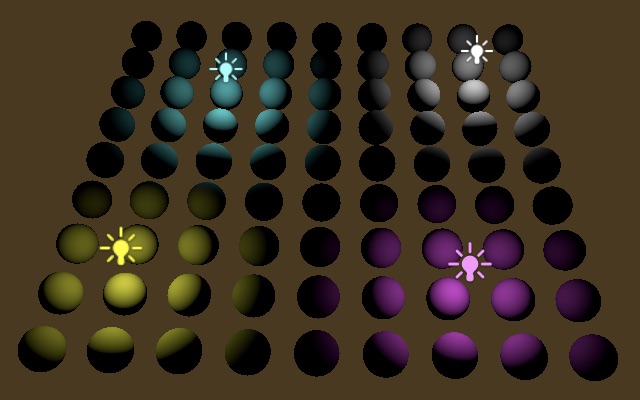
attenuation.w = 1f;

那么面光源呢？

Unity的轻量级和默认管线不支持实时面光源，我们也是不支持。面光源只在光照贴图中使用，我们将在之后支持。

5 逐对象光源

我们现在最多支持四个逐对象光源。事实上，我们总是对每个对象计算四个光源，即使当那些光源已不必要了。例如，考虑一个9x9的共81个球体的网格，在靠近角落的地方有四个点光源。如果这些光源的范围大致设置为网格的四分之一，那么大多数的球体最后只会受一个光源的影响，有些是两个，还有些一个也没有。



81个球体的网格，带有四个点光源。

现在81个球体被一次draw call所渲染—假设GPU实例化开启—但是光源贡献度被计算了四次在每个球体的片元着色器中。如果我们可以以某种方式让每个对象只计算所需要的光源，这样会更好些。这样我们还可以增加可见光源的支持数量。

5.1 光源索引

由于剔除的原因，Unity会确定那些可见的光源，这也包括算出哪些光源会影响对象。我们可以要求Unity发送这些信息到GPU中，以光源索引列表的形式。

Unity目前支持两种光源索引格式。第一种方法是存储最多八个索引在两个float4变量中，为每个对象设置。第二种方法是将多有对象的光源索引列表放在一个缓存中，就像GPU实列数据存储方式。然而，第二种方式在Unity 2018.3中是不可用的，只有前者是支持的。因此尽管这种方式不理想，我们目前也只能限制自己使用第一种方式。

我们通过设置绘制设置里的rendererConfiguration字段来将光源索引以float4字段形式设置到RenderConfiguration.PerObjectLightIndices8中。

var drawSettings = new DrawRendererSettings(

camera, new ShaderPassName("SRPDefaultUnlit")

) {

flags = drawFlags,

rendererConfiguration = RendererConfiguration.PerObjectLightIndices8

};

//drawSettings.flags = drawFlags;

drawSettings.sorting.flags = SortFlags.CommonOpaque;

Unity现在必须为每个对象设置额外的GPU数据，这会影响GPU实例化。Unity尝试将那些受同样光源影响的对象分为一组，但是更喜欢以距离来分组。还有，光源索引会基于每个对象的相对光源重要性进行排序，这会进一步分割批次。在这个网格案例中，我最终得到了30个draw call，这大大的超过了1，但是也远小于81.

索引可以通过unity\_4LightIndices0和unity\_4LightIndices1向量来获取，它们应该是UnityPerDraw缓存的一部分。除此之外，这还有untiy\_LightIndicesOffsetAndCount，它是另外一个float4向量。它的Y分量包含了影响对象的光源数量。它的X分量包含了一个当第二种方法启用时的偏移，因此我们可以忽略它。

CBUFFER\_START(UnityPerDraw)

float4x4 unity\_ObjectToWorld;

float4 unity\_LightIndicesOffsetAndCount;

float4 unity\_4LightIndices0, unity\_4LightIndices1;

CBUFFER\_END

现在我们可以限制自己仅在需要时调用DiffuseLight。但是我们需要检索正确的光源索引。因为我们现在支持最多四个可见光源，所以要用的就是unity\_4LightIndices0，我们可以像数组一样索引来检索它合适的分量。

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

int lightIndex = unity\_4LightIndices0[i];

diffuseLight +=

DiffuseLight(lightIndex, input.normal, input.worldPos);

}

虽然没有可见的改变—假设这里最多四个可见光源—GPU做的工作较少，因为它只计算相关光源的贡献度。你可以用frame debugger去检查在每次draw call中有多少个光源最终被使用。shader变得更复杂了，因为我们现在使用可变的循环而不是固定的。最终会有更好或更坏的性能表现就说不清了。我们支持的可见光越多，那么这个新方法就更好。



两个光源影响一个对象，索引3和1。

注意，由于我们不再循环遍历最大的可见光数量，就不再需要清除最后不使用的光源数据。

void ConfigureLights () {

//int i = 0;

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

…

}

//for (; i < maxVisibleLights; i++) {

// visibleLightColors[i] = Color.clear;

//}

}

5.2 更多的可见光源

我们的新方法使得以支持更多可见光源而不必自动地增加GPU必须要做的工作。让我们增加上限到16，和轻量级光源用的是一样的。这要求我们每帧发送更多的数据到GPU中，但是大多数兑现只会受少数的光源影响。调整shader中的MAX\_VISIBLE\_LIGHT。

#define MAX\_VISIBLE\_LIGHTS 16

以及MyPipeling中的maxVisibleLights。

const int maxVisibleLights = 16;

重新编译后，Unity会警告我们超出了之前的数组大小。不幸的是，不可以仅调整shader中的固定数组大小。那是图形API的限制，这不是我们想改就能改的。使用新的大小必须要重启，所以你要重启Unity编辑器。

在我们继续之前，添加更多的光源到场景中，我们需要意识到unity\_4LightIndices0仅包含最多四个索引，即使一个对象限制被超过四个的光源所影响。为了防止不正确的结果，我们需要确保我们的光源循环不会超过四次。

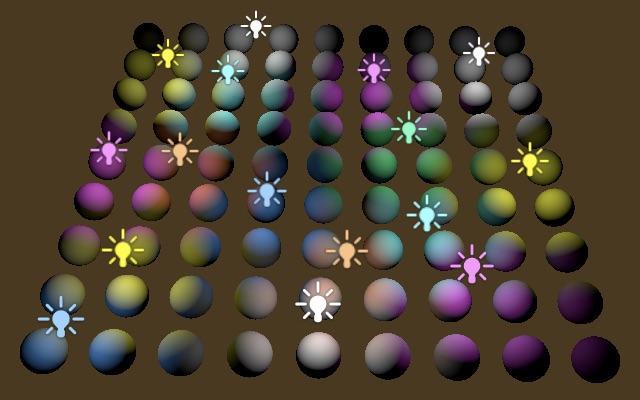
for (int i = 0; i < min(unity\_LightIndicesOffsetAndCount.y, 4); i++) {

int lightIndex = unity\_4LightIndices0[i];

diffuseLight +=

DiffuseLight(lightIndex, input.normal, input.worldPos);

}



十六个光源，每个对象最多四个。

但是我们不必限制自己每个对象最多四个光源。这还有unity\_4LightIndices1，它可以包含另外四个索引。让我们简单地在第一个循环后添加第二个，开始索引为4并从untiy\_4LightIndices1中检索光源索引。这样增加了每个对象最多地光源数量到八。不过，我们要确保不会超过八，因为在场景中对于对象来说可能会被更多的光源影响。

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

int lightIndex = unity\_4LightIndices0[i];

diffuseLight +=

DiffuseLight(lightIndex, input.normal, input.worldPos);

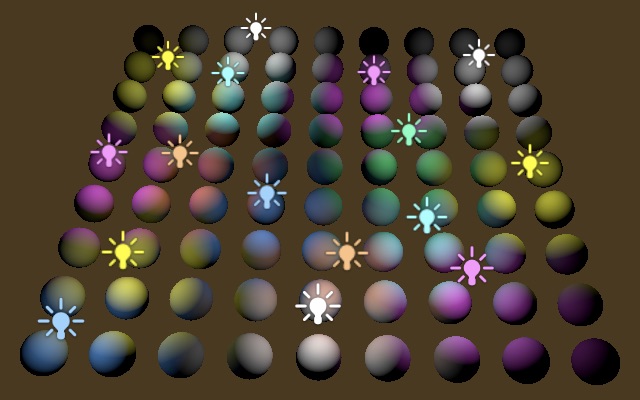
}

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

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

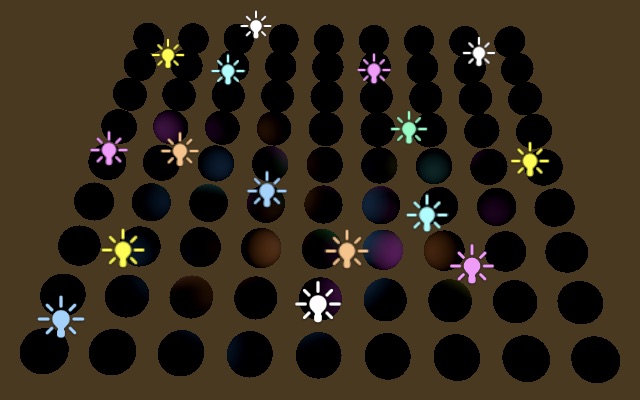
diffuseLight +=

DiffuseLight(lightIndex, input.normal, input.worldPos);

}

最多每对象八个光源。

因为光源索引基于相对重要性来存储，通常来说第二组四光源就不如第一个明显，大多数对象不会受太多光源的影响。为了看见额外的四光源的不同之处，你可以暂时性地关闭第一个循环。



每个对象跳过头四个光源。

5.3 逐顶点光源

因为第二组四光源的视觉重要性低于第一组，我们可以通过逐顶点而不是逐光源地计算它们的贡献度使它们有更小的开销。光源贡献度会在两顶点之间线性插值，它有更小的精度，但是对于微妙的漫反射光来说是可以接受的，只要光源距离比三角边长度大得多。

虽然可以微调我们支持的像素数量和顶点光源，但是我们只需要简单地移动第二个光源循环到LitPassVertex中，它只要求调整使用的变量。这意味着我们支持最多四个像素级光源加上四个顶点级光源。逐顶点光源必须要被添加到VertexOutput和被使用作为LitPassFragment中的diffuseLight初始值。

struct VertexOutput {

float4 clipPos : SV\_POSITION;

float3 normal : TEXCOORD0;

float3 worldPos : TEXCOORD1;

float3 vertexLighting : TEXCOORD2;

UNITY\_VERTEX\_INPUT\_INSTANCE\_ID

};

VertexOutput LitPassVertex (VertexInput input) {

VertexOutput output;

UNITY\_SETUP\_INSTANCE\_ID(input);

UNITY\_TRANSFER\_INSTANCE\_ID(input, output);

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

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

output.normal = mul((float3x3)UNITY\_MATRIX\_M, input.normal);

output.worldPos = worldPos.xyz;

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);

}

return output;

}

float4 LitPassFragment (VertexOutput input) : SV\_TARGET {

UNITY\_SETUP\_INSTANCE\_ID(input);

input.normal = normalize(input.normal);

float3 albedo = UNITY\_ACCESS\_INSTANCED\_PROP(PerInstance, \_Color).rgb;

float3 diffuseLight = input.vertexLighting;

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

…

}

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

// …

//}

float3 color = diffuseLight \* albedo;

return float4(color, 1);

}

5.4 太多的可见光源

尽管我们现在支持最多16个可见光源，因为场景中有足够多的光源，我们最终还可能超过这个限制。在这种情况下，当渲染时会忽略整体中最不重要的光源。然而，那仅仅是因为我们不复制他们的数据到shader中。Unity不知道这些，并且不会从每个对象中的光源索引列表中消除这些光源。因此我们最终可以使用超过界限的光照索引。为了防止这种情况，我们必须告诉Unity哪些光源已被消除。

我们可以获取所有可见光源的索引列表，通过在剔除结果中调用GetLightIndexMap。Unity允许我们去修改这个映射然后将它分配回剔除结果中，通过调用SetLightIndexMap。重点是Unity会跳过所有索引被改为-1的光源。在ConfigureLights的最后为所有超过最大值的光源做这些。

void ConfigureLights（）{

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

…

}

int [] lightIndices = cull.GetLightIndexMap（）;

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

lightIndices [i] = - 1 ;

}

cull.SetLightIndexMap（lightIndices）;

}

惟一我们真正需要去做这些的情况是，当我们最后有太多的可见光源，而这种情况不是每时每刻都会发生。

if (cull.visibleLights.Count > maxVisibleLights) {

int[] lightIndices = cull.GetLightIndexMap();

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

lightIndices[i] = -1;

}

cull.SetLightIndexMap(lightIndices);

}

不幸的是，GetLightIndexMap每次调用会创建一个新数组，因此我们的管线现在每帧都分配内存，最后我们会有太多的可见光源。我们现在不可以对此做任何的事情，但是未来的Unity发行版会给我们个免内存分配的GetLightIndexMap替代方案。

5.5 零可见光源

另一个可能的情况是零可见光源。这应该能行，但是不幸的是这种情况下Untiy尝试去设置光源索引会导致崩溃。我们可以避免通过当最少有一个可见光源的仅使用逐对象光源索引来避免崩溃。

var drawSettings = new DrawRendererSettings(

camera, new ShaderPassName("SRPDefaultUnlit")

) {

flags = drawFlags//,

//rendererConfiguration = RendererConfiguration.PerObjectLightIndices8

};

if (cull.visibleLights.Count > 0) {

drawSettings.rendererConfiguration =

RendererConfiguration.PerObjectLightIndices8;

}

如果这里没有光源，我们还可以完全跳过调用ConfigureLights。

if (cull.visibleLights.Count > 0) {

ConfigureLights();

}

Unity没有设置光源数据的一方面影响是它们保持了设置给前一个对象的值。因此我们会得出最终所有的对象的光源数量不为零。为了避免这种情况，我们会手动地设置untiy\_LightIndicesOffsetAndCount为零。

static int lightIndicesOffsetAndCountID =

Shader.PropertyToID("unity\_LightIndicesOffsetAndCount");

…

void Render (ScriptableRenderContext context, Camera camera) {

…

if (cull.visibleLights.Count > 0) {

ConfigureLights();

}

else {

cameraBuffer.SetGlobalVector(

lightIndicesOffsetAndCountID, Vector4.zero

);

}

…

}

下一篇教程是聚光灯阴影。