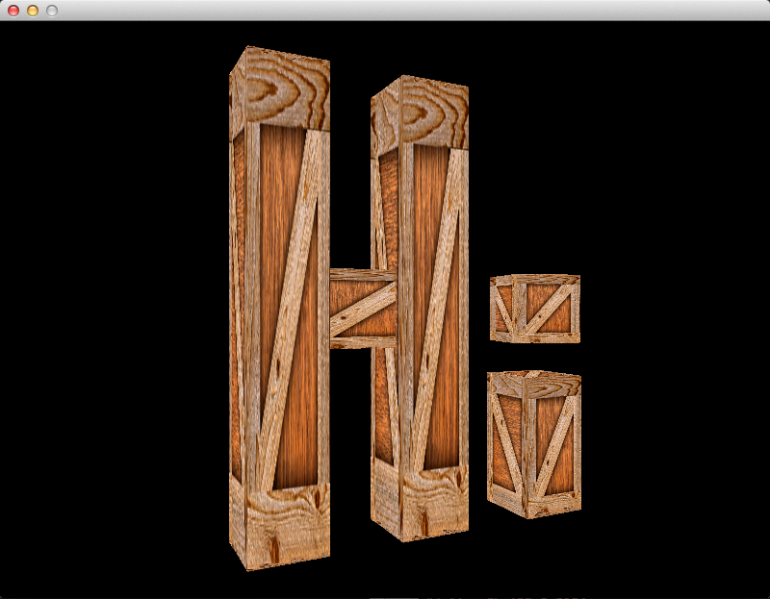
# [Modern OpenGL 05 – Model Assets & Instances](https://www.tomdalling.com/blog/modern-opengl/05-model-assets-and-instances/)



在本文中，我们将重构代码更像一个3D引擎/框架。具体地说，我们将用代表“资产（assets）”和“实例（instances）”的结构替换一些全局变量，最后，我们将拥有一个木箱资产，五个实例将在3D中拼出“HI”。

访问代码

从这里下载所有代码作为一个zip：

<https://github.com/tomdalling/opengl-series/archive/master.zip>

本系列文章中的所有代码都可以从github:

<https://github.com/tomdalling/opengl-series>获得。您可以从该页面下载所有文件的zip，或者您可以在熟悉Git的情况下克隆存储库。

本文基于前一篇文章的代码。

这篇文章的代码可以在[source/05\_asset\_instance](https://github.com/tomdalling/opengl-series/tree/master/source/05_asset_instance)文件夹中找到。在OS X上，打开根文件夹中的opengl-series.xcodeproj文件，并选择与本文对应的目标。在Windows上，在VisualStudio 2013中打开opengl-series.sln文件，并打开与本文对应的项目。

该项目包括所有的依赖关系，因此您不必安装或配置任何额外的东西。如果你在编译和运行代码方面任何问题，请联系我。

资产（assets）的一般定义

“资产”这个词很广，可以指各种各样的东西。其他3D引擎和框架可能使用“资源”一词而不是“资产”。“资产”一词通常包括音乐、声音效果、粒子发射器、着色器、网格、游戏级别等东西。为了这篇文章的目的，我们将定义一个可以被绘制的3D对象的资产。通常被称为“模型”。

为了这篇文章的目的，我们将定义一个可以被绘制的3D对象——通常被称为“模型”。

在更复杂的3D引擎中，模型通常由网格和材料组成。网格通常包含每个顶点数据：顶点、纹理坐标等。一个材质通常包含着色器，以及着色器的统一变量的一些值，例如：纹理、颜色、光泽等。在本文的代码中，我们将没有单独的M类。ESE和材料。为了简单起见，我们将有一个名为ModelAsset 的结构，它是材质和网格的组合。

ModelAsset 结构体定义

我们将用这个结构表示资产（assets）：

struct ModelAsset {

tdogl::Program\* shaders;

tdogl::Texture\* texture;

GLuint vbo;

GLuint vao;

GLenum drawType;

GLint drawStart;

GLint drawCount;

};

每个资产都包含着色器、单个纹理、VBO、VAO和所有glDrawArrays的参数。基本上，它包含了绘制整个3D物体所必需的一切。在以前的文章中，我们已经看到了所有这些变量，但是它们要么是全局的，要么是硬编码的常量。将这些元素组合成一个结构可以让我们在以后有多个资产，例如：木箱、茶壶和树。

您会注意到每个资产都有自己的着色器。这允许我们为不同的资产使用不同的着色器。单个着色器也可以由多个资产共享，因为它被存储为指针。

每个资产也有一个单一的纹理。大多数3D引擎都有多重纹理支持，但为了简单起见，我们只需要坚持每一个纹理。

VBO包含所有的顶点和纹理坐标，与前面的文章相同。

VAO也与以前的文章一样。

drawType、drawStart 和drawCount 变量将将glDrawArrays的参数保存。在前几篇文章中，这三个参数只是硬编码的常数，但它们变成了变量，以便使代码更可重用。

实例（instances）的一般定义

实例是具有独特的位置、大小和其他属性的资产。将资产与实例分离的主要原因是您可以拥有单个资产的多个实例。资产越少，你需要的视频内存就越少。

实例是具有单个位置、大小和其他属性的资产。将资产与实例分离的主要原因是您可以拥有单个资产的多个实例。

例如，可以通过创建单个树资产的100个实例来创建森林场景。每个实例将处于不同的位置，并且有略微不同的大小，再旋转一点点。从观众的角度来看，它看起来像100棵不同的树。从程序员的角度来看，实际上只有一棵树被绘制了100次。

实例vs实体

您可能会发现，“实例”听起来非常类似于您在其他框架和引擎中看到的“实体”。事实上，他们都有一个位置，一个大小，一个旋转，你可以画出来。您可以设计代码，以便实例和实体是同一事物，或者实体类继承实例类，反之亦然。然而，我们可以想出一个更好的设计。

在某些情况下，用实体而不是实例是更好的实际。例如，你的相机可能是一个实体，或者你有一个“隐形”实体是无形的，但是当玩家与它发生碰撞时会发生一些事情。

在其他情况下，您会希望拥有实例而不是实体。例如，也许游戏中有一个玩家无法到达的静态区域，所以你只需要画一组没有任何其他实体特征的实例，例如碰撞检测、动画、运动等等。

从上面的例子，我们可以得出结论，实体和实例是独立的事物，所以不应该从另一个继承。一个好的经验法则，不管你在编程什么，都倾向于构建而不是继承，所以让我们这样做。

一个体面的设计将是每个实体都有一个实例指针。如果实例指针为NULL，则该实体不能被绘制，并且是不可见的。如果您希望实例不是实体，那么也可以这样做，因为实例不以任何方式依赖实体。基本上，实例类仅用于绘图，实体类包含其余的功能。

如果你继续遵循这条设计的道路，最终你会拥有一个实体系统架构。在我看来，实体系统架构是相当优雅的，我可能会在将来写一篇关于它们的文章。

ModelInstance 结构

我们将用这个结构表示实例：

struct ModelInstance {

ModelAsset\* asset;

glm::mat4 transform;

};

这个结构非常简单。它只包含一个资产和一个转换矩阵。该矩阵是控制实例的大小、位置和旋转所必需的全部。每个实例矩阵通常被称为“模型矩阵”。

在本文的代码中，我们有一个资产和五个实例。这意味着一个资产将被绘制五次。每次资产绘制，它将有一个不同的变换矩阵应用，这将改变资产的位置、大小和旋转。

集成ModelAsset和ModelInstance

首先让我们看看全局变量的变化。

// new globals this article

ModelAsset gWoodenCrate;

std::list<ModelInstance> gInstances;

// deleted globals from last article

/\*

tdogl::Texture\* gTexture = NULL;

tdogl::Program\* gProgram = NULL;

GLuint gVAO = 0;

GLuint gVBO = 0;

\*/

// unchanged globals

GLFWwindow\* gWindow = NULL;

double gScrollY = 0.0;

tdogl::Camera gCamera;

GLfloat gDegreesRotated = 0.0f;

删除的全局变量现在都是ModelAsset 结构体的一部分。我们现在仅有一个资产，所以现在我们将把它保留在全局变量gWoodenCrate 中。最后，我们在变量gInstances中有一个实例列表。

现在让我们从LoadWoodenCrateAsset 函数的开始，看看资产是如何加载的：

static void LoadWoodenCrateAsset() {

gWoodenCrate.shaders = LoadShaders("vertex-shader.txt", "fragment-shader.txt");

gWoodenCrate.drawType = GL\_TRIANGLES;

gWoodenCrate.drawStart = 0;

gWoodenCrate.drawCount = 6\*2\*3;

gWoodenCrate.texture = LoadTexture("wooden-crate.jpg");

glGenBuffers(1, &gWoodenCrate.vbo);

glGenVertexArrays(1, &gWoodenCrate.vao);

//...

该代码与前一篇文章中的LoadCube 函数几乎相同，除了设置全局变量之外，它还在gWoodenCrate中设置了变量。此外，LoadShaders 和LoadTexture 函数现在返回一个值而不是直接设置全局变量。事实上，我们正在移除全局变量的使用表明代码的结构正在变得更好。

现在，我们已经看到了如何加载该资产，接下来让我们看看CreateInstances 函数内部的实例是如何生成的：

static void CreateInstances() {

ModelInstance dot;

dot.asset = &gWoodenCrate;

dot.transform = glm::mat4();

gInstances.push\_back(dot);

ModelInstance i;

i.asset = &gWoodenCrate;

i.transform = translate(0,-4,0) \* scale(1,2,1);

gInstances.push\_back(i);

ModelInstance hLeft;

hLeft.asset = &gWoodenCrate;

hLeft.transform = translate(-8,0,0) \* scale(1,6,1);

gInstances.push\_back(hLeft);

ModelInstance hRight;

hRight.asset = &gWoodenCrate;

hRight.transform = translate(-4,0,0) \* scale(1,6,1);

gInstances.push\_back(hRight);

ModelInstance hMid;

hMid.asset = &gWoodenCrate;

hMid.transform = translate(-6,0,0) \* scale(2,1,0.8);

gInstances.push\_back(hMid);

}

对于这五个实例中的任一个，我们将资产设置到&gWoodenCrate变量，然后将转换矩阵设置为单位阵的，然后将实例追加到gInstances 实例列表中。

函数translate 和scale 只是返回矩阵的便利函数。翻译控制实例的位置，比例控制大小。

单位矩阵是一个不做任何变换的特殊矩阵。

注意，第一个实例没有任何转换，该实例的矩阵是单位矩阵。单位矩阵是一个不做任何变换的特殊矩阵。我们将在前一篇文章中实现第一个实例的旋转，让我们看看Update 函数，看看它是如何完成的。

void Update(float secondsElapsed) {

//rotate the first instance in `gInstances`

const GLfloat degreesPerSecond = 180.0f;

gDegreesRotated += secondsElapsed \* degreesPerSecond;

while(gDegreesRotated > 360.0f) gDegreesRotated -= 360.0f;

gInstances.front().transform = glm::rotate(glm::mat4(), gDegreesRotated, glm::vec3(0,1,0));

//...

唯一的区别是添加了最后一条语句：

gInstances.front().transform = glm::rotate(glm::mat4(), gDegreesRotated, glm::vec3(0,1,0));

这使用gInstances的第一个实例，并基于最近更新的gDegreesRotated 全局变量来设置转换矩阵。

现在我们有一个需要绘制的实例列表，接下来看Render 函数：

static void Render() {

// clear everything

glClearColor(0, 0, 0, 1); // black

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// render all the instances

std::list<ModelInstance>::const\_iterator it;

for(it = gInstances.begin(); it != gInstances.end(); ++it){

RenderInstance(\*it);

}

// swap the display buffers (displays what was just drawn)

glfwSwapBuffers();

}

This function is simpler than in the last article. Now, it just loops over the gInstances list, and calls RenderInstance for every instance. Let's look in RenderInstance:

这个函数比上一篇文章要简单。现在，它只是循环gInstances 列表上，并调用每个实例的RenderInstance 。让我们来看看RenderInstance的实现：

static void RenderInstance(const ModelInstance& inst) {

ModelAsset\* asset = inst.asset;

tdogl::Program\* shaders = asset->shaders;

//bind the shaders

shaders->use();

//set the shader uniforms

shaders->setUniform("camera", gCamera.matrix());

shaders->setUniform("model", inst.transform);

shaders->setUniform("tex", 0); //set to 0 because the texture will be bound to GL\_TEXTURE0

//bind the texture

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, asset->texture->object());

//bind VAO and draw

glBindVertexArray(asset->vao);

glDrawArrays(asset->drawType, asset->drawStart, asset->drawCount);

//unbind everything

glBindVertexArray(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

shaders->stopUsing();

}

代码注释很好地解释了函数的步骤。此函数中的代码与前一篇文章中的Render 函数代码几乎相同，除了它使用ModelInstance 实例对象而不是全局对象。

差不多就是这样。我们已经将代码重组为资产和实例，这比使用全局变量更灵活。为了测试灵活性，尝试添加更多的实例。或者，如果你足够勇敢，尝试添加一个新的资产与不同的纹理，或不同的着色器。

优化

本文中的代码是天真的，未优化的。它仍然跑得足够快，所以这不是一个问题在这个阶段。然而，有一些相当简单的优化会使渲染更快。

当前的实现绑定和解开相同的着色器五次。如果着色器对于所有五个实例都是相同的，那么它只需要绑定一次。在下面的伪代码中，一个更优化的渲染循环看起来会像这样：

currentShader = NULL;

foreach(instance in gInstances) {

if(instance.asset.shader != currentShader){

if(currentShader) currentShader.stopUsing();

currentShader = instance.asset.shader;

currentShader.use();

}

RenderInstance(instance);

}

currentShader.stopUsing();

这样，如果着色器不改变，那么它不会被解除绑定。绑定和解除绑定着色器是相对耗时的操作，因此消除不必要的着色器绑定将加快循环。

要用多个不同的着色器来完成这项工作，您需要用着色器对gStasts列表进行排序。这样，所有具有相同着色器的实例将被分组在一起，因此每个着色器只会绑定一次。

下一个最耗时的操作是绑定每个实例的纹理。我们可以以与着色器绑定完全相同的方式优化纹理绑定。用着色器对实例进行排序，然后按纹理排序。在循环中，检查是否已经绑定了正确的纹理，如果是的话，不要解除绑定和重新绑定它。

场景图

在其他引擎/框架中，您可能会发现实例是分层存储的，也就是说，它们存储在树形结构中，而不是我们正在使用的std::list列表。实例可以称为“节点”。每个实例都有父和多个子。这基本上是场景图的定义。

既可以有实例列表，也可以有场景图。如果使用在本文的实例vs实体部分中解释的相同设计原理，则解决方案将是每个场景图节点具有指向实例的指针。在优化过程中，将实例列表保存到场景图中是有帮助的。

我们不需要场景图，所以现在让我们忽略它们。

OpenGL例示绘制功能

有一个OpenGL扩展是ARB\_draw\_instanced，它提供了本文所描述的功能。它包括将模型矩阵放入VBO，并调用glDrawArraysInstanced 来代替glDrawArrays。当渲染同一资产的数千个实例时，它可以用来获得更好的性能。使用它来绘制一个立方体的五个实例是多余的，但是如果你需要画一个虚构的实例，那么你应该考虑使用它。

这个功能在OpenGL ES 3中可用，这意味着它目前不存在于大多数移动设备上。在桌面上，它可以在OpenGL 2.1中作为一个扩展，并成为OpenGL 3.1中的核心配置文件的一部分。

未来文章内容预告

在下一篇文章中，我们将开始实施渲染，从单点光的漫反射开始。

追加资源

看看各种3D引擎的结构，如Unity、RiLiHT、OGR3D和LIGBDX，可能是有趣的。

一篇很好的文章，包括视频，关于游戏编程中的合成与继承

[OpenGL维基上的ARB\_draw\_instanced实例说明](http://www.opengl.org/wiki/Vertex_Rendering" \l "Instancing)

[链接到实体系统wiki上的实体系统的文章](http://entity-systems.wikidot.com/es-approaches)

# [Modern OpenGL 06 – Diffuse Point Lighting](https://www.tomdalling.com/blog/modern-opengl/06-diffuse-point-lighting/)

这是第一篇关于光源的文章。我们将开始我们的光源冒险通过实现任一像素，单点光源的漫射光源。在后面的文章中，我们将实现环境光源和镜面反射光源，定向光源，聚光灯，衰减，并使用多个光源。

访问代码

从这里下载所有代码作为一个zip：

<https://github.com/tomdalling/opengl-series/archive/master.zip>

本系列文章中的所有代码都可以从github:

<https://github.com/tomdalling/opengl-series>获得。您可以从该页面下载所有文件的zip，或者您可以在熟悉Git的情况下克隆存储库。

本文基于前一篇文章的代码。

这篇文章的代码可以在[source/06\_diffuse\_lighting](https://github.com/tomdalling/opengl-series/tree/master/source/06_diffuse_lighting) 文件夹中找到。在OS X上，打开根文件夹中的opengl-series.xcodeproj文件，并选择与本文对应的目标。在Windows上，在VisualStudio 2013中打开opengl-series.sln文件，并打开与本文对应的项目。

该项目包括所有的依赖关系，因此您不必安装或配置任何额外的东西。如果你在编译和运行代码方面任何问题，请联系我。

本文的键盘操作

|  |  |
| --- | --- |
| W | Move forward |
| A | Move left |
| S | Move backward |
| D | Move right |
| X | Move up |
| Z | Move down |

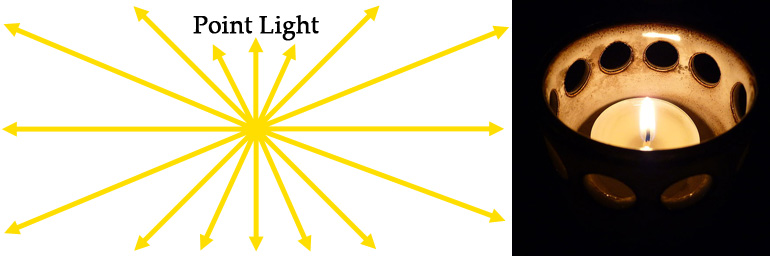
|  |  |
| --- | --- |
| 1 | Set light position to camera position |
| 2 | Set light intensities to green |
| 3 | Set light intensities to red |
| 4 | Set light intensities to white |

点光源

点光源从一个点向外辐射光，就像蜡烛一样。

在本文中，我们将实施一种称为点光源的光。点光源从一个点向外辐射光，就像蜡烛一样。如果你看这篇文章的截图，它看起来就像是一根无形的蜡烛被固定在木箱上。

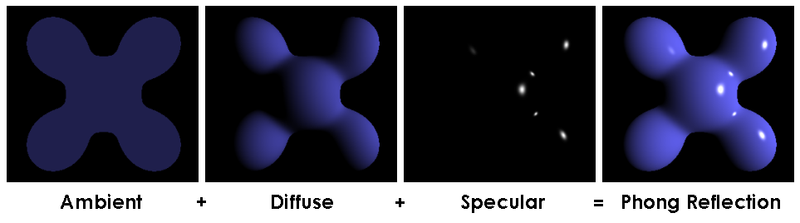
还有其他常见类型的灯，如定向光源和聚光灯，但我们将在后面的文章中介绍。



庞氏反射模型

庞氏反射模型提供了一种基于光源和表面的参数计算像素颜色的方法。

我们将实施庞氏反射模型的漫反射分量。庞氏反射模型提供了一种基于光源和表面的参数计算像素颜色的方法。光源参数包括光的位置/方向和光的颜色/强度。表面参数包括表面的颜色、表面所面对的方向（通常是法向）和表面的“光泽”。



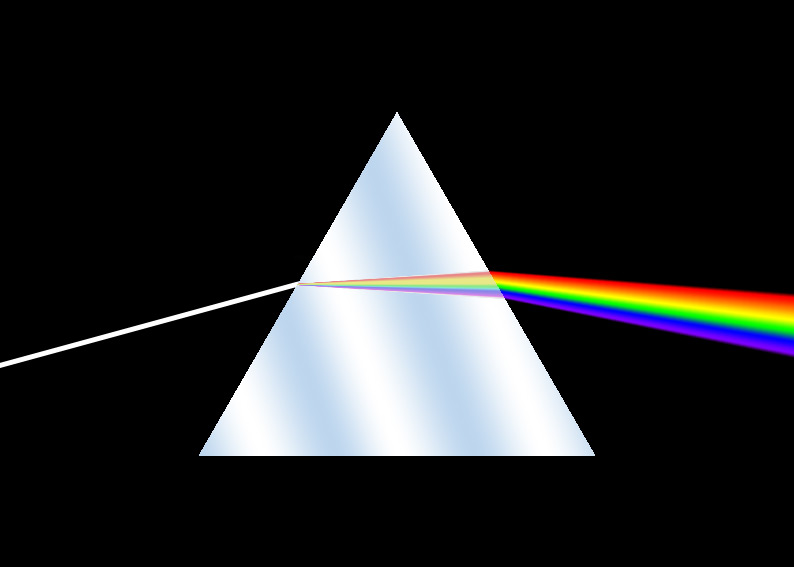
庞氏反射模型有三个组成部分：环境、漫反射和镜面反射。漫反射分量是最重要的一个，正如上图所示。环境组件被用来阻止物体的未照亮的背面是纯黑色的，因为在大多数3D场景中纯黑色看起来是人工的。镜面组件是使物体看起来有光泽或阴暗的原因。因为我们只在本文中实现了扩散组件，木箱板在背面是纯黑色的，并且不能发光。我们将在接下来的几篇文章中实现环境和镜面组件。

光源强度

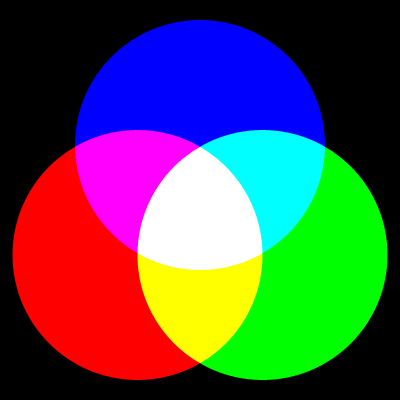
白光包含所有的颜色。

庞氏反射模型是松散地基于光在现实世界中的行为方式。因此，为了理解OpenGL中的光照，我们必须了解光的物理——虽然不是很多，但足以让我们的3D场景看起来更逼真。

白光包含了我们人眼所能看到的所有颜色。这可以通过将白光照射到棱镜中来证明，这使得光折射成彩虹。



另一种证明这一点的方法是获得三种不同颜色的光——红、绿、蓝，并把它们放在暗室里的白色表面上。如果你这样做，你会看到下面的图像。



我们可以从中得出一些结论：

白色=红色+绿色+蓝色

黄色=红色+绿色

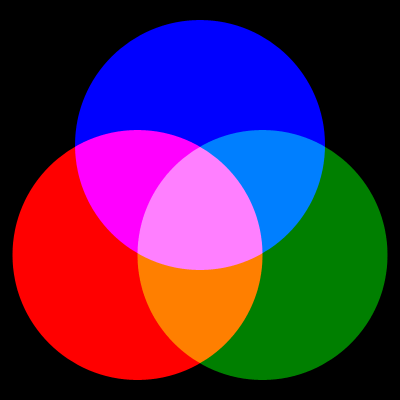
Cyan（浅蓝色）=蓝色+绿色

品红（紫色）=红色+蓝色

黑色=没有颜色

使用三种颜色的光，我们可以制作八种不同的颜色：红、绿、蓝、黄、青、洋红、黑和白。

但是其他颜色，比如橙色呢？好吧，如果你把绿灯弄得像以前一样亮，你会看到下面的图像。



颜色是不同强度的红色、绿色和蓝色光的组合。

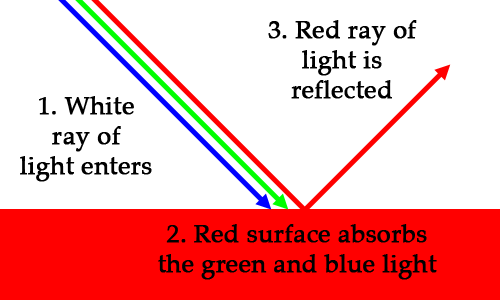
光的颜色称为光的强度。

降低绿色的强度（亦即亮度）已经产生了一些新的颜色：深绿色，天蓝色，橙色和粉红色。

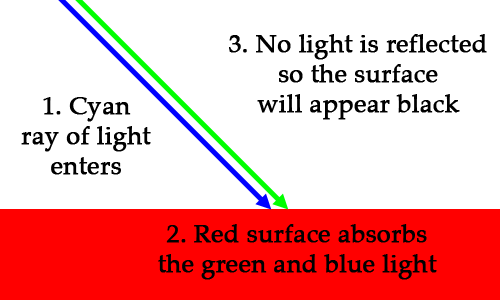
正如你所看到的，颜色是不同强度的红色，绿色和蓝色光的组合。这就是为什么光的颜色被称为光的强度。当我们在代码中设置光的颜色时，我们将使用一个vec3 来保存红色、绿色和蓝色的强度。

颜色的吸收与反射

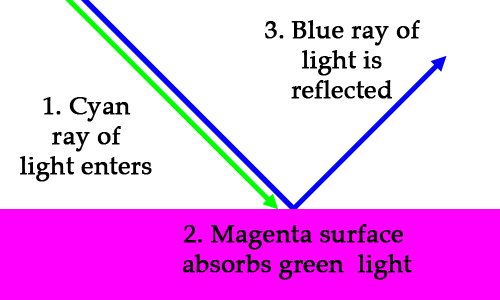
假设你在看一辆红色的车。太阳发出一束白光。光线从车上反弹，进入你的眼睛。你的眼睛检测到光线只有红光，这就是为什么你看到一辆红色的车而不是白色的车。我们知道白光包含所有的颜色，那么绿色和蓝色发生了什么？绿光和蓝光被表面**吸收**，红光被**反射**。



如果我们在红色汽车上闪耀一个纯青色（蓝色+绿色）光呢？如果这辆车是纯红色的，它看起来会是黑色的，因为它能吸收100%的光线。



若青色（蓝色+绿色）光投射在品红色（红色+蓝色）面上会如何？



因此，如果你把一个浅蓝色的手电筒照到一个粉红色的表面上，表面看起来会是深蓝色的。这很奇怪，但却是真的。

表面的RGB颜色表示光是如何被表面吸收和反射的。

如果你查看每个颜色的RGB值，你会注意到这些值代表反射率。（0，0，0）是黑色，这意味着不反射光。（1，1，1）是白色，这意味着反射所有的光。（1，0，0）是红色，这意味着只反射红色。青色是（0，1，1），这意味着只反映蓝色和绿色。表面的RGB颜色表示光是如何被表面吸收和反射的。

计算反射颜色很简单。基本公式是：强度×表面颜色=反射强度。例如：

cyan light × magenta surface = blue light

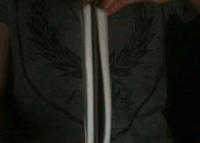
(0, 1, 1) × (1, 0, 1) = (0, 0, 1)

乘法是通过单独地将每个RGB分量相乘来实现的，例如：

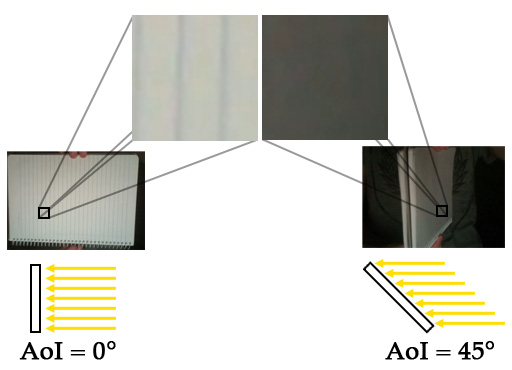
(X,Y,Z)(A,B,C)=(XA,YB,ZC)

入射角

这是一个生动的动画，是我在灯光前旋转记事本：



该动画演示了光入射角（AoI）如何影响表面（记事本）的颜色。注意记事本在面对光前线时是如何亮的。当记事本从最亮的位置旋转时，表面变得越来越暗。

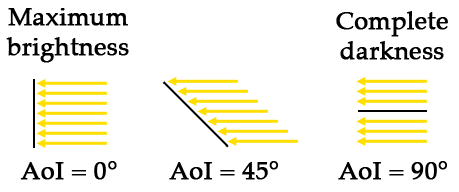


光线入射到表面的角度称为入射角（AoI）。

入射角影响表面的亮度。

光线入射到表面的角度称为入射角（AoI）。入射角影响表面的亮度。AoI是漫反射的基础，我们将在本文中实现。

让我们来看看AoI的两个极端：最大亮度和完全黑暗。



当表面垂直于光线（AOI＝0°）时，出现最大亮度。当表面平行于光线（AoI＝90°）时，会发生完全黑暗。如果AoI大于90°，则射线击中表面的背面。如果光线击中背部，那么它肯定不会击中前方，所以像素也应该完全黑暗。

如果我们把亮度表示为一个单一的数字，其中0是完全暗的，1是最大亮度，那么很容易根据AoI的余弦来计算。公式是亮度＝cos（AoI）。让我们看看一些角度的余弦，只是为了证明它是有效的：

cos( 0°) = 1.00 (100% of maximum brightness)

cos( 5°) = 0.98 ( 98% of maximum brightness)

cos( 45°) = 0.71 ( 71% of maximum brightness)

cos( 85°) = 0.09 ( 9% of maximum brightness)

cos( 90°) = 0.00 (Completely dark)

cos(100°) = -0.17 (Completely dark. Negative value means light is hitting the back side)

一旦我们有一个亮度值在0和1之间，我们可以通过反射光的强度乘以它来获得像素的最终颜色。这里有一个青色光的例子：

亮度×光强度=像素的最终颜色

1.0 × (0, 1, 1) = (0, 1, 1) (cyan, unchanged)

0.5 × (0, 1, 1) = (0, 0.5, 0.5) (绿松石色, 黑暗的青色)

0.0 × (0, 1, 1) = (0, 0, 0) (black)

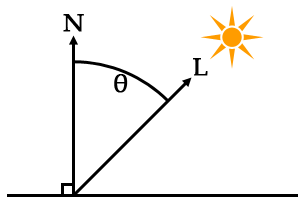
这个“亮度”值在0到1之间有时被称为“扩散系数”。

表面法线

法线是垂直于（直角，90°）到表面的单位矢量。

为了计算AoI，我们首先需要知道每个表面所面对的方向。表面所面对的方向称为该表面的法线。法线是垂直于（直角，90°）表面的单位矢量。

入射角被定义为表面法线与光源方向之间的夹角。



N=表面法线向量

L＝从表面到光源的矢量

θ=入射角

从表面到光源的矢量L可以用矢量减法计算，如下所示：

L=光位置-表面位置

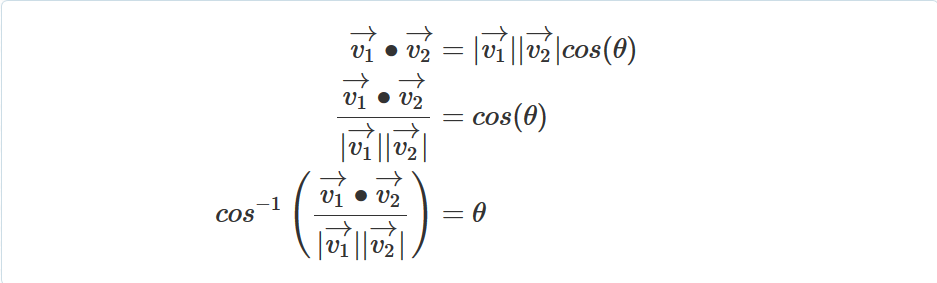
如果你需要复习矢量数学，你可能希望阅读本系列的第04篇文章。

表面法线通常由VBO提供，同样提供顶点和纹理坐标。当我们进入本文的代码时，我们将为VBO中的每个顶点添加一个法向量。

计算两个向量之间的夹角：点积

可以利用向量的点积计算两个向量之间的夹角。点积是一个带两个向量的运算，结果是数值（标量）。令人震惊的是，点积看起来像是数学记号中的一个点：。在GLSL和GLM中，它是一个称为“dot”的函数：dot（V1，V2）和glm::dot（V1，V2）。

点积的结果与两个向量之间的夹角有关。确切的关系是：



在V1和V2为向量的情况下，θ是两个向量之间的夹角，|v⃗|是向量v的模量。

与上面柿子完全相同的代码，如下：

dot(v1,v2) == length(v1)\*length(v2)\*cos(angle)

dot(v1,v2)/(length(v1)\*length(v2)) == cos(angle)

acos(dot(v1,v2)/(length(v1)\*length(v2))) == angle

length是一个返回向量模量的GLSL函数。

我们实际上不需要知道AoI，我们只需要cos（AoI），它代表亮度。片段着色器中的亮度计算是基于上面公式的中间行：

// cos(angle) = dot(v1, v2)/ (length(v1) \* length(v2))

float brightness = dot(normal, surfaceToLight) / (length(surfaceToLight) \* length(normal));

法线的矩阵变换

法线通常是在模型空间中提供的，这意味着它们在应用任何变换之前都与资产的顶点相对应。然而，当我们计算从表面到光的矢量时，这是在世界空间中完成的。世界空间是所有3D物体被定位/缩放/旋转到3D场景中的位置的地方。例如，在模型空间中，我们木箱资产的中心是（0，0，0）。在我们定位和调整板条箱来拼出3D场景中的“Hi”之后，那些板条箱在世界空间里。从模型空间到世界空间的转换是由场景中每个实例的“模型矩阵”来完成的，即本文中代码的ModelInstance::transform。

当我们变换一个资产的顶点时，我们也必须变换法线。

当顶点从模型空间转换到世界空间时，它们可能已经被旋转。如果一个表面的顶点已经被旋转，那么表面现在面对不同的方向，所以表面的法线将是不同的。这意味着当我们变换一个资产的顶点时，我们也必须变换法线。

到目前为止，我们只使用矩阵来变换坐标。问题是法线不是坐标，它们是表示方向的单位向量。旋转变换是很好的，因为旋转单位矢量得到另一单位矢量，但是缩放或平移法向将导致不正确的法向。解决方案是将法线乘以不同的矩阵——一个具有平移和缩放部分固定的矩阵。

缩放或平移法向量会导致错误的法向。

删除4x4矩阵的平移部分很简单：我们只需删除第四列和行，将其转换为3x3矩阵。固定缩放有点棘手，但我会直接跳到答案，这是逆矩阵和转置矩阵。我们还需要重新调整每个法向量后，当它已被转换，以确保它仍然是单位向量。GLSL是这样做的：

mat3 normalMatrix = transpose(inverse(mat3(model)));vec3 transformedNormal = normalize(normalMatrix \* normal);

model 变量是原来的4x4模型变换矩阵。mat3 函数移除矩阵的平移部分。inverse 函数和transpose 函数将修正缩放部分。最后，在用normalMatrix \* normal变换原始法线之后，normalize 函数将确保变换后的法线是单位向量。

优化注意：重新计算每一个片段/像素的正常变换矩阵对性能不是很好。为了获得更好的性能，在C++中计算矩阵，使它在着色器中设为uniform变量。

顶点着色器

唷！前面有大量的阅读，但现在我们进入了代码阶段。让我们先来看一下顶点着色器。

#version 150

uniform mat4 camera;

uniform mat4 model;

in vec3 vert;

in vec2 vertTexCoord;

in vec3 vertNormal;

out vec3 fragVert;

out vec2 fragTexCoord;

out vec3 fragNormal;

void main() {

// Pass some variables to the fragment shader

fragTexCoord = vertTexCoord;

fragNormal = vertNormal;

fragVert = vert;

// Apply all matrix transformations to vert

gl\_Position = camera \* model \* vec4(vert, 1);

}

这与上一篇文章大致相同。主要的区别是，我们有一个表面法线的新输入变量称为vertNormal。此外，我们将三个变量直接发送到片段着色器而不修改它们：顶点、法线和纹理坐标。

片段着色器

现在让我们看看片段着色器，在那里所有的渲染计算都完成了。

#version 150

uniform mat4 model;

uniform sampler2D tex;

uniform struct Light {

vec3 position;

vec3 intensities; //a.k.a the color of the light

} light;

in vec2 fragTexCoord;

in vec3 fragNormal;

in vec3 fragVert;

out vec4 finalColor;

void main() {

//calculate normal in world coordinates

mat3 normalMatrix = transpose(inverse(mat3(model)));

vec3 normal = normalize(normalMatrix \* fragNormal);

//calculate the location of this fragment (pixel) in world coordinates

vec3 fragPosition = vec3(model \* vec4(fragVert, 1));

//calculate the vector from this pixels surface to the light source

vec3 surfaceToLight = light.position - fragPosition;

//calculate the cosine of the angle of incidence

float brightness = dot(normal, surfaceToLight) / (length(surfaceToLight) \* length(normal));

brightness = clamp(brightness, 0, 1);

//calculate final color of the pixel, based on:

// 1. The angle of incidence: brightness

// 2. The color/intensities of the light: light.intensities

// 3. The texture and texture coord: texture(tex, fragTexCoord)

vec4 surfaceColor = texture(tex, fragTexCoord);

finalColor = vec4(brightness \* light.intensities \* surfaceColor.rgb, surfaceColor.a);

}

我们有一个新的着色器称为light，它是一个包含光的位置和强度/颜色的结构。

变量fragTexCoord、fragNormal 和fragVert 都直接来自顶点着色器。fragTexCoord是纹理坐标，正如我们在以前的文章中看到的。fragNormal 是这个片段/像素的未转化表面法线。fragVert 是我们绘制的表面顶点的未转化坐标。

主体的第一部分将常态转换为世界空间，如本文的前一节所解释的。

main 函数的第一部分将常态转换为世界空间，如本文的前一节所解释的。

mat3 normalMatrix = transpose(inverse(mat3(model)));vec3 normal = normalize(normalMatrix \* fragNormal);

下一部分将表面坐标fragVert 转化为世界空间。

vec3 fragPosition = vec3(model \* vec4(fragVert, 1));

Uniform变量model 是4x4矩阵，所以我们将坐标转换成vec4，以便进行变换，然后将其转换为vec3。

接下来，我们计算从表面坐标到光坐标的矢量，它们都在世界空间中。

vec3 surfaceToLight = light.position - fragPosition;

接下来，我们计算亮度，等于cos（angleOfIncidence）。正如本文前面所解释的，我们使用法向的点积和指向光的矢量。

float brightness = dot(normal, surfaceToLight) / (length(surfaceToLight) \* length(normal));

记住亮度可以是负的，这意味着光正在撞击表面的背面——我们将亮度限制在0到1之间。负值将变成0，这意味着完全黑暗。

brightness = clamp(brightness, 0, 1);

最后，使用亮度、光强和表面颜色，我们可以计算片段/像素的颜色。

vec4 surfaceColor = texture(tex, fragTexCoord);

finalColor = vec4(brightness \* light.intensities \* surfaceColor.rgb, surfaceColor.a);

texture(tex, fragTexCoord)将从纹理获得表面的颜色。这种表面颜色决定了光的强度是如何反射或吸收的。记住公式 强度×表面颜色=反射强度。我们在这里使用代码light.intensities \* surfaceColor.rgb来实现。

在用light.intensities \* surfaceColor.rgb计算反射强度之后，我们将它们乘以brightness 以获得最终颜色。乘以亮度会使入射角的反射强度变暗。

光强度是vec3 (RGB),，但最终颜色是vec4 （RGBA），所以我们将强度转换为vec4 ，并将alpha通道设置为来自纹理的Alpha值：surfaceColor.a。这意味着在纹理透明的地方，渲染的表面将是透明的。

Main.CPP的变化

大多数的渲染设置是在片段着色器，所以C++代码的变化是相当小的。

我们有一个新的全局称为gLight ，它是片段着色器中的光结构的一致副本。

struct Light {

glm::vec3 position;

glm::vec3 intensities; //a.k.a. the color of the light

};

// ...

Light gLight;

我们更新VBO以包括每个顶点的法线。旧的缓冲区数据是这样的：

// X Y Z U V

//bottom

-1.0f,-1.0f,-1.0f, 0.0f, 0.0f,

1.0f,-1.0f,-1.0f, 1.0f, 0.0f,

-1.0f,-1.0f, 1.0f, 0.0f, 1.0f,

1.0f,-1.0f,-1.0f, 1.0f, 0.0f,

1.0f,-1.0f, 1.0f, 1.0f, 1.0f,

-1.0f,-1.0f, 1.0f, 0.0f, 1.0f,

//...

现在看起来改成这样：

// X Y Z U V Normal

//bottom

-1.0f,-1.0f,-1.0f, 0.0f, 0.0f, 0.0f, -1.0f, 0.0f,

1.0f,-1.0f,-1.0f, 1.0f, 0.0f, 0.0f, -1.0f, 0.0f,

-1.0f,-1.0f, 1.0f, 0.0f, 1.0f, 0.0f, -1.0f, 0.0f,

1.0f,-1.0f,-1.0f, 1.0f, 0.0f, 0.0f, -1.0f, 0.0f,

1.0f,-1.0f, 1.0f, 1.0f, 1.0f, 0.0f, -1.0f, 0.0f,

-1.0f,-1.0f, 1.0f, 0.0f, 1.0f, 0.0f, -1.0f, 0.0f,

//...

上面显示的顶点构成木箱的底面。注意它们是如何具有相同的法向，（0，-1, 0），它是指向y轴的单位向量。木箱板上有六个面，每个面都有一个法线，它直接指向X、Y或Z轴的正方向或负方向。

因为VBO的格式已经改变为包括法线，所以我们也必须改变VAO。这是旧的VAO设置代码：

// connect the xyz to the "vert" attribute of the vertex shader

glEnableVertexAttribArray(gWoodenCrate.shaders->attrib("vert"));

glVertexAttribPointer(gWoodenCrate.shaders->attrib("vert"), 3, GL\_FLOAT, GL\_FALSE, 5\*sizeof(GLfloat), NULL);

// connect the uv coords to the "vertTexCoord" attribute of the vertex shader

glEnableVertexAttribArray(gWoodenCrate.shaders->attrib("vertTexCoord"));

glVertexAttribPointer(gWoodenCrate.shaders->attrib("vertTexCoord"), 2, GL\_FLOAT, GL\_TRUE, 5\*sizeof(GLfloat), (const GLvoid\*)(3 \* sizeof(GLfloat)));

这是新的VAO设置代码：

// connect the xyz to the "vert" attribute of the vertex shader

glEnableVertexAttribArray(gWoodenCrate.shaders->attrib("vert"));

glVertexAttribPointer(gWoodenCrate.shaders->attrib("vert"), 3, GL\_FLOAT, GL\_FALSE, 8\*sizeof(GLfloat), NULL);

// connect the uv coords to the "vertTexCoord" attribute of the vertex shader

glEnableVertexAttribArray(gWoodenCrate.shaders->attrib("vertTexCoord"));

glVertexAttribPointer(gWoodenCrate.shaders->attrib("vertTexCoord"), 2, GL\_FLOAT, GL\_TRUE, 8\*sizeof(GLfloat), (const GLvoid\*)(3 \* sizeof(GLfloat)));

// connect the normal to the "vertNormal" attribute of the vertex shader

glEnableVertexAttribArray(gWoodenCrate.shaders->attrib("vertNormal"));

glVertexAttribPointer(gWoodenCrate.shaders->attrib("vertNormal"), 3, GL\_FLOAT, GL\_TRUE, 8\*sizeof(GLfloat), (const GLvoid\*)(5 \* sizeof(GLfloat)));

基本上是一样的，只是加上法线。

现在让我们看看渲染在RenderInstance 函数中的变化。在渲染每个实例之前，我们所要做的就是在着色器中设置光的位置和颜色。我们把位置和强度直接从全局变量的gLight 中移开。

shaders->setUniform("light.position", gLight.position);

shaders->setUniform("light.intensities", gLight.intensities);

接下来，让我们看看新的键盘操作来改变Update 函数内部的光源设置。

//move light

if(glfwGetKey(gWindow, '1'))

gLight.position = gCamera.position();

// change light color

if(glfwGetKey(gWindow, '2'))

gLight.intensities = glm::vec3(1,0,0); //redelse if(glfwGetKey(gWindow, '3'))

gLight.intensities = glm::vec3(0,1,0); //greenelse if(glfwGetKey(gWindow, '4'))

gLight.intensities = glm::vec3(1,1,1); //white

按下1键将光源的位置设置到照相机的当前位置。移动光将允许我们观察入射角如何影响表面的亮度。

按下2, 3和4键将改变光的颜色。

在AppMin函数中，当程序启动时，我们设置光的初始位置和颜色。

gLight.position = gCamera.position();

gLight.intensities = glm::vec3(1,1,1); //white

最后，有一些与渲染无关的变化。我提高了相机的移动速度，因为它看起来有点慢。我还把相机的近平面距离从0.01增加到0.5，因为0.01太小，这会导致木板箱上出现奇怪的锯齿状边缘。

gCamera.setNearAndFarPlanes(0.5f, 100.0f);

就是这样！我们现在有一个单一的、漫射的点光源。这是3D渲染的第一步，我们将在接下来的几篇文章中构建它。

逐顶点与逐片段渲染

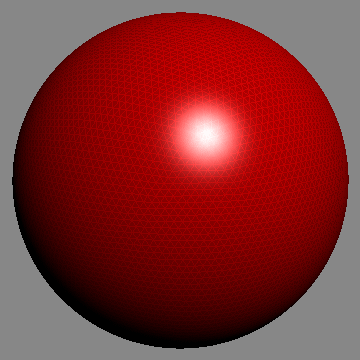
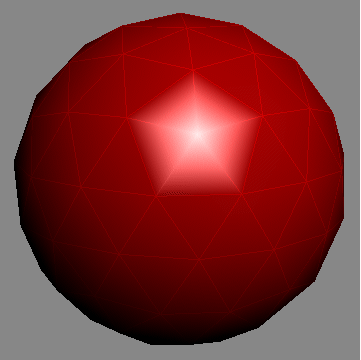
在这篇文章中，我们已经实现了逐片段渲染，也称为旁氏着色渲染法，请不要与旁氏反射模型混淆。我们可以选择，实现逐顶点渲染，也称为高洛德着色渲染法。这两种实现之间的差异在于执行渲染计算：在片段着色器中，或者在顶点着色器中。旧的OpenGL固定函数流水线实现逐顶点渲染，如果你想要逐片段渲染你必须编写自己的着色器。逐片段渲染看起来优于逐顶点渲染，这是一个从旧的固定功能管道切换到着色器的好理由。

如果我们选择逐顶点渲染，亮度将在每个顶点-即，在每一个木箱的拐角处计算。然后，亮度将在每个三角形中的所有像素内进行插值。例如，如果一个拐角是黑暗的，而另一个是光明的，两个拐角之间的像素会从黑暗变亮。现在让我们来看看本文截图中的一个板条箱：



注意所有的拐角都是暗的，因为明亮的区域正好在表面的中间。如果我们实现了逐顶点渲染，这看起来很恐怖。所有的拐角都是黑暗的，所以整个表面也会是黑暗的。它看起来是不对的，好像表面前面没有光。

这是逐顶点渲染的问题-它经常看起来错误，特别是如果模型的顶点离得很远。一种解决方案是增加模型中顶点的数量。下面的两幅图像显示每个顶点渲染应用到两个球：一个有几个顶点（低聚），一个有很多顶点（高多边形）。



低聚球有一个奇怪的五角大厦形状的亮点，但高聚球有一个圆形的亮点，看起来更正确。

使用逐片段渲染，我们可回避这个问题。我们不是计算木箱的每个角落的亮度，而是计算每个像素的亮度。为什么逐片段渲染看起来比逐顶点渲染好：它对每个表面做更多的照明计算，所以它更精确。

然而，还有一个折衷方案。逐顶点渲染需要三个三角形计算。逐片段渲染需要逐个计算每个片段的三角形覆盖，这通常是远远超过三。此外，屏幕分辨率的增加加重了每个三角形的碎片数量。更多的计算意味着渲染需要更长的时间，所以逐顶点渲染通常比逐片段渲染快。虽然新的硬件可以很容易地处理逐片段渲染，但渲染速度过低仍然是老的硬件和移动设备的问题。

未来文章一窥

在接下来的几篇文章中，我们将实施更多的渲染方法。我们将实施旁氏反射模型的环境和镜面组件。我们还将研究其他类型的光：定向光源和聚光灯。

追加资源

[教程8：基于OpenGL教程的基本着色](http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/)

[第4章：用Joe Groff的现代OpenGL演示文稿绘制动态三维场景。](http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-4:-Rendering-a-Dynamic-3D-Scene-with-Phong-Shading.html)

从学习现代3D图形编程书看：

[灯亮](http://www.arcsynthesis.org/gltut/Illumination/Tutorial%2009.html)

[正规变换](http://www.arcsynthesis.org/gltut/Illumination/Tut09%20Normal%20Transformation.html)

[教程18——OGLDEV现代OpenGL教程的漫射照明](http://ogldev.atspace.co.uk/www/tutorial18/tutorial18.html)

来自OpenGL编程的电子书：

[漫反射](http://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Diffuse_Reflection)

[关于每个顶点与每个片段照明的一些讨论。](http://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Smooth_Specular_Highlights)

相关维基百科文章：

入射角

庞氏反射模型

漫反射

旁氏渲染法

高洛德着色渲染法

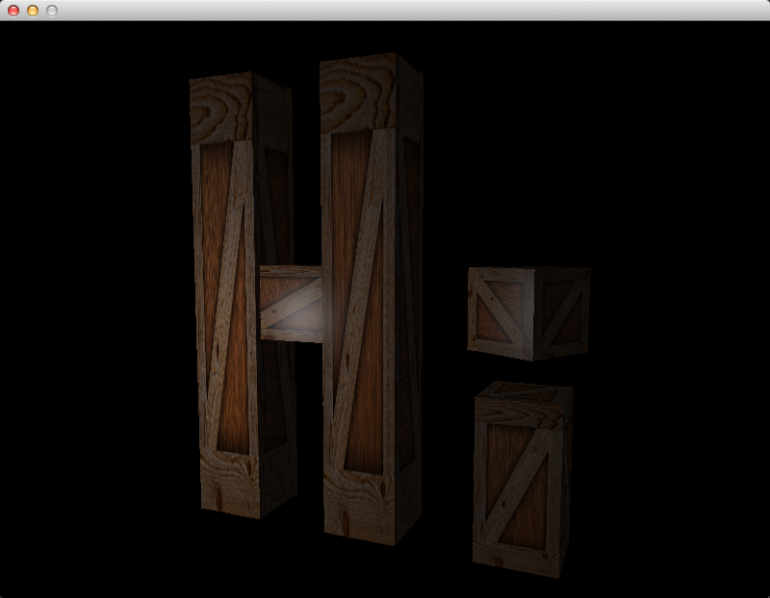
白光

加色

法向（几何）

向量点积

# [Modern OpenGL 07 – More Lighting: Ambient, Specular, Attenuation, Gamma](https://www.tomdalling.com/blog/modern-opengl/07-more-lighting-ambient-specular-attenuation-gamma/)



在这篇文章中，我们将继续实现光源。我们将实现旁氏反射模型的环境和镜面分量、衰减和伽马校正。

访问代码

从这里下载所有代码作为一个zip：

<https://github.com/tomdalling/opengl-series/archive/master.zip>

本系列文章中的所有代码都可以从github:

<https://github.com/tomdalling/opengl-series>获得。您可以从该页面下载所有文件的zip，或者您可以在熟悉Git的情况下克隆存储库。

本文基于前一篇文章的代码。

这篇文章的代码可以在[source/07\_more\_lighting](https://github.com/tomdalling/opengl-series/tree/master/source/07_more_lighting) 文件夹中找到。在OS X上，打开根文件夹中的opengl-series.xcodeproj文件，并选择与本文对应的目标。在Windows上，在VisualStudio 2013中打开opengl-series.sln文件，并打开与本文对应的项目。

该项目包括所有的依赖关系，因此您不必安装或配置任何额外的东西。如果你在编译和运行代码方面任何问题，请联系我。

漫射分量diffuse

我们覆盖了前一篇文章中的漫反射组件，但是我已经重构了本文中的代码。新的GLSL看起来像这样：

vec3 normal = normalize(transpose(inverse(mat3(model))) \* fragNormal);

vec3 surfacePos = vec3(model \* vec4(fragVert, 1));

vec4 surfaceColor = texture(materialTex, fragTexCoord);

vec3 surfaceToLight = normalize(light.position - surfacePos);

float diffuseCoefficient = max(0.0, dot(normal, surfaceToLight));

vec3 diffuse = diffuseCoefficient \* surfaceColor.rgb \* light.intensities;

diffuseCoefficient （扩散效率）以前被称为brightness（亮度）。它使用max 函数而不是clamp 来避免负值。

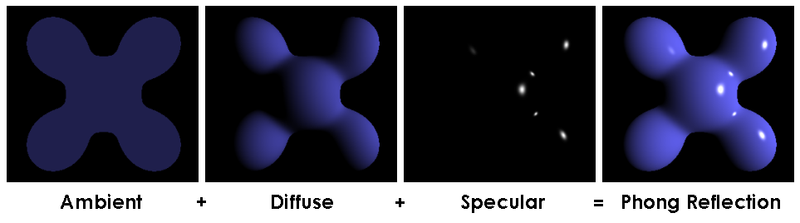
此外，由于normal 和surfaceToLight 都是单位向量，因此点积代码从：

dot(normal, surfaceToLight) / (length(normal) \* length(surfaceToLight))

简化为：

dot(normal, surfaceToLight)

环境分量ambient



旁氏反射模型的环境分量基本上指定最小亮度。即使没有光直接照射表面，环境分量也会轻微地照亮表面以阻止其变成纯黑色。所有表面的环境亮度是恒定的。

旁氏反射模型的环境分量基本上指定最小亮度。

我们将使用光源的原始强度的百分比来计算环境分量。我们将这个环境百分比存储为一个值介于零（0%）和一（100%）的浮点数在变量ambientCoefficient中。例如，如果环境系数为0.05（5%），反射光强度为（1，0，0），则为纯红光，那么环境分量将是（0.05，0，0），这是非常暗淡的红光。

计算片段着色器中的环境分量的GLSL看起来如下：

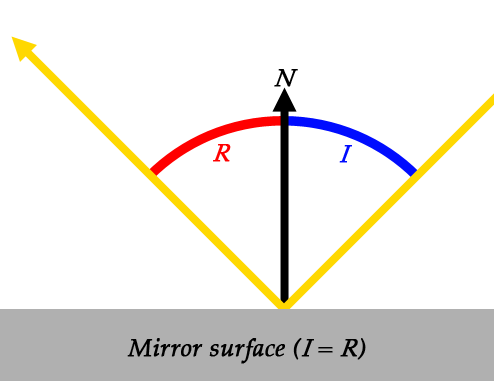
vec3 ambient = light.ambientCoefficient \* surfaceColor.rgb \* light.intensities;

这与我们在上一篇文章中实现的漫射分量相同，只是我们使用light.ambientCoefficient而不是brightness（现在称为diffuseCoefficient扩散系数）。

镜面反射分量

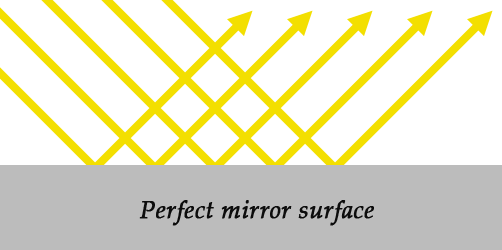
镜面组分使表面看起来有光泽。“镜面”这个词的意思是“像一面镜子”，它在这里被使用，因为闪亮的贴片（反射光）是光的假想反射，就像镜子会反射光。

先让我们先来看看镜子是如何反射光的。

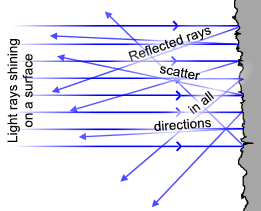


N和I是法向和入射角（AoI），这是我们在上一篇文章中看到的。R是新的，它代表反射角（AoR）。反射角是反射光线与表面法线之间的夹角。它与入射角相反。

当光入射到一个完美的镜面上时，AoI和AOR相等。也就是说，如果光以30°的角度入射，它将以30°的角度反射。

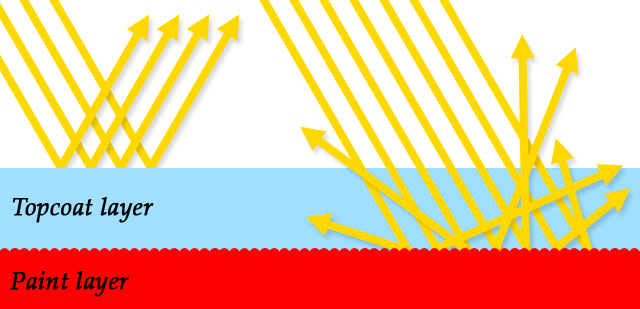


现在让我们看看那些不平整镜子的表面。



当光线照射不规则的表面时，就像上面所示的那样，光可以在任何方向反射。这是漫反射和镜面分量之间的差异：漫反射分量模型不规则表面，镜面分量模型镜面状。

许多表面都具有镜面反射和漫反射分量。汽车的外表面就是一个很好的例子。这不是一个完美的镜子，但它可以有足够的光泽看到你的反射。这是因为表面在一层透明面漆下面有一层油漆。油漆层是扩散的，但面漆层是镜面状的。面漆也很清楚，所以一些光被反射，但是一些光直射到下面的油漆层。



镜面分量通常是与漫反射分量不同的颜色。

注意，当面漆反射光线时，光线不会照射到油漆层。通常，油漆层会通过吸收一些强度来改变光线的颜色，但如果光线不触及油漆，就不会发生。这意味着镜面分量通常是与漫射分量不同的颜色。大多数镜面不吸收任何东西，它们只是反射所有的光，这意味着镜面颜色是白色的。这就是为什么汽车的发亮部分是白色的，即使油漆是红色的。

为了计算镜面分量，我们基本上要解决如下问题：如果表面是一个完美的镜子，它会把光线从光源直接反射到照相机里吗？为了得到答案，我们：

计算入射矢量，它是从光到表面的矢量。

利用AoI = AoR 规则，基于表面法线和入射矢量计算反射矢量。

计算从表面到相机的向量。

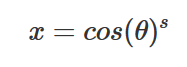
获取反射矢量与表面到摄像机矢量之间的夹角。

如果角度很小，那么我们得出结论，是的，光被直接反射到相机中。

就像漫反射分量一样，我们实际上不会计算角度。我们将只使用cos(angle) ，因为它的范围从0到1，这更有用。

这个角度有多小？这取决于表面的光泽。我们需要一个变量来表示光泽，这个变量被称为“镜面指数”。镜面指数越大，表面就越亮。这是由艺术家来把玩这个值，直到它看起来正确。

为了应用镜面指数，我们取cos(angle) 并将其提升到镜面指数的幂。这就产生了“镜面系数”，这是反射的亮度。



X是镜面系数

*θ*是角度

S是镜面指数。

我们将使用一个GLSL统一变量，称为“materialShininess ”来保存镜面指数。计算整个镜面分量的GLSL看起来是这样的：

vec3 incidenceVector = -surfaceToLight; //a unit vector

vec3 reflectionVector = reflect(incidenceVector, normal); //also a unit vector

vec3 surfaceToCamera = normalize(cameraPosition - surfacePosition); //also a unit vector

float cosAngle = max(0.0, dot(surfaceToCamera, reflectionVector));

float specularCoefficient = pow(cosAngle, materialShininess);

vec3 specularComponent = specularCoefficient \* materialSpecularColor \* light.intensities;

incidenceVector 是从光源指向表面，这是surfaceToLight的相反方向。有一个称为reflect （反射）的GLSL函数，它利用AoI = AoR规则计算基于表面法线和入射矢量的反射矢量。利用矢量减法计算surfaceToCamera ，并将其normalize为单位向量。然后我们使用点积来计算cosAngle，如前一篇文章中所描述的。然后，通过提高materialShininess 的cosAngle 来计算specularCoefficient 。

最后，通过镜面系数乘以材料的镜面颜色和光强度，得到整个镜面分量。这与漫射和环境组件非常相似，除了使用materialSpecularColor ，而不是从纹理获取表面颜色。

在本文的GLSL代码中，我们实际上使用了上面的代码的简要版本。下面是片段着色器的实际代码：

float specularCoefficient = 0.0;

if(diffuseCoefficient > 0.0)

specularCoefficient = pow(max(0.0, dot(surfaceToCamera, reflect(-surfaceToLight, normal))), materialShininess);

vec3 specular = specularCoefficient \* materialSpecularColor \* light.intensities;

注意diffuseCoefficient ＞0的检验。这是必要的，因为反射函数将反映入射向量从表面的正面和背面。表面的背面不接收任何光，所以它们根本不能发光。如果表面远离光，diffuseCoefficient 将等于零，因此我们在计算镜面系数之前先检查。

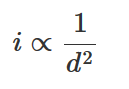
衰减

在上一篇文章的代码中，将光源移离木箱并没有使木箱显得更暗。这显然是错误的，因为如果你在现实生活中把蜡烛从表面移开，那么表面就会变暗。为了解决这个问题，我们将实施衰减。为了看到衰减的作用，按住1键，它设置了光的位置，并把相机移离木箱。如果你在前一篇文章中做了同样的事情，你会注意到表面并没有变暗，但在本文中却会。

衰减是光强度在距离上的损失。距离越大，强度越低。我们将表示衰减作为剩余光的百分比，在一个值为0和1之间的浮标中。例如，衰减值为0.2意味着80%的光强度已经丢失，并且只有20%的强度仍然存在。

衰减是光强度在距离上的损失。距离越大，强度越低。

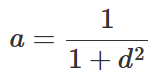
在真实世界中，衰减与距离平方的倒数成正比：



i是强度

d是距离

我们将使用这个公式的修改版本。首先，如果距离为零，我们希望避免零分误差，所以我们稍微修改公式得到：

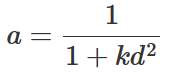


a是衰减

d是距离

现在，如果d是零，那么a将是1，这意味着光处于最大强度。

第二，我们可能想要控制强度随距离减小的速度。也许我们希望一些光在很长的距离上发光而不需要太多的衰减，而其他的光只能在短距离的情况下发光。为了控制衰减，我们将添加一个变量，我们将称之为k：



a是衰减

d是距离

k是任意“衰减因子”

上面的公式在我们的片段着色器中的实现像这样：

float attenuation = 1.0 / (1.0 + light.attenuation \* pow(distanceToLight, 2));

light.attenuation（衰减变量）是在上面的公式中的k。

我们将应用衰减到漫反射和镜面分量，而不是环境分量。记住，环境分量是一个恒定的最小亮度，所以降低亮度低于最小值是没有意义的。

GLSL综合环境分量，漫反射分量和镜面分量，包括衰减，的实现如下：

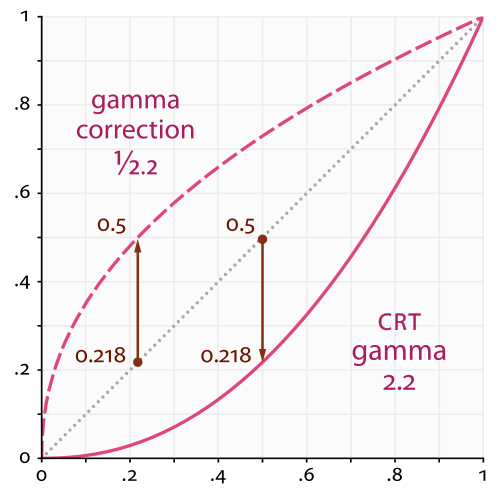
vec3 linearColor = ambient + attenuation\*(diffuse + specular);

这几乎是片段/像素的最终颜色。只差最后一步是做伽马校正。

伽马校正

到目前为止，我们所有的照明计算都假设我们在使用线性色彩空间。在线性色彩空间中，如果你把RGB颜色值变为原来两倍，那么屏幕上的像素应该是亮度的两倍。例如，100%的红色（1，0，0）应该是50%倍红色（0.5，0，0）的两倍。

问题是，计算机屏幕不能在线性颜色空间中显示颜色。100%红实际上比50%红亮4.5倍，这使得3D场景中的亮度看起来是错误的。这是因为计算机监视器模仿老CRT监视器的行为方式。我也曾读到，它与CRT显示器无关，它因为人眼感知亮度的方式而变得必要的。无论哪种方式，伽马校正将允许我们调整3D场景的亮度以使其看起来更好。



上图中，中间的虚线表示我们正在工作的线性颜色空间。底部的实线表示计算机显示器显示的颜色空间。注意虚线上的0.5如何匹配实线上的0.218。这意味着，如果我们计算一个RGB值为0.5，当它在显示器上显示时，它看起来实际上是0.218，所以所有的东西看起来都太暗了。顶部的虚线表示伽马校正后的RGB值，这增加了亮度。伽马校正的颜色太亮，显示器颜色太暗，所以当它们组合时，结果看起来才是正确的。

还要注意这些线是如何在0和1相遇的。这意味着伽马校正不会影响最大亮度和最小亮度——它会影响所有中间的亮度。

伽马校正是一种改变RGB颜色亮度的操作。我们在线性色彩空间中进行所有的照明计算，然后在进行屏幕显示之前，进行伽马校正以调整颜色。这使得所有的照明看起来都是正确的，而不是看起来太暗。

伽马校正是一种改变RGB颜色亮度的操作。

伽马校正很容易实现。你把每个RGB值提升到gamma的幂。一些游戏给用户一个亮度设置，允许他们改变伽马值，但是我们将只使用本文中的常数值1/2.2，这是CRT监视器的正确值。

我们将使用GLSL函数pow进行伽马校正。GLSL实现如下：

vec3 gamma = vec3(1.0/2.2);

finalColor = pow(linearColor, gamma);

pow函数将第一个参数提升为第二个参数的幂。它可以以数字或向量作为参数。如果参数是向量，它将第一向量的每个元素加到第二向量的每个元素。上面的代码只是一个较短的写作方式：

vec3 gamma = vec3(1.0/2.2, 1.0/2.2, 1.0/2.2);

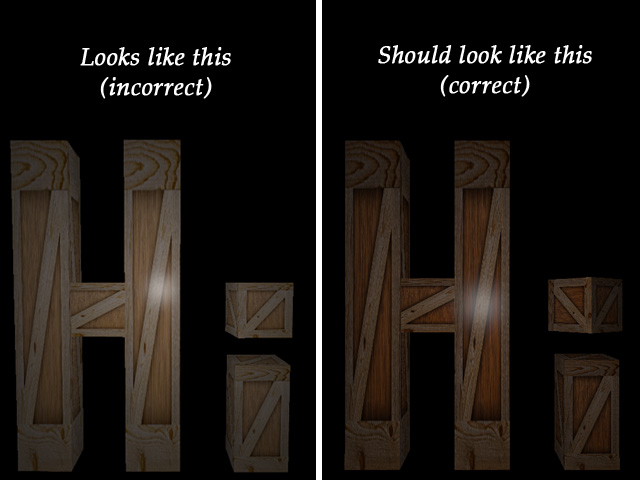
finalColor = vec3(pow(linearColor.r, gamma.r),

pow(linearColor.g, gamma.g),

pow(linearColor.b, gamma.b));

它采用线性颜色矢量的红色、绿色和蓝色分量，并将它们全部提升到1／2.2的幂。

在实施伽马校正和运行程序后，它看起来实在太亮了。



伽马校正的图像看起来不对，但是正确的图像看起来像最后一篇文章中的一个，没有任何伽马校正。这是因为我们实际上做了两次伽马校正。没错，这篇文章之前我们已经有了一些伽马校正。大多数图像文件格式已经包含伽马校正的RGB值。我们作为纹理使用的wooden-crate.jpg 文件已经被伽马校正，然后伽玛第二次校正它使它看起来太亮。我们可以不执行伽马校正，这将使纹理看起来正确，然后光照会看起来不对。

我们将取消纹理加载时的伽马校正。谢天谢地，这在OpenGL中是非常简单的。跟大多数图像医院，我们的木箱图像是在SRGB颜色空间，已经做了伽马校正。我们可以改变glTexImage2D 的内部格式化参数，告诉OpenGL纹理数据已经在SRGB颜色空间中，OpenGL将自动将像素校正为线性颜色空间。对于RGB像素数据，我们将内部格式从GL\_RGB转换为GL\_SRGB。对于RGBA像素数据，我们将其从GL\_RGBA转换为 GL\_SRGB\_ALPHA。在对tdogl::Texture类进行此更改之后，3D场景再次看起来正确。

片段着色器代码更改

下面是整个片段着色器：

#version 150

uniform mat4 model;

uniform vec3 cameraPosition;

// material settings

uniform sampler2D materialTex;

uniform float materialShininess;

uniform vec3 materialSpecularColor;

uniform struct Light {

vec3 position;

vec3 intensities; //a.k.a the color of the light

float attenuation;

float ambientCoefficient;

} light;

in vec2 fragTexCoord;

in vec3 fragNormal;

in vec3 fragVert;

out vec4 finalColor;

void main() {

vec3 normal = normalize(transpose(inverse(mat3(model))) \* fragNormal);

vec3 surfacePos = vec3(model \* vec4(fragVert, 1));

vec4 surfaceColor = texture(materialTex, fragTexCoord);

vec3 surfaceToLight = normalize(light.position - surfacePos);

vec3 surfaceToCamera = normalize(cameraPosition - surfacePos);

//ambient

vec3 ambient = light.ambientCoefficient \* surfaceColor.rgb \* light.intensities;

//diffuse

float diffuseCoefficient = max(0.0, dot(normal, surfaceToLight));

vec3 diffuse = diffuseCoefficient \* surfaceColor.rgb \* light.intensities;

//specular

float specularCoefficient = 0.0;

if(diffuseCoefficient > 0.0)

specularCoefficient = pow(max(0.0, dot(surfaceToCamera, reflect(-surfaceToLight, normal))), materialShininess);

vec3 specular = specularCoefficient \* materialSpecularColor \* light.intensities;

//attenuation

float distanceToLight = length(light.position - surfacePos);

float attenuation = 1.0 / (1.0 + light.attenuation \* pow(distanceToLight, 2));

//linear color (color before gamma correction)

vec3 linearColor = ambient + attenuation\*(diffuse + specular);

//final color (after gamma correction)

vec3 gamma = vec3(1.0/2.2);

finalColor = vec4(pow(linearColor, gamma), surfaceColor.a);

}

有一种新的称为cameraPosition的统一变量，这是计算镜面分量所必需的。您可能会遇到其他没有相机位置的片段着色器。如果照明计算是在摄像机空间中完成的，而不是世界空间，那么摄像机的位置总是原点（0，0，0）。 前述方式是相当常见的，但是我们将继续在世界空间中工作，因为它直观，类似于上一篇文章中的代码。

Light 结构体有两个新的元素：attenuation 和ambientCoefficient。attenuation 变量是k，即我们在本文中较早的衰减部分看到的值。ambientCoefficient 变量是用于环境组件的光色值的百分比，这也在本文中早些时候解释过。

还有新的材料统一变量：

uniform sampler2D materialTex;

uniform float materialShininess;

uniform vec3 materialSpecularColor;

在以前的文章中，materialTex 变量与tex相同，更名是为与其他材质的统一变量相适应。materialShininess 变量是镜面指数。materialSpecularColor 变量决定了光在镜面计算中的吸收和反射。请记住，镜面反射和漫射颜色通常是不同的。漫反射颜色来自纹理（materialTex），镜面颜色来自materialSpecularColor变量。

注意：在本文的前一版本中，材料统一变量处于一种称为Material的GLSL结构中。不幸的是，GLSL 1.5不允许内部结构中的采样器（如materialTex），一些驱动程序不会编译片段着色器。

着色器的main 函数是通过计算几个变量，然后代码的每一段对应于本文中的一节：

计算环境分量

计算漫反射分量

计算镜面分量

计算衰减

结合环境，扩散和镜面组件，衰减应用

执行伽马校正

C++代码的变化

本文中的大部分更改都是在片段着色器中进行的。C++的变化只有在片段着色器设置新的制服。

ModelAsset 结构体添加具有用于闪烁和镜面颜色的新变量：

struct ModelAsset {

tdogl::Program\* shaders;

tdogl::Texture\* texture;

GLuint vbo;

GLuint vao;

GLenum drawType;

GLint drawStart;

GLint drawCount;

GLfloat shininess; //new this article

glm::vec3 specularColor; //new this article

};

Light 结构体添加具有衰减和环境系数的新变量：

struct Light {

glm::vec3 position;

glm::vec3 intensities;

float attenuation; //new this article

float ambientCoefficient; //new this article

};

在LoadWoodenCrateAsset的内部我们设置新的材料变量：

gWoodenCrate.shininess = 80.0;

gWoodenCrate.specularColor = glm::vec3(1.0f, 1.0f, 1.0f);

80的光泽意味着板条箱会很有光泽。镜面颜色被设置为白色，这是最常见的镜面颜色。

在AppMin内部，我们设置了新的光源变量：

gLight.attenuation = 0.2f;

gLight.ambientCoefficient = 0.005f;

将环境系数设置为0.005意味着最小亮度是最大亮度的0.5%。这应该使板条箱的后部很暗，但不是纯黑色。衰减变量没有任何特定的值-我只是调整它，直到我发现一个看起来不错的值。

所有新的着色器统一变量都设置在RenderInstance 函数中，如下：

//set the shader uniforms

shaders->setUniform("camera", gCamera.matrix());

shaders->setUniform("model", inst.transform);

shaders->setUniform("materialTex", 0);

shaders->setUniform("materialShininess", asset->shininess);

shaders->setUniform("materialSpecularColor", asset->specularColor);

shaders->setUniform("light.position", gLight.position);

shaders->setUniform("light.intensities", gLight.intensities);

shaders->setUniform("light.attenuation", gLight.attenuation);

shaders->setUniform("light.ambientCoefficient", gLight.ambientCoefficient);

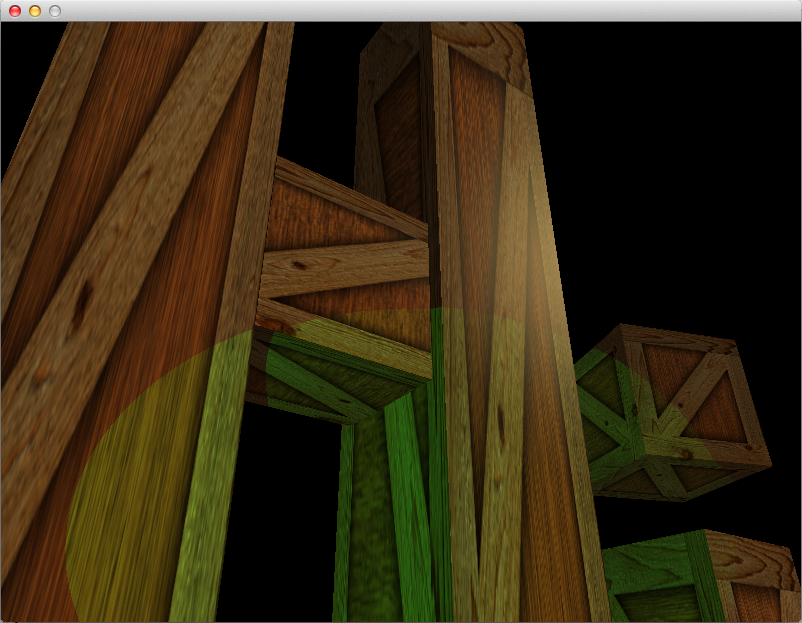
shaders->setUniform("cameraPosition", gCamera.position());

在本文中早些时候伽马校正部分解释，tdogl/Texture.cpp也有一个小的变化，以便在加载时撤消纹理上的伽马校正。它现在通过GL\_SRGB 而不是GL\_RGB 作为glTexImage2D 函数的第三个参数。这将使OpenGL撤消纹理上的伽马校正，以便我们可以在片段着色器中重新进行伽马校正。

未来文章一窥

在下一篇文章中，我们将实现方向灯和聚光灯。我们也将同时获得多个灯工作。

# [Modern OpenGL 08 – Even More Lighting: Directional Lights, Spotlights, & Multiple Lights](https://www.tomdalling.com/blog/modern-opengl/08-even-more-lighting-directional-lights-spotlights-multiple-lights/)



在本文中，我们将添加方向灯，聚光灯，并允许多个灯，而不是一个。这是最后一篇关于照明的文章——至少有一段时间。

访问代码

从这里下载所有代码作为一个zip：

<https://github.com/tomdalling/opengl-series/archive/master.zip>

本系列文章中的所有代码都可以从github:

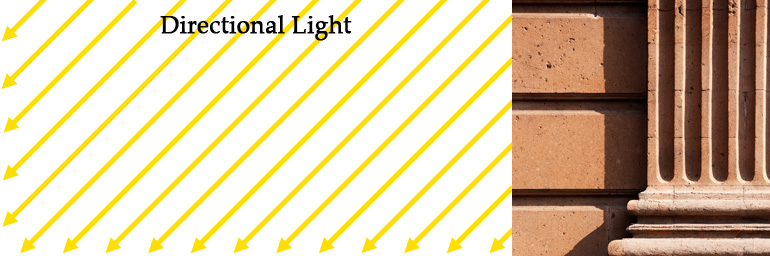
<https://github.com/tomdalling/opengl-series>获得。您可以从该页面下载所有文件的zip，或者您可以在熟悉Git的情况下克隆存储库。

本文基于前一篇文章的代码。

这篇文章的代码可以在[source/08\_even\_more\_lighting](https://github.com/tomdalling/opengl-series/tree/master/source/08_even_more_lighting) 文件夹中找到。在OS X上，打开根文件夹中的opengl-series.xcodeproj文件，并选择与本文对应的目标。在Windows上，在VisualStudio 2013中打开opengl-series.sln文件，并打开与本文对应的项目。

该项目包括所有的依赖关系，因此您不必安装或配置任何额外的东西。如果你在编译和运行代码方面任何问题，请联系我。

定向光源



定向光源是在单一、均匀的方向上发光的光源。也就是说，所有的光线都是平行的。纯定向光不存在（除了激光器之外）？但是它们经常被用在计算机图形学中，模拟非常遥远的强光源，比如太阳。太阳向各个方向放射光，就像点光源一样。然而，在一个巨大的距离上，光线到达地球的微小部分看起来几乎是平行的。

正如我们在上一篇[关于均匀坐标](https://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/)的文章中看到的，定向光可以被认为是无限远的点光源。这导致了与衰减的不幸交互作用。衰减是光强度在距离上的减小-距离越大，光越暗。如果有最小的衰减量，在无限远处，光就变得无限暗淡（即看不见）。出于这个原因，实现定向光源，使得它们忽略衰减。如果我们用方向性的光来代表太阳，这种感觉是有意义的，因为我们在地球上看到的阳光并没有减弱。也就是说，阳光并不会越靠近地面，就越变暗。

定向光源的实现使得它们忽略衰减。

与点光源不同的是，定向光源不需要位置坐标。定向光源只需要一个代表所有光线方向的三维矢量。然而，我们着色器中的GLSL照明代码期望每个光都有一个位置。幸运的是，我们可以通过设置W＝0来表示具有均匀坐标的方向光的方向。如前面[关于均匀坐标的文章](https://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/)所解释的，当W＝0在4D坐标中时，它代表朝向无限远的点的方向。因此，如果坐标代表朝向太阳的方向，我们可以否定它来产生远离太阳的方向，这是光线的方向。使用这种表示位置矢量内的方向性光的方法，我们可以有这样的代码：

我们可以通过设置W＝0来表示具有均匀坐标的方向光的方向。

uniform vec4 lightPosition;

// check if this is a directional light

if(lightPosition.w == 0.0) {

// it is a directional light.

// get the direction by converting to vec3 (ignore W) and negate it

vec3 lightDirection = -lightPosition.xyz;

} else {

// NOT a directional light

}

作为一个例子，我们假设lightPosition 坐标是（1，0，0，0），它代表正X轴上无限远的点。如果我们取x，y，z值，然后对它们取相反数，我们得到（-1，0，0），这表明光沿着x轴负方向上发光。

我们在前一篇文章中实现的GLSL照明代码使用从表面到光的方向（在一个名为surfaceToLight的变量）中，所以我们实际上不需要对任何值取反。本文中的定向光源的GLSL看起来是这样的：

vec3 ApplyLight(Light light, vec3 surfaceColor, vec3 normal, vec3 surfacePos, vec3 surfaceToCamera)

{

vec3 surfaceToLight;

float attenuation = 1.0;

if(light.position.w == 0.0) {

//directional light

surfaceToLight = normalize(light.position.xyz);

attenuation = 1.0; //no attenuation for directional lights

} else {

// NOT directional light

//... (code ommited)

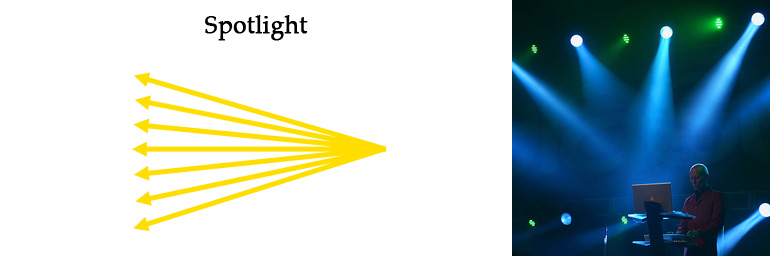
}

// the rest of the lighting calculation

//... (code ommited)

}

聚光灯



我们将实施的最后一个类型的光源是聚光灯。聚光灯与点光源非常相似，除了没有向所有方向辐射光外，聚光灯的光线被限制成锥形。光线不像定向光源那样平行，但它们照射着一个大致的方向。想想手电筒，有一个灯泡像点光源一样，但也有一个反射的曲面限制了光线照射的方向。

我们把聚光灯看做具有锥形限制的点光源。

我们把聚光灯看做具有锥形限制的点光源。适用于点光源的一切也适用于聚光灯，除了两个额外的变量：圆锥的方向和圆锥的角度。这里是本文的轻结构的定义：

uniform struct Light {

vec4 position;

vec3 intensities;

float attenuation;

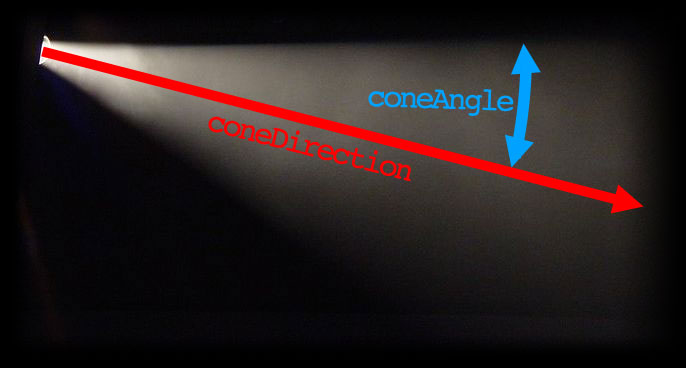
float ambientCoefficient;

float coneAngle; // new

vec3 coneDirection; // new

};

为了实现聚光灯，增加了两个新的变量：coneAngle 圆锥角和coneDirection圆锥方向。coneDirection圆锥方向变量是从圆锥体的顶点穿过圆锥体的中心的方向。coneAngle 圆锥角变量是中心线和圆锥面之间的角度。



实现相当简单。对于每个像素，我们检查它是在光锥的内部还是外部。如果它在里面，我们继续照明计算，我们通常会。如果像素在锥体之外，那么我们将衰减因子设置为零，这将使光线不可见。下面是实现锥约束的GLSL：

如果像素在锥体之外，那么我们将衰减因子设置为零。

float lightToSurfaceAngle = degrees(acos(dot(-surfaceToLight, normalize(light.coneDirection))));

if(lightToSurfaceAngle > light.coneAngle){

attenuation = 0.0;

}

第一行相当紧凑，所以让我解释一下，把它分解成更小的步骤：

// 1. Get the direction for the center of the cone. The `normalize`

// function is called just in case `light.coneDirection` isn't

// already a unit vector.

vec3 coneDirection = normalize(light.coneDirection);

// 2. Get the direction of the ray of light. This is the opposite

// of the direction from the surface to the light.

vec3 rayDirection = -surfaceToLight;

// 3. Get the angle between the center of the cone and the ray of light.

// The combination of `acos` and `dot` return the angle in radians, then

// we convert it to degrees.

float lightToSurfaceAngle = degrees(acos(dot(rayDirection, coneDirection)))

// 4. Check if the angle is outside of the cone. If so, set the attenuation

// factor to zero, to make the light ray invisible.

if(lightToSurfaceAngle > light.coneAngle){

attenuation = 0.0;

}

锥形限制将应用于所有非定向光源，包括点光源。为了停止应用于点光源的限制，我们所要做的就是将coneAngle 圆锥角设置为180.0，这意味着光可以向各个方向发光。

多光源

前述文章中的片段着色器只实现了一个光，现在我们用一组光替换单个光。然后，我们可以在片段着色器中循环执行它们，并将它们全部应用到场景中。我们在这里并没有实现任何新的东西。我们只是重构现有的片段着色器代码并添加一个循环。

第一步是从碎片着色器中移除单一的光均匀，并用一个数组替换它。这是前面的文章中的旧的GLSL：

// code from previous article (a single light)uniform struct Light {

vec3 position;

vec3 intensities; //a.k.a the color of the light

float attenuation;

float ambientCoefficient;

} light;

下面是一个新的代码，具有一组光源：

// array of lights

#define MAX\_LIGHTS 10

uniform int numLights;

uniform struct Light {

vec4 position;

vec3 intensities; //a.k.a the color of the light

float attenuation;

float ambientCoefficient;

float coneAngle;

vec3 coneDirection;

} allLights[MAX\_LIGHTS];

此实现与OpenGL的旧固定函数流水线中的照明实现非常类似。最大光源数量设置在MAX\_LIGHTS 常数。然后有一个统一的变量，保持灯阵列，称为allLights。最后，有一个统一的变量来保存我们实际使用的灯的数量，称为numLights。

下一步是重构GLSL代码，使其在数组上循环。我们将所有的照明代码提取到一个名为ApplyLight的函数中，该函数对单个光进行整个照明计算。以下是ApplyLight 函数的GLSL：

vec3 ApplyLight(Light light, vec3 surfaceColor, vec3 normal, vec3 surfacePos, vec3 surfaceToCamera)

{

vec3 surfaceToLight;

float attenuation = 1.0;

if(light.position.w == 0.0) {

//directional light

surfaceToLight = normalize(light.position.xyz);

attenuation = 1.0; //no attenuation for directional lights

} else {

//point light

surfaceToLight = normalize(light.position.xyz - surfacePos);

float distanceToLight = length(light.position.xyz - surfacePos);

attenuation = 1.0 / (1.0 + light.attenuation \* pow(distanceToLight, 2));

//cone restrictions (affects attenuation)

float lightToSurfaceAngle = degrees(acos(dot(-surfaceToLight, normalize(light.coneDirection))));

if(lightToSurfaceAngle > light.coneAngle){

attenuation = 0.0;

}

}

//ambient

vec3 ambient = light.ambientCoefficient \* surfaceColor.rgb \* light.intensities;

//diffuse

float diffuseCoefficient = max(0.0, dot(normal, surfaceToLight));

vec3 diffuse = diffuseCoefficient \* surfaceColor.rgb \* light.intensities;

//specular

float specularCoefficient = 0.0;

if(diffuseCoefficient > 0.0)

specularCoefficient = pow(max(0.0, dot(surfaceToCamera, reflect(-surfaceToLight, normal))), materialShininess);

vec3 specular = specularCoefficient \* materialSpecularColor \* light.intensities;

//linear color (color before gamma correction)

return ambient + attenuation\*(diffuse + specular);

}

ApplyLight 函数以单个光作为参数，但它也需要一些描述正在被照亮的表面的参数：surfaceColor、normal、surfacePos和surfaceToCamera。因为所有的光都作用在同一个表面上，所以我们计算这些表面相关的变量一次，并把它们作为每个光源的参数。

通过将所有的照明代码提取成一个功能，我们可以循环所有的光源。对于每个光源，我们调用ApplyLight ，并将所有结果加在一起以获得表面的颜色：

vec3 linearColor = vec3(0);

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

linearColor += ApplyLight(allLights[i], surfaceColor.rgb, normal, surfacePos, surfaceToCamera);

}

C++代码的变化

对C++代码主要只是反映了GLSL的变化。

Light 结构略有变化，以适应定向光源和聚光灯。位置元素从glm::vec3变为glm::vec4，并且添加两个聚光锥变量coneDirection 圆锥方向和coneAngle圆锥角。

struct Light {

glm::vec4 position;

glm::vec3 intensities; //a.k.a. the color of the light

float attenuation;

float ambientCoefficient;

float coneAngle; // new

glm::vec3 coneDirection; // new

};

在全局变量，我们将单光源变为 std::vector容器列表 ：

// Light gLight; // used to be this in previous article

std::vector<Light> gLights;

在RenderInstance 函数中，我们现在必须在设置统一变量uniform时循环遍历所有的光源：

shaders->setUniform("numLights", (int)gLights.size());

for(size\_t i = 0; i < gLights.size(); ++i){

SetLightUniform(shaders, "position", i, gLights[i].position);

SetLightUniform(shaders, "intensities", i, gLights[i].intensities);

SetLightUniform(shaders, "attenuation", i, gLights[i].attenuation);

SetLightUniform(shaders, "ambientCoefficient", i, gLights[i].ambientCoefficient);

SetLightUniform(shaders, "coneAngle", i, gLights[i].coneAngle);

SetLightUniform(shaders, "coneDirection", i, gLights[i].coneDirection);

}

这使用了一个名为SetLightUniform的函数，它实现使用结构体元素和索引而构造了统一变量（例如"allLights[2].coneAngle"）。

template <typename T>

void SetLightUniform(tdogl::Program\* shaders, const char\* propertyName, size\_t lightIndex, const T& value)

{

std::ostringstream ss;

ss << "allLights[" << lightIndex << "]." << propertyName;

std::string uniformName = ss.str();

shaders->setUniform(uniformName.c\_str(), value);

}

在AppMain，我们创建了聚光灯和定向光源：

// setup lights

Light spotlight;

spotlight.position = glm::vec4(-4,0,10,1);

spotlight.intensities = glm::vec3(2,2,2); //strong white light

spotlight.attenuation = 0.1f;

spotlight.ambientCoefficient = 0.0f; //no ambient light

spotlight.coneAngle = 15.0f;

spotlight.coneDirection = glm::vec3(0,0,-1);

Light directionalLight;

directionalLight.position = glm::vec4(1, 0.8, 0.6, 0); //w == 0 indications a directional light

directionalLight.intensities = glm::vec3(0.4,0.3,0.1); //weak yellowish lightdirectionalLight.ambientCoefficient = 0.06;

gLights.push\_back(spotlight);

gLights.push\_back(directionalLight);

在Update 函数中，我们使键1改变聚光灯的位置和方向：

//move light

if(glfwGetKey('1')){

gLights[0].position = glm::vec4(gCamera.position(), 1.0);

gLights[0].coneDirection = gCamera.forward();

}

再谈光照

在这一点上，我们基本上已经实现了在OpenGL的旧的固定函数API中可用的所有照明选项。我们本应进一步实施逐片段照明代替逐顶点照明，但是还有更多的改进我们还没有讲到。光照是一个巨大的课题，所以现在我要简要地介绍一些先进的光照主题。

布林-旁氏反射模型

本系列中的照明是旁氏反射模型的一种实现。这个算法有一个更好的版本，叫做[布林-旁氏反射模型](http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model)。布林-旁氏反射模型更精确，性能也稍好一些。

更华丽的聚光灯

本文中的聚光灯实现是非常基础的。它可以通过使用[混合函数](https://www.opengl.org/sdk/docs/man/html/mix.xhtml)软化GLSL中的硬边来改善。你也可以采样手电筒的纹理来阻止光线看起来如此圆滑和平坦。

延迟渲染

我们实现照明的方式被称为向前渲染，并且有一些恼人的问题与之相关。首先，我们能拥有的灯的数量是有限的。其次，每一个被绘制的像素都需要计算每一个光——即使是不影响像素的光——这可能是一个性能问题。理想情况下，我们希望能拥有数千盏性能良好的灯。为了解决这些问题，我们可以使用[延迟渲染](http://gamedev.stackexchange.com/questions/74/what-is-deferred-rendering)。

延迟渲染是一种将渲染分成多个通道的技术。第一通道呈现没有任何光的几何形状。在随后的通道中，光源每次渲染一个。光照作用在每一个像素上，而不是每个片段，从而提高性能。还可以限制每个光源所影响的像素以进一步提高性能。

综上所述，延迟渲染允许大量的光具有良好的性能，但它比前向渲染更复杂。

阴影

尽管每个人都喜欢它们的外表，但是漂亮的阴影是尚未解决的复杂问题，极其复杂。以下所有的技术都比这一系列文章所涵盖的要复杂得多。

[阴影贴图](http://codeflow.org/entries/2013/feb/15/soft-shadow-mapping/)可能是最简单的技术，但是产生的阴影可能是奇怪的和像素化的。

[阴影体积](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch11.html)提供逐像素完美的结果，但需要复杂的几何处理，而此运算速度很慢。

当光和几何都是静态的，那么[光照图](http://en.wikipedia.org/wiki/Lightmap)是准确的，并且具有良好的性能。

甚至不要让我开始讲[环境光遮蔽](http://gamedev.stackexchange.com/questions/23/what-is-ambient-occlusion)。

高动态范围图像（HDR）



注意在我们的片段着色器中，所有颜色的颜色是如何叠加在一起的。RGBA值应该在0到1的范围内，那么如果有大量的光，那么总和会大于1，会发生什么？颜色会被夹住，看起来怪怪的。此外，如果灯光太暗，整个场景看起来基本上是黑色的，没有细节。高动态范围（HDR）渲染有助于解决这些问题。

人眼的调节取决于它所观察到的亮度。如果你在黑暗的房间里呆的时间够长，你的瞳孔会扩张，让更多的光线进入你的视网膜，这使得房间看起来更明亮。如果你走到外面阳光明媚的地方，正好相反，这样你就不会因为强烈的光线而失明。HDR渲染类型模仿你的眼睛是如何工作的，以保持细节在非常黑暗和非常明亮的场景中可见。在照明计算中允许RGB值超过1，然后将这些值重新调整，使它们很好地匹配在0到1范围内。

地下散射



光不只是反射表面，它也可以穿过它们。当光穿透表面时，它改变了表面的颜色。渲染这种颜色变化称为地下散射。

即使皮肤看起来相当不透明，但通常使用地下散射来真实地绘制人体皮肤。没有它，皮肤往往看起来像涂了塑料。

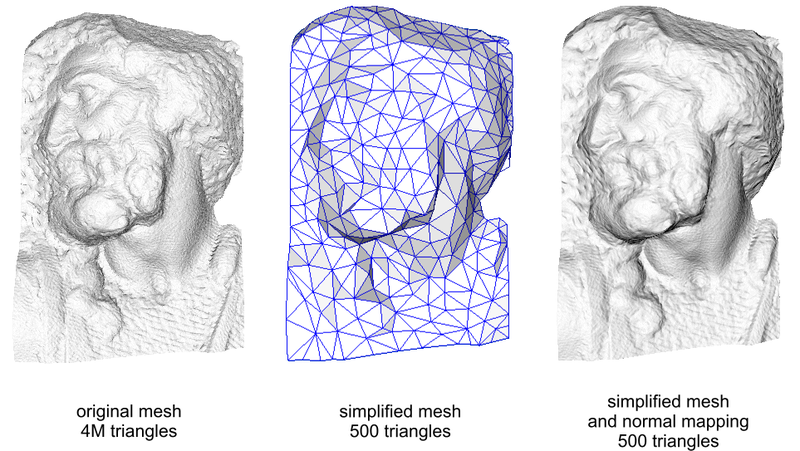
发光表面



在我们的实现中，只是光源照亮表面。然而，一些表面提供了它们自己的照明，这使得它们看起来像是发光的。假想一个萤火虫，在黑暗的贴纸上发光，或者那些怪异发光的蘑菇。

发射光源很容易在OpenGL中实现。向着色器发送额外的颜色统一变量，连同材质和光泽，并将该颜色添加到最终颜色。或者，你可以发送一个额外的纹理而不是单一的颜色。

法向贴图



由于性能原因，3D网格对它们所能包含的顶点的数量有限制。可以取走很多顶点使表面几何形状粗糙或凹凸不平一点，因此通常使用法向贴图。法向贴图可以用来使一个看起来更小的角度来看3D模型，更真实一些。

法线贴图是影响表面法线的纹理。这就像我们已经实现的表面纹理类似，只是它包含XYZ矢量而不是RGB颜色。表面法线是照明计算的重要组成部分，它影响每个像素的亮度。

结论

这就覆盖了我们目前的照明实施。在添加方向灯和聚光灯之后，我们重新创建了旨在替代旧的固定函数OpenGL API的所有可用功能。