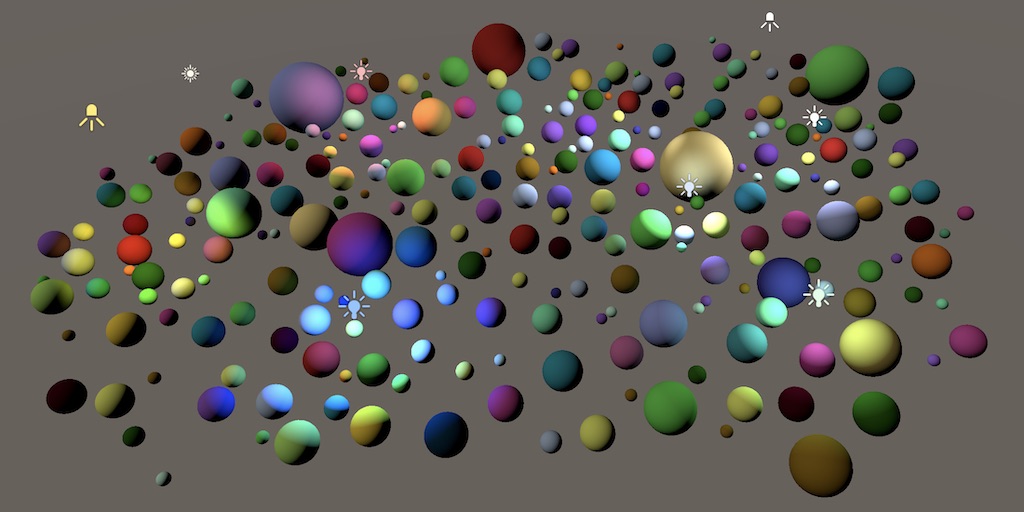
**光照-单pass前向渲染**

* **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

}

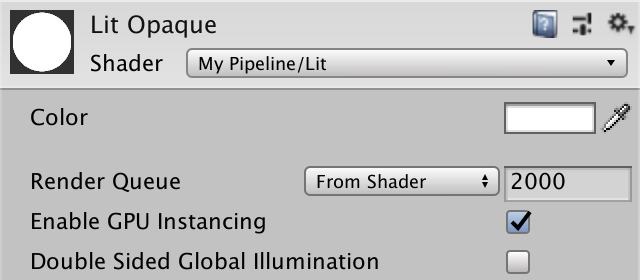
}

}

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);

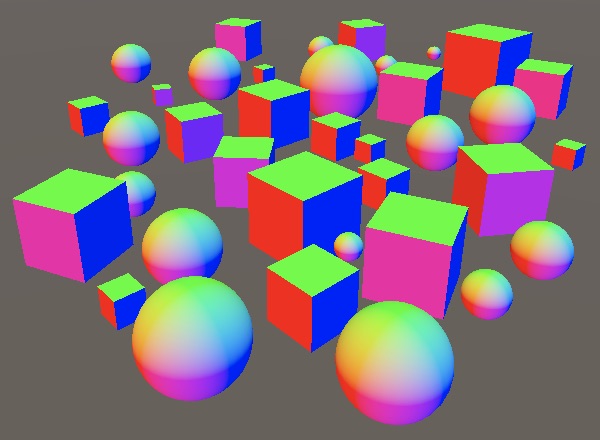
input.normal = normalize(input.normal);

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

float3 color = input.normal;

return float4(color, 1);

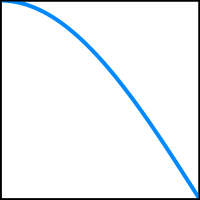
}



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);

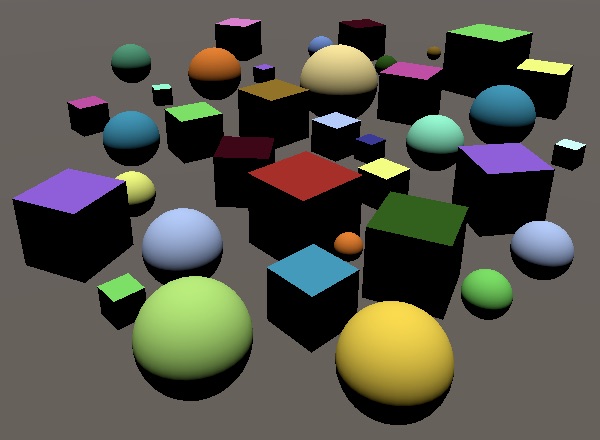
float3 albedo = UNITY\_ACCESS\_INSTANCED\_PROP(PerInstance, \_Color).rgb;

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。

* 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

当然，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);

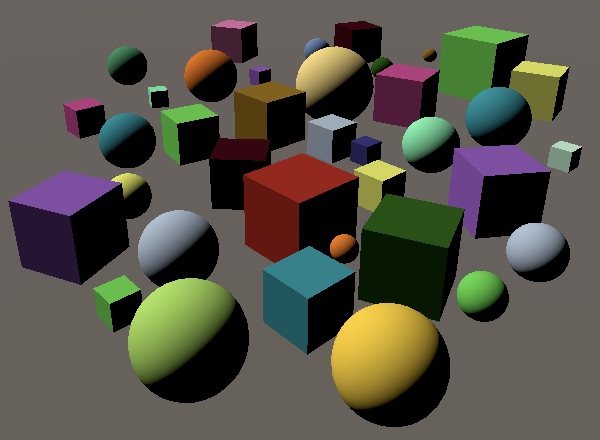
v.x = -v.x;

v.y = -v.y;

v.z = -v.z;

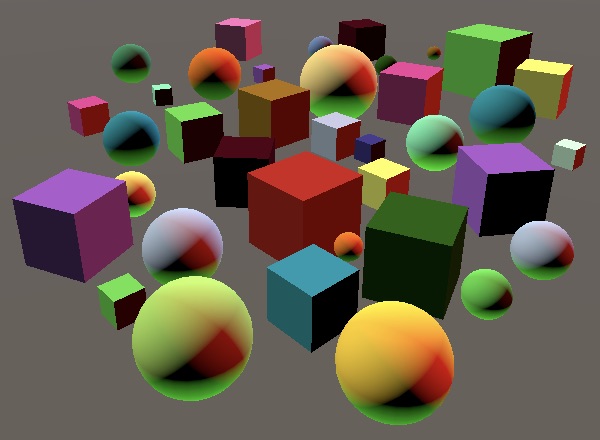
visibleLightDirections[i] = v;

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。



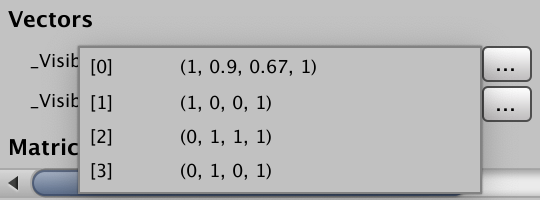
单个平行光的漫反射着色。

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;

}