

LearnOpenGL CN

None

None

None

Table of contents

1. 欢迎来到OpenGL的世界	3
1.1 为什么要阅读这些教程呢?	3
1.2 你将学会什么呢?	3
1.3 关于中文翻译	3
2. 目录	5
2.1 简介	5
2.2 入门	7
2.3 光照	106
2.4 模型加载	155
2.5 高级OpenGL	171
2.6 高级光照	282
2.7 PBR	407
2.8 实战	471
2.9 历史存档	565
3. 代码仓库	574

1. 欢迎来到OpenGL的世界

欢迎来到OpenGL的世界。这个工程只是我([Joey de Vries](#))的一次小小的尝试，希望能够建立起一个完善的OpenGL教学平台。无论你学习OpenGL是为了学业，找工作，或仅仅是因为兴趣，这个网站都将能够教会你现代(Core-profile) OpenGL从基础，中级，到高级的知识。LearnOpenGL的目标是使用易于理解的形式，使用清晰的例子，展现现代OpenGL的所有知识点，并与此同时为你以后的学习提供有用的参考。

如果您喜欢这个系列教程的话，不妨向Joey de Vries的[Paypal](#)进行捐赠，支持一下作者，让这个教程能够持续完善并更新。

1.1 为什么要阅读这些教程呢？

在互联网上，有关学习OpenGL的有成千上万的文档与资源，然而其中大部分的资源仅仅讨论了OpenGL的立即渲染模式（Immediate Mode，通常会说旧OpenGL），亦或是不完整，缺少适当的文档，甚至是仅仅不适合你的口味。所以，我的目标是提供一个既完整，又易懂的平台供人们学习。

如果你很享受那些提供手把手指导的教程，那些提供清晰例子的教程，以及那些不会一下将你淹没在细节中的教程，那么我的这些教程很可能就很适合你。我的教程旨在让那些没有图形编程经验的人们能够理解，又让那些有经验的读者有阅读下去的兴趣。我的教程同样也讨论了一些常用的概念，只需要你再有一点创造力，就能将你的想法变成真正的3D程序。如果如果你觉得前面这些讲的都是你，欢迎继续阅读我的教程。



1.2 你将学会什么呢？

我这些教程的核心是现代OpenGL。学习（和使用）现代OpenGL需要用户对图形编程以及OpenGL的幕后运作有非常好的理解才能在编程中有很好的发挥。所以，我们会首先讨论核心的图形学概念，OpenGL怎样将像素绘制到屏幕上，以及如何利用黑科技做出一些很酷的效果。

除了核心概念之外，我们还会讨论许多有用的技巧，它们都可以用在你的程序中，比如说在场景中移动，做出漂亮的光照，加载建模软件导出的自定义模型，做一些很酷的后期处理技巧等。最后，我们也将会使用我们已学的知识从头开始做一个小游戏，让你真正体验一把图形编程的魅力。

1.3 关于中文翻译

这里是LearnOpenGL教程的中文翻译，英文版的地址为：<https://learnopengl.com/>

由于翻译可能无法做到精确表达原文意思，我们推荐您在对问题有疑惑的时候去阅读一下英文版的教程。如果您对翻译有更好的建议，可以去我们的[GitHub工程](#)上提交Issue或者Pull Request。如果是对教程的内容有问题，请先查看原文，如果不是翻译错误的话，请直接在原网站评论区向作者（JoeyDeVries）反馈。

如果教程中的源码无法打开的话，可以到教程的[GitHub页面](#)上去寻找所需的代码，每一节教程的源码以及练习都位于 `src` 目录下的对应章节目录中。

2. 目录

2.1 简介

你到这里来可能是想学习计算机图形的工作原理，并且自己做一些很酷的东西。自己做东西是非常有趣的，同样也能给你带来对图形编程的兴趣。然而，在你开始学习旅程之前，有这么几点注意事项。

2.1.1 前置知识

由于OpenGL是一个图形API，并不是一个独立的平台，它需要一个编程语言来工作，在这里我们使用的是C++。所以，对C++的熟练掌握在学习这个教程中是必不可少的。当然，我仍将尝试解释大部分用到的概念，包括一些高级的C++话题，所以，你并不一定要是一个C++专家才能来学习。不过，请确保你至少应该能写个比‘Hello World’复杂的程序。如果你对C++不是很熟悉，我推荐您学习一下www.learnccpp.com上的免费教程。

除此之外，我们也将用到一些数学知识（线性代数、几何、三角学），同样我也会尝试解释所有的必备的数学概念。但是，毕竟我不是一个数学家，即使我的解释可能会很容易理解，但是这些解释都是不全面的。所以，在必须的时候我会链接一些不错的资料，它们会将这些概念解释的更加全面。不要被必须的数学知识吓到了，几乎所有的概念只要有基础的数学背景都可以理解。我也会将数学的内容压缩至极限。大部分的功能甚至都不需要你理解所有的数学知识，只要你会使用就行。

2.1.2 结构

LearnOpenGL被分解成了许多大的主题。每个主题包括一些小节，每个小节中会对不同的概念进行详细的解释。所有的主题都可以在目录中找到。这些主题是按照线性来学习的（所以建议从上到下来读，除非有特殊指示），每个页面将会解释每个概念的背景理论和实际操作。

为了让教程更容易理解，结构更鲜明，本站采用了方框和代码块。

方框

绿色方框是一些注释或者是对于OpenGL或讨论主题有用的特性/提示。

红色方框是一些警告或者一些你需要特别注意的特性。

译注

蓝色方框是翻译时为了帮助读者理解附加的一些信息。

代码

你在网站中将会看到很多小片的代码，它们将会在下面这样的代码框中：

```
// 这个方框是代码
```

由于这样只提供了代码的片段，当需要的时候我会提供个链接到当前工程的源代码的。

颜色标记

有一些词语会以不同颜色显示出来，用来表示这些词语有不同的意义：

- 定义：绿色的字是定义，即一个重要的概念或名称，这些词语你能经常见到。
- 程序逻辑：红色的字是函数的名称或者是类名。
- 变量：蓝色的字是变量，包括所有的OpenGL常量。

现在你应该对这个网站的结构有一些了解了，你现在可以进入「入门」这一章，开始你的OpenGL学习生涯吧！

2.2 入门

2.2.1 OpenGL

原文 [OpenGL](#)

作者 JoeyDeVries

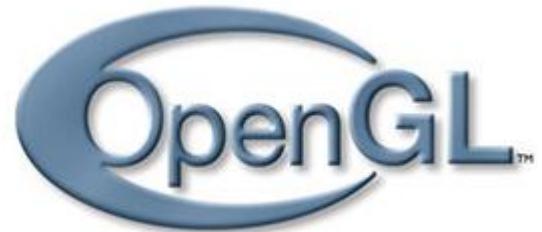
翻译 gji_1992, Krasjet

校对 暂未校对

在开始这段旅程之前我们先了解一下OpenGL到底是什么。一般它被认为是一个API([Application Programming Interface](#), 应用程序编程接口), 包含了一系列可以操作图形、图像的函数。然而, OpenGL本身并不是一个API, 它仅仅是一个由[Khronos组织](#)制定并维护的规范(Specification)。

OpenGL规范严格规定了每个函数该如何执行, 以及它们的输出值。至于内部具体每个函数是如何实现(Implement)的, 将由OpenGL库的开发者自行决定(译注: 这里开发者是指编写OpenGL库的人)。因为OpenGL规范并没有规定实现的细节, 具体的OpenGL库允许使用不同的实现, 只要其功能和结果与规范相匹配(亦即, 作为用户不会感受到功能上的差异)。

实际的OpenGL库的开发者通常是显卡的生产商。你购买的显卡所支持的OpenGL版本都为这个系列的显卡专门开发的。当你使用Apple系统的时候, OpenGL库是由Apple自身维护的。在Linux下, 有显卡生产商提供的OpenGL库, 也有一些爱好者改编的版本。这也意味着任何时候OpenGL库表现的行为与规范规定的不一致时, 基本都是库的开发者留下的bug。



由于OpenGL的大多数实现都是由显卡厂商编写的, 当产生一个bug时通常可以通过升级显卡驱动来解决。这些驱动会包括你的显卡能支持的最新版本的OpenGL, 这也是为什么总是建议你偶尔更新一下显卡驱动。

所有版本的OpenGL规范文档都被公开的寄存在Khronos那里。有兴趣的读者可以找到OpenGL3.3(我们将要使用的版本)的[规范文档](#)。如果你想深入到OpenGL的细节(只关心函数功能的描述而不是函数的实现), 这是个很好的选择。如果你想知道每个函数具体的运作方式, 这个规范也是一个很棒的参考。

核心模式与立即渲染模式

早期的OpenGL使用[立即渲染模式](#)(Immediate mode, 也就是[固定渲染管线](#)), 这个模式下绘制图形很方便。OpenGL的大多数功能都被库隐藏起来, 开发者很少有控制OpenGL如何进行计算的自由。而开发者迫切希望能有更多的灵活性。随着时间推移, 规范越来越灵活, 开发者对绘图细节有了更多的掌控。立即渲染模式确实容易使用和理解, 但是效率太低。因此从OpenGL3.2开始, 规范文档开始废弃立即渲染模式, 并鼓励开发者在OpenGL的[核心模式](#)(Core-profile)下进行开发, 这个分支的规范完全移除了旧的特性。

当使用OpenGL的核心模式时, OpenGL迫使我们使用现代的函数。当我们试图使用一个已废弃的函数时, OpenGL会抛出一个错误并终止绘图。现代函数的优势是更高的灵活性和效率, 然而也更难于学习。立即渲染模式从OpenGL实际运作中抽象掉了很多细节, 因此它在易于学习的同时, 也很难让人去把握OpenGL具体是如何运作的。现代函数要求使用者真正理解OpenGL和图形编程, 它有一些难度, 然而提供了更多的灵活性, 更高的效率, 更重要的是可以更深入的理解图形编程。

这也是为什么我们的教程面向OpenGL3.3的核心模式。虽然上手更困难，但这份努力是值得的。

现今，更高版本的OpenGL已经发布（写作时最新版本为4.5），你可能会问：既然OpenGL 4.5都出来了，为什么我们还要学习OpenGL 3.3？答案很简单，所有OpenGL的更高的版本都是在3.3的基础上，引入了额外的功能，并没有改动核心架构。新版本只是引入了一些更有效率或更有用的方式去完成同样的功能。因此，所有的概念和技术在现代OpenGL版本里都保持一致。当你的经验足够，你可以轻松使用来自更高版本OpenGL的新特性。

当使用新版本的OpenGL特性时，只有新一代的显卡能够支持你的应用程序。这也是为什么大多数开发者基于较低版本的OpenGL编写程序，并只提供选项启用新版本的特性。

在有些教程里你会看见更现代的特性，它们同样会以这种红色注释方式标明。

扩展

OpenGL的一大特性就是对扩展(Extension)的支持，当一个显卡公司提出一个新特性或者渲染上的大优化，通常会以扩展的方式在驱动中实现。如果一个程序在支持这个扩展的显卡上运行，开发者可以使用这个扩展提供的一些更先进更有效的图形功能。通过这种方式，开发者不必等待一个新的OpenGL规范面世，就可以使用这些新的渲染特性了，只需要简单地检查一下显卡是否支持此扩展。通常，当一个扩展非常流行或者非常有用的时候，它将最终成为未来的OpenGL规范的一部分。

使用扩展的代码大多看上去如下：

```
if(GL_ARB_extension_name)
{
    // 使用硬件支持的全新的现代特性
}
else
{
    // 不支持此扩展：用旧的方式去做
}
```

使用OpenGL3.3时，我们很少需要使用扩展来完成大多数功能，当需要的时候，本教程将提供适当的指示。

状态机

OpenGL自身是一个巨大的状态机(State Machine)：一系列的变量描述OpenGL此刻应当如何运行。OpenGL的状态通常被称为OpenGL上下文(Context)。我们通常使用如下途径去更改OpenGL状态：设置选项，操作缓冲。最后，我们使用当前OpenGL上下文来渲染。

假设当我们想告诉OpenGL去画线段而不是三角形的时候，我们通过改变一些上下文变量来改变OpenGL状态，从而告诉OpenGL如何去绘图。一旦我们改变了OpenGL的状态为绘制线段，下一个绘制命令就会画出线段而不是三角形。

当使用OpenGL的时候，我们会遇到一些状态设置函数(State-changing Function)，这类函数将会改变上下文。以及状态使用函数(State-using Function)，这类函数会根据当前OpenGL的状态执行一些操作。只要你记住OpenGL本质上是个大状态机，就能更容易理解它的大部分特性。

对象

OpenGL库是用C语言写的，同时也支持多种语言的派生，但其内核仍是一个C库。由于C的一些语言结构不易被翻译到其它的高级语言，因此OpenGL开发的时候引入了一些抽象层。“对象(Object)”就是其中一个。

在OpenGL中一个对象是指一些选项的集合，它代表OpenGL状态的一个子集。比如，我们可以用一个对象来代表绘图窗口的设置，之后我们就可以设置它的大小、支持的颜色位数等等。可以把对象看做一个C风格的结构体(Struct)：

```
struct object_name {
    float option1;
    int option2;
    char[] name;
};
```

译注

在更新前的教程中一直使用的都是OpenGL的基本类型，但由于作者觉得在本教程系列中并没有一个必须使用它们的原因，所有的类型都改为了自带类型。但是请仍然记住，使用OpenGL的类型的好处是保证了在各平台中每一种类型的大小都是统一的。你也可以使用其它的定宽类型(Fixed-width Type)来实现这一点。

当我们使用一个对象时，通常看起来像如下一样（把OpenGL上下文看作一个大的结构体）：

```
// OpenGL的状态
struct OpenGL_Context {
    ...
    object* object_Window_Target;
    ...
};

// 创建对象
unsigned int objectId = 0;
glGenObject(1, &objectId);
// 绑定对象至上下文
 glBindObject(GL_WINDOW_TARGET, objectId);
// 设置当前绑定到 GL_WINDOW_TARGET 的对象的一些选项
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH, 800);
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_HEIGHT, 600);
// 将上下文对象设回默认
 glBindObject(GL_WINDOW_TARGET, 0);
```

这一小段代码展现了你以后使用OpenGL时常见的工作流。我们首先创建一个对象，然后用一个id保存它的引用（实际数据被储存在后台）。然后我们将对象绑定至上下文的目标位置（例子中窗口对象目标的位置被定义成`GL_WINDOW_TARGET`）。接下来我们设置窗口的选项。最后我们将目标位置的对象id设回0，解绑这个对象。设置的选项将被保存在`objectId`所引用的对象中，一旦我们重新绑定这个对象到`GL_WINDOW_TARGET`位置，这些选项就会重新生效。

目前提供的示例代码只是OpenGL如何操作的一个大致描述，通过阅读以后的教程你会遇到很多实际的例子。

使用对象的一个好处是在程序中，我们不止可以定义一个对象，并设置它们的选项，每个对象都可以是不同的设置。在我们执行一个使用OpenGL状态的操作的时候，只需要绑定含有需要的设置的对象即可。比如说我们有一些作为3D模型数据（一栋房子或一个人物）的容器对象，在我们想绘制其中任何一个模型的时候，只需绑定一个包含对应模型数据的对象就可以了（当然，我们需要先创建并设置对象的选项）。拥有数个这样的对象允许我们指定多个模型，在想画其中任何一个的时候，直接将对应的对象绑定上去，便不需要再重复设置选项了。

让我们开始吧

你现在已经知道一些OpenGL的相关知识了，OpenGL规范和库，OpenGL幕后大致的运作流程，以及OpenGL使用的一些传统技巧。不要担心你还没有完全消化它们，后面的教程我们会仔细地讲解每一个步骤，你会通过足够的例子来真正掌握OpenGL。如果你已经做好了开始下一步的准备，我们可以在[这里](#)开始创建OpenGL上下文以及我们的第一个窗口了。

附加资源

- opengl.org : OpenGL官方网站。
- [OpenGL registry](http://OpenGL Registry): 包含OpenGL各版本的规范和扩展。

2.2.2 创建窗口

原文 [Creating a window](#)

作者 JoeyDeVries

翻译 ggy_1992, Krasjet

校对 暂未校对

译注

注意，由于作者对教程做出了更新，之前本节使用的是GLEW库，但现在改为了使用GLAD库，关于GLEW配置的部分现在已经修改，但我仍决定将这部分教程保留起来，放到一个历史存档中，如果有需要的话可以到[这里](#)来查看。

在我们画出出色的效果之前，首先要做的就是创建一个OpenGL上下文(Context)和一个用于显示的窗口。然而，这些操作在每个系统上都是不一样的，OpenGL有意将这些操作抽象(Abstract)出去。这意味着我们不得不自己处理创建窗口，定义OpenGL上下文以及处理用户输入。

幸运的是，有一些库已经提供了我们所需的功能，其中一部分是特别针对OpenGL的。这些库节省了我们书写操作系统相关代码的时间，提供给我们一个窗口和一个OpenGL上下文用来渲染。最流行的几个库有GLUT，SDL，SFML和GLFW。在教程里我们将使用GLFW。你可以随意选用其他类似的库，大多数库的配置方法和GLFW差不多。

GLFW

GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文、定义窗口参数以及处理用户输入，对我们来说这就够了。

本节和下一节的目标是把GLFW环境配好能且能够跑起来，并保证它正确创建了OpenGL上下文并显示出一个简单的窗口来让我们随意使用。这篇教程会一步步教你如何获取、编译、链接GLFW库。我们使用的是Microsoft Visual Studio 2019 IDE（操作过程在更新的Visual Studio都是相同的）。如果你用的不是Visual Studio（或者用的是它的旧版本）请不要担心，大多数IDE上的操作都是类似的。

构建GLFW

GLFW可以从它官方网站的[下载页](#)上获取。GLFW已提供为Visual Studio（2012到2019都有）预编译好的二进制版本和相应的头文件，但是为了完整性我们将从编译源代码开始。所以我们要下载源代码包。

本教程中，我们将采用64位构建所有的库。因此如果您使用的是预编译的二进制文件，请确保你下载的是64位的二进制文件。

下载源码包之后，将其解压并打开。我们只需要里面的这些内容：

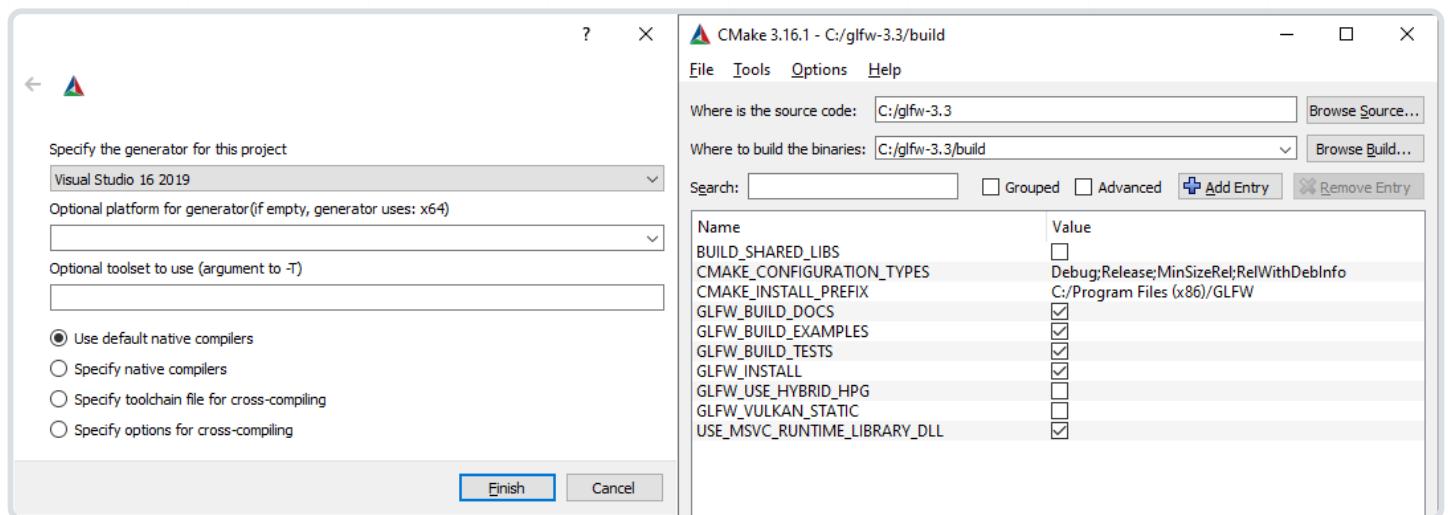
- 编译生成的库
- `include`文件夹

从源代码编译库可以保证生成的库完全适合你的操作系统和CPU的，而预编译的二进制文件则并非总是提供（有时候，即便提供了预编译的二进制文件，也可能不适用于您的系统）。开放源代码所产生问题在于：并不是每个人都用相同的IDE或者构建系统来搞开发，因而提供的项目/解决方案文件可能和一些人的IDE不兼容。所以人们必须使用给定的.c/.cpp和.h/.hpp文件来自己建立项目/解决方案，这是一项很枯燥的工作。但因此也诞生了一个叫做CMake的工具。

CMake

CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件。这允许我们从GLFW源码创建一个Visual Studio 2019工程文件，之后进行编译。首先，我们需要从[这里](#)下载安装CMake。

当CMake安装成功后，你可以选择从命令行或者GUI启动CMake，由于我们不想让事情变得太过复杂，我们选择用GUI。CMake需要一个源代码目录和一个存放编译结果的目标文件目录。源代码目录我们选择GLFW的源代码的根目录，然后我们新建一个 *build* 文件夹，选中作为目标目录。



在设置完源代码目录和目标目录之后，点击Configure(设置)按钮，让CMake读取设置和源代码。我们接下来需要选择工程的生成器，由于我们使用的是Visual Studio 2019，我们选择 Visual Studio 16 选项（因为Visual Studio 2019的内部版本号是16）。CMake会显示可选的编译选项用来配置最终生成的库。这里我们使用默认设置，并再次点击Configure(设置)按钮保存设置。保存之后，点击Generate(生成)按钮，生成的工程文件会在你的**build**文件夹中。

编译

在**build**文件夹里可以找到**GLFW.sln**文件，用Visual Studio 2019打开。因为CMake已经配置好了项目，并按照默认配置将其编译为64位的库，所以我们直接点击Build Solution(生成解决方案)按钮，然后在**build/src/Debug**文件夹内就会出现我们编译出的库文件**glfw3.lib**。

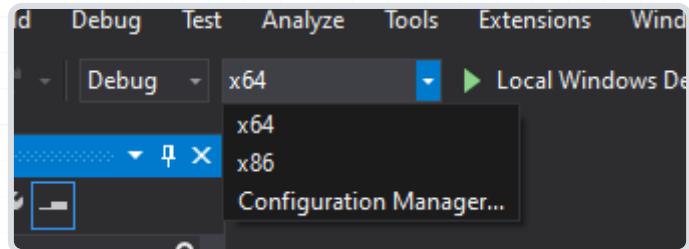
库生成完毕之后，我们需要让IDE知道库和头文件的位置。有两种方法：

1. 找到IDE或者编译器的**/lib**和**/include**文件夹，添加GLFW的**include**文件夹里的文件到IDE的**/include**文件夹里去。用类似的方法，将**glfw3.lib**添加到**/lib**文件夹里去。虽然这样能工作，但这不是推荐的方式，因为这样会让你很难去管理库和**include**文件，而且重新安装IDE或编译器可能会导致这些文件丢失。
2. 推荐的方式是建立一个新的目录包含所有的第三方库文件和头文件，并且在你的IDE或编译器中指定这些文件夹。我个人会使用一个单独的文件夹，里面包含**Libs**和**Include**文件夹，在这里存放OpenGL工程用到的所有第三方库和头文件。这样我的所有第三方库都在同一个位置（并且可以共享至多台电脑）。然而这要求你每次新建一个工程时都需要告诉IDE/编译器在哪能找到这些目录。

完成上面步骤后，我们就可以使用GLFW创建我们的第一个OpenGL工程了！

我们的第一个工程

首先，打开Visual Studio，创建一个新的项目。如果VS提供了多个选项，选择Visual C++，然后选择**Empty Project(空项目)**（别忘了给你的项目起一个合适的名字）。由于我们将在64位模式中执行所有操作，而新项目默认是32位的，因此我们需要将Debug旁边顶部的下拉列表从x86更改为x64：

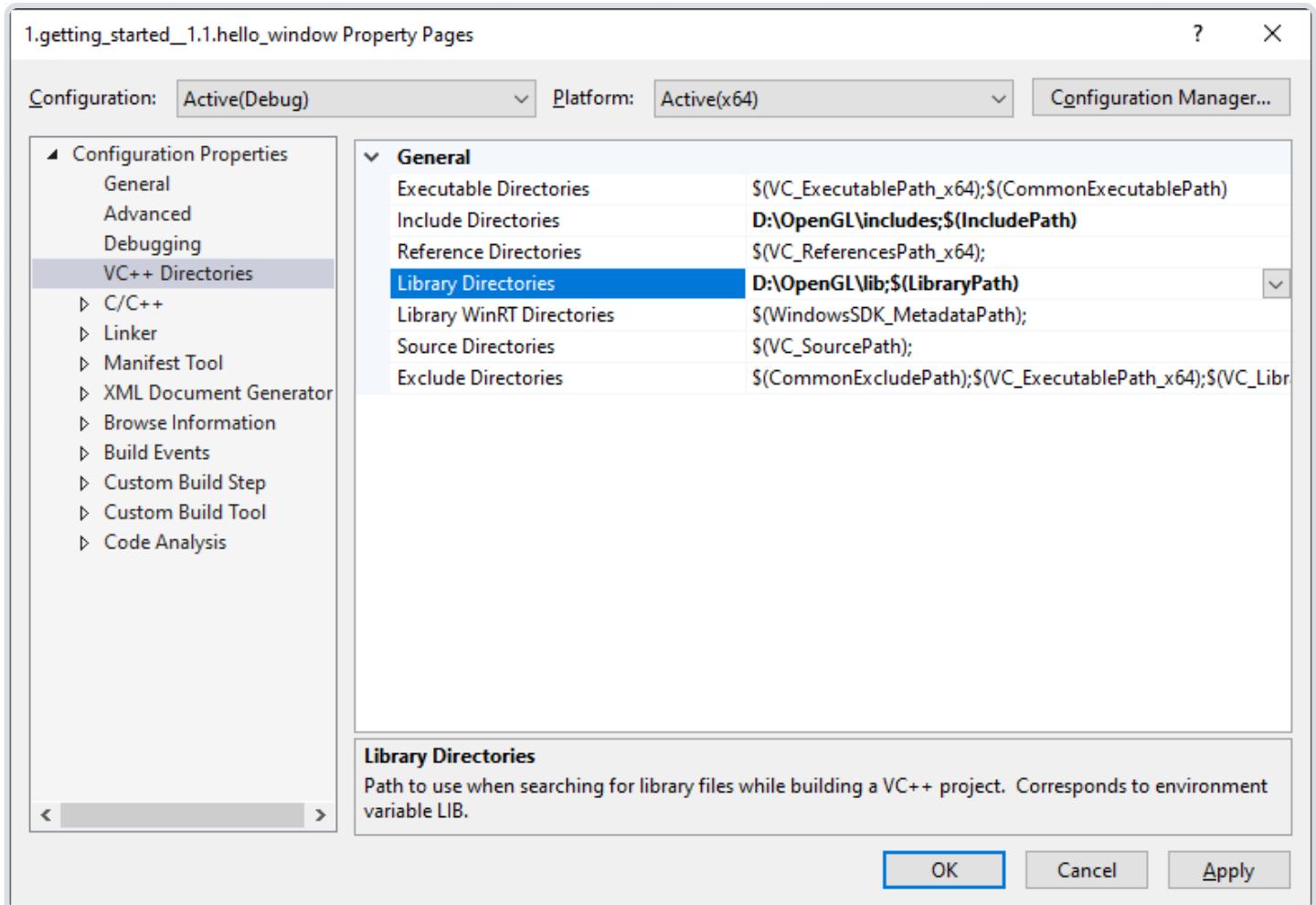


现在我们终于有一个空的工作空间了，开始创建我们第一个OpenGL程序吧！

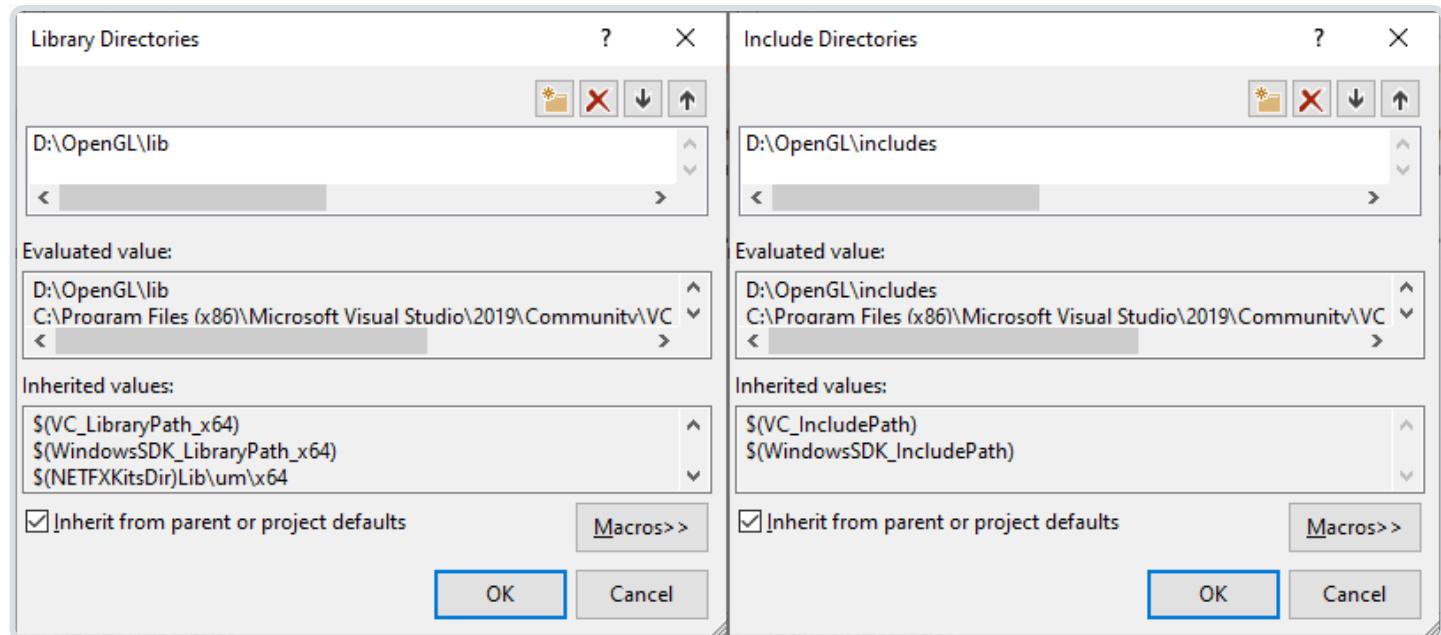
链接

为了使我们的程序使用GLFW，我们需要把GLFW库**链接**(Link)进工程。这可以通过在链接器的设置里指定我们要使用`glfw3.lib`来完成，但是由于我们将第三方库放在另外的目录中，我们的工程还不知道在哪寻找这个文件。于是我们首先需要将我们放第三方库的目录添加进设置。

要添加这些目录（需要VS搜索库和include文件的地方），我们首先进入Project Properties(工程属性，在解决方案窗口里右键项目)，然后选择**VC++ Directories(VC++ 目录)**选项卡（如下图）。在下面的两栏添加目录：

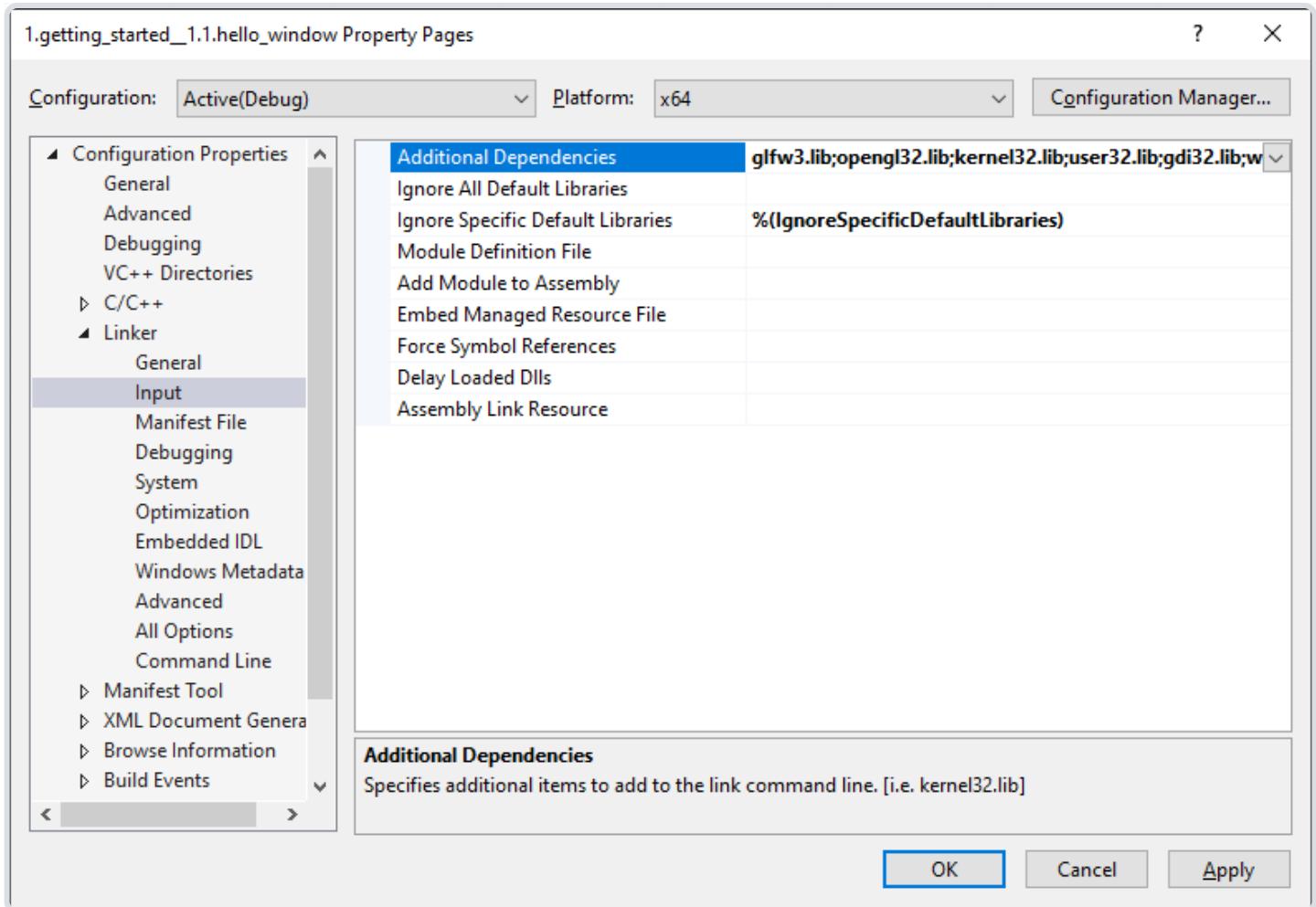


这里你可以把自己的目录加进去，让工程知道到哪去搜索。你需要手动把目录加在后面，也可以点击需要的位置字符串，选择选项，之后会出现类似下面这幅图的界面，图是选择Include Directories(包含目录)时的界面：



这里可以添加任意多个目录，IDE会从这些目录里寻找头文件。所以只要你将GLFW的**Include**文件夹加进路径中，你就可以使用
`<GLFW/...>` 来引用头文件。库文件夹也是一样的。

现在VS可以找到所需的所有文件了。最后需要在**Linker(链接器)**选项卡里的**Input(输入)**选项卡里添加glfw3.lib这个文件：



要链接一个库我们必须告诉链接器它的文件名。库名字是`glfw3.lib`，我们把它加到Additional Dependencies(附加依赖项)字段中(手动或者使用选项都可以)。这样GLFW在编译的时候就会被链接进来了。除了GLFW之外，你还需要添加一个链接条目链接到OpenGL的库，但是这个库可能因为系统的不同而有一些差别。

Windows上的OpenGL库

如果你是Windows平台，`opengl32.lib`已经包含在Microsoft SDK里了，它在Visual Studio安装的时候就默认安装了。由于这篇教程用的是VS编译器，并且是在Windows操作系统上，我们只需将`opengl32.lib`添加进连接器设置里就行了。值得注意的是，OpenGL库64位版本的文件名仍然是`opengl32.lib`（和32位版本一样），虽然很奇怪但确实如此。

Linux上的OpenGL库

在Linux下你需要链接`libGL.so`库文件，这需要添加`-lGL`到你的链接器设置中。如果找不到这个库你可能需要安装Mesa，Nvidia或AMD的开发包，这部分因平台而异（而且我也不熟悉Linux）就不仔细讲解了。

接下来，如果你已经添加GLFW和OpenGL库到连接器设置中，你可以用如下方式添加GLFW头文件：

```
#include <GLFW/glfw3.h>
```

对于用GCC编译的Linux用户建议使用这个命令行选项 `-lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi -ldl`。没有正确链接相应的库会产生 *undefined reference* (未定义的引用) 这个错误。

GLFW的安装与配置就到此为止。

GLAD

到这里还没有结束，我们仍然还有一件事要做。因为OpenGL只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于OpenGL驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。所以任务就落在了开发者身上，开发者需要在运行时获取函数地址并将其保存在一个函数指针中供以后使用。取得地址的方法因平台而异，在Windows上会是类似这样：

```
// 定义函数原型
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);
// 找到正确的函数并赋值给函数指针
GL_GENBUFFERS glGenBuffers = (GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");
// 现在函数可以被正常调用了
GLuint buffer;
glGenBuffers(1, &buffer);
```

你可以看到代码非常复杂，而且很繁琐，我们需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中 GLAD是目前最新，也是最流行的库。

配置GLAD

GLAD是一个[开源](#)的库，它能解决我们上面提到的那个繁琐的问题。GLAD的配置与大多数的开源库有些许的不同，GLAD使用了一个[在线服务](#)。在这里我们能够告诉GLAD需要定义的OpenGL版本，并且根据这个版本加载所有相关的OpenGL函数。

打开GLAD的[在线服务](#)，将语言(Language)设置为C/C++，在API选项中，选择3.3以上的OpenGL(gl)版本（我们的教程中将使用3.3版本，但更新的版本也能用）。之后将模式(Profile)设置为Core，并且保证选中了生成加载器(Generate a loader)选项。现在可以先（暂时）忽略扩展(Extensions)中的内容。都选择完之后，点击生成(Generate)按钮来生成库文件。

GLAD现在应该提供给你了一个zip压缩文件，包含两个头文件目录，和一个glad.c文件。将两个头文件目录（glad和KHR）复制到你的Include文件夹中（或者增加一个额外的项目指向这些目录），并添加glad.c文件到你的工程中。

经过前面的这些步骤之后，你就应该可以将以下的指令加到你的文件顶部了：

```
#include <glad/glad.h>
```

点击编译按钮应该不会给你提示任何的错误，到这里我们就已经准备好继续学习[下一节](#)去真正使用GLFW和GLAD来设置OpenGL上下文并创建一个窗口了。记得确保你的头文件和库文件的目录设置正确，以及链接器里引用的库文件名正确。如果仍然遇到错误，可以先看一下评论有没有人遇到类似的问题，请参考额外资源中的例子或者在下面的评论区提问。

附加资源

- [GLFW: Window Guide](#)：GLFW官方的配置GLFW窗口的指南。
- [Building applications](#)：提供了很多编译或链接相关的信息和一大列错误及对应的解决方案。
- [GLFW with Code::Blocks](#)：使用Code::Blocks IDE编译GLFW。

- [Running CMake](#): 简要的介绍如何在Windows和Linux上使用CMake。
- [Writing a build system under Linux](#) : Wouter Verholst写的一个autotools的教程，讲的是如何在Linux上编写构建系统，尤其是针对这些教程。
- [Polytonic/Glitter](#): 一个简单的样板项目，它已经提前配置了所有相关的库；如果你想要很方便地搞到一个LearnOpenGL教程的范例工程，这也是很不错的。

2.2.3 你好，窗口

原文 [Hello Window](#)

作者 JoeyDeVries

翻译 Geequlim, Krasjet

校对 暂未校对

让我们试试能不能让GLFW正常工作。首先，新建一个 `.cpp` 文件，然后把下面的代码粘贴到该文件的最前面。

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

请确认是在包含GLFW的头文件之前包含了GLAD的头文件。GLAD的头文件包含了正确的OpenGL头文件（例如 `GL/gl.h`），所以需要在其它依赖于OpenGL的头文件之前包含GLAD。

接下来我们创建 `main` 函数，在这个函数中我们将会实例化GLFW窗口：

```
int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    return 0;
}
```

首先，我们在 `main` 函数中调用 `glfwInit` 函数来初始化GLFW，然后我们可以使用 `glfwWindowHint` 函数来配置GLFW。

`glfwWindowHint` 函数的第一个参数代表选项的名称，我们可以从很多以 `GLFW_` 开头的枚举值中选择；第二个参数接受一个整型，用来设置这个选项的值。该函数的所有选项以及对应的值都可以在 [GLFW's window handling](#) 这篇文档中找到。如果你现在编译你的 `cpp` 文件会得到大量的 *undefined reference* (未定义的引用) 错误，也就是说你并未顺利地链接GLFW库。

由于本站的教程都是基于OpenGL 3.3版本展开讨论的，所以我们需要告诉GLFW我们要使用的OpenGL版本是3.3，这样GLFW会在创建OpenGL上下文时做出适当的调整。这也可以确保用户在没有适当的OpenGL版本支持的情况下无法运行。我们将主版本号(Major)和次版本号(Minor)都设为3。我们同样明确告诉GLFW我们使用的是核心模式(Core-profile)。明确告诉GLFW我们需要使用核心模式意味着我们只能使用OpenGL功能的一个子集（没有我们已不再需要的向后兼容特性）。如果使用的是Mac OS X系统，你还需要加下面这行代码到你的初始化代码中这些配置才能起作用（将上面的代码解除注释）：

```
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

请确认您的系统支持OpenGL3.3或更高版本，否则此应用有可能会崩溃或者出现不可预知的错误。如果想要查看OpenGL版本的话，在Linux上运行 `glxinfo`，或者在Windows上使用其它的工具（例如 [OpenGL Extension Viewer](#)）。如果你的OpenGL版本低于3.3，检查一下显卡是否支持OpenGL 3.3+（不支持的话你的显卡真的太老了），并更新你的驱动程序，有必要的话请更新显卡。

接下来我们创建一个窗口对象，这个窗口对象存放了所有和窗口相关的数据，而且会被GLFW的其他函数频繁地用到。

```
GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

`glfwCreateWindow`函数需要窗口的宽和高作为它的前两个参数。第三个参数表示这个窗口的名称（标题），这里我们使用`"LearnOpenGL"`，当然你也可以使用你喜欢的名称。最后两个参数我们暂时忽略。这个函数将会返回一个`GLFWwindow`对象，我们会在其它的GLFW操作中使用到。创建完窗口我们就可以通知GLFW将我们窗口的上下文设置为当前线程的主上下文了。

GLAD

在之前的教程中已经提到过，GLAD是用来管理OpenGL的函数指针的，所以在调用任何OpenGL的函数之前我们需要初始化GLAD。

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

我们给GLAD传入了用来加载系统相关的OpenGL函数指针地址的函数。GLFW给我们的`glfwGetProcAddress`，它根据我们编译的系统定义了正确的函数。

视口

在我们开始渲染之前还有一件重要的事情要做，我们必须告诉OpenGL渲染窗口的尺寸大小，即视口(Viewport)，这样OpenGL才只能知道怎样根据窗口大小显示数据和坐标。我们可以通过调用`glViewport`函数来设置窗口的维度(Dimension)：

```
glViewport(0, 0, 800, 600);
```

`glViewport`函数前两个参数控制窗口左下角的位置。第三个和第四个参数控制渲染窗口的宽度和高度（像素）。

我们实际上也可以将视口的维度设置为比GLFW的维度小，这样子之后所有的OpenGL渲染将会在一个更小的窗口中显示，这样子的话我们也可以将一些其它元素显示在OpenGL视口之外。

OpenGL幕后使用`glViewport`中定义的位置和宽高进行2D坐标的转换，将OpenGL中的位置坐标转换为你的屏幕坐标。例如，OpenGL中的坐标(-0.5, 0.5)有可能（最终）被映射为屏幕中的坐标(200,450)。注意，处理过的OpenGL坐标范围只为-1到1，因此我们事实上将(-1到1)范围内的坐标映射到(0, 800)和(0, 600)。

然而，当用户改变窗口的大小的时候，视口也应该被调整。我们可以对窗口注册一个回调函数(Callback Function)，它会在每次窗口大小被调整的时候被调用。这个回调函数的原型如下：

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

这个帧缓冲大小函数需要一个`GLFWwindow`作为它的第一个参数，以及两个整数表示窗口的新维度。每当窗口改变大小，GLFW会调用这个函数并填充相应的参数供你处理。

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

我们还需要注册这个函数，告诉GLFW我们希望每当窗口调整大小的时候调用这个函数：

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

当窗口被第一次显示的时候`framebuffer_size_callback`也会被调用。对于视网膜(Retina)显示屏，`width`和`height`都会明显比原输入值更高一点。

我们还可以将我们的函数注册到其它很多的回调函数中。比如说，我们可以创建一个回调函数来处理手柄输入变化，处理错误消息等。我们会在创建窗口之后，渲染循环初始化之前注册这些回调函数。

2.2.4 准备好你的引擎

我们可不希望只绘制一个图像之后我们的应用程序就立即退出并关闭窗口。我们希望程序在我们主动关闭它之前不断绘制图像并能够接受用户输入。因此，我们需要在程序中添加一个while循环，我们可以把它称之为**渲染循环**(Render Loop)，它能在我们让GLFW退出前一直保持运行。下面几行的代码就实现了一个简单的渲染循环：

```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

- `glfwWindowShouldClose`函数在我们每次循环的开始前检查一次GLFW是否被要求退出，如果是的话该函数返回 `true` 然后渲染循环便结束了，之后为我们就可以关闭应用程序了。
- `glfwPollEvents`函数检查有没有触发什么事件（比如键盘输入、鼠标移动等）、更新窗口状态，并调用对应的回调函数（可以通过回调方法手动设置）。
- `glfwSwapBuffers`函数会交换颜色缓冲（它是一个储存着GLFW窗口每一个像素颜色值的大缓冲），它在这一迭代中被用来绘制，并且将会作为输出显示在屏幕上。

双缓冲(Double Buffer)

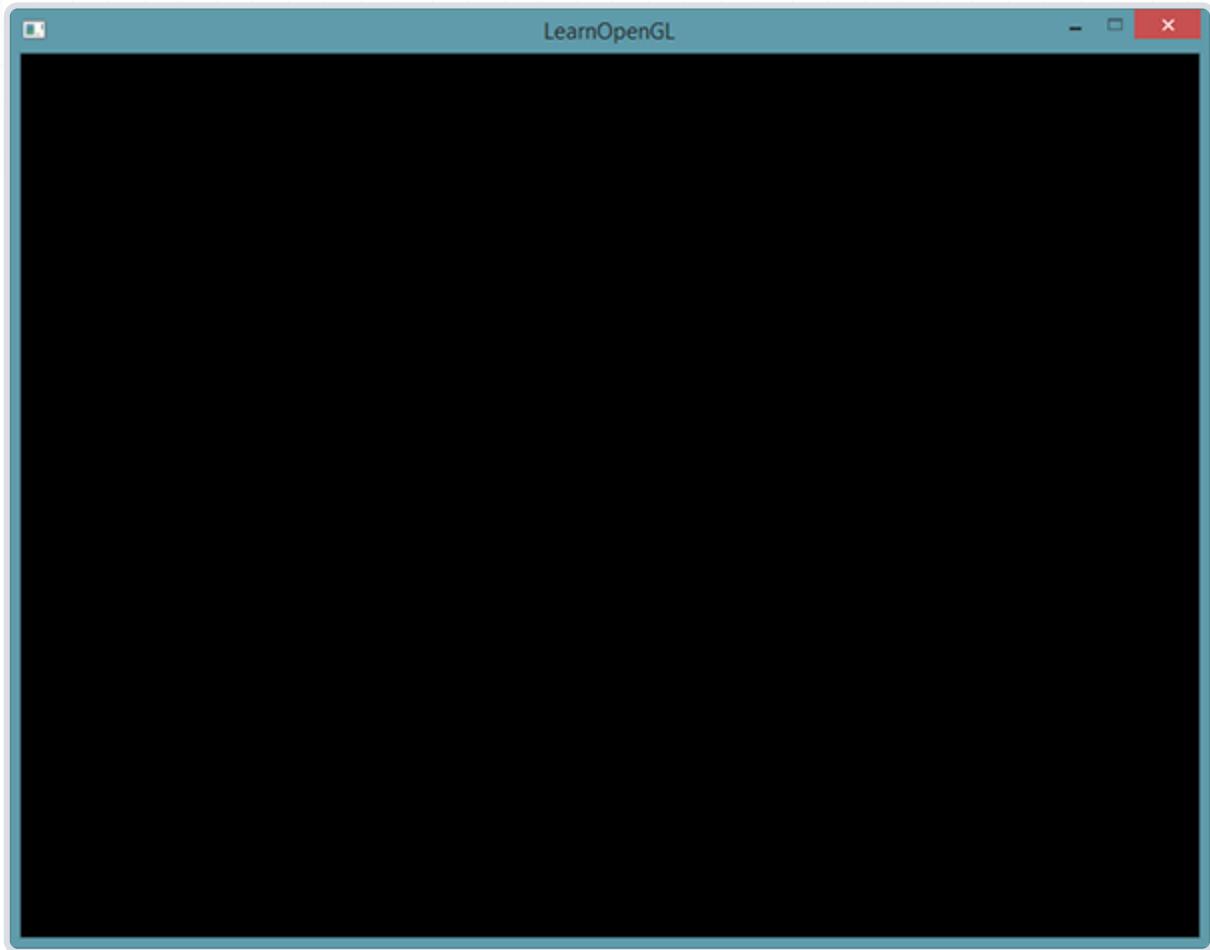
应用程序使用单缓冲绘图时可能会存在图像闪烁的问题。这是因为生成的图像不是一下子被绘制出来的，而是按照从左到右，由上而下逐像素地绘制而成的。最终图像不是在瞬间显示给用户，而是通过一步一步生成的，这会导致渲染的结果很不真实。为了规避这些问题，我们应用双缓冲渲染窗口应用程序。前缓冲保存着最终输出的图像，它会在屏幕上显示；而所有的渲染指令都会在后缓冲上绘制。当所有的渲染指令执行完毕后，我们交换(Swap)前缓冲和后缓冲，这样图像就立即呈显出来，之前提到的不真实感就消除了。

最后一件事

当渲染循环结束后我们需要正确释放/删除之前的分配的所有资源。我们可以在`main`函数的最后调用`glfwTerminate`函数来完成。

```
glfwTerminate();
return 0;
```

这样便能清理所有的资源并正确地退出应用程序。现在你可以尝试编译并运行你的应用程序了，如果没做错的话，你将会看到如下的输出：



如果你看见了一个非常无聊的黑色窗口，那么就对了！如果你没得到正确的结果，或者你不知道怎么把所有东西放到一起，请到[这里](#)参考源代码。

如果程序编译有问题，请先检查连接器选项是否正确，IDE中是否导入了正确的目录（前面教程解释过）。并且请确认你的代码是否正确，直接对照上面提供的源代码就行了。如果还是有问题，欢迎评论，我或者其他人可能会帮助你的。

输入

我们同样也希望能够在GLFW中实现一些输入控制，这可以通过使用GLFW的几个输入函数来完成。我们将会使用GLFW的 `glfwGetKey` 函数，它需要一个窗口以及一个按键作为输入。这个函数将会返回这个按键是否正在被按下。我们将创建一个 `processInput` 函数来让所有的输入代码保持整洁。

```
void processInput(GLFWwindow *window)
{
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}
```

这里我们检查用户是否按下了返回键(Esc)（如果没有按下，`glfwGetKey` 将会返回 `GLFW_RELEASE`）。如果用户的确按下了返回键，我们将通过 `glfwSetWindowShouldClose` 使用把 `WindowShouldClose` 属性设置为 `true` 的方法关闭GLFW。下一次while循环的条件检测将会失败，程序将会关闭。

我们接下来在渲染循环的每一个迭代中调用 `processInput`：

```
while (!glfwWindowShouldClose(window))
{
    processInput(window);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

这就给我们一个非常简单的方式来检测特定的键是否被按下，并在每一帧做出处理。

渲染

我们要把所有的渲染(Rendering)操作放到渲染循环中，因为我们想让这些渲染指令在每次渲染循环迭代的时候都能被执行。代码将会是这样的：

```
// 渲染循环
while(!glfwWindowShouldClose(window))
{
    // 输入
    processInput(window);

    // 渲染指令
    ...

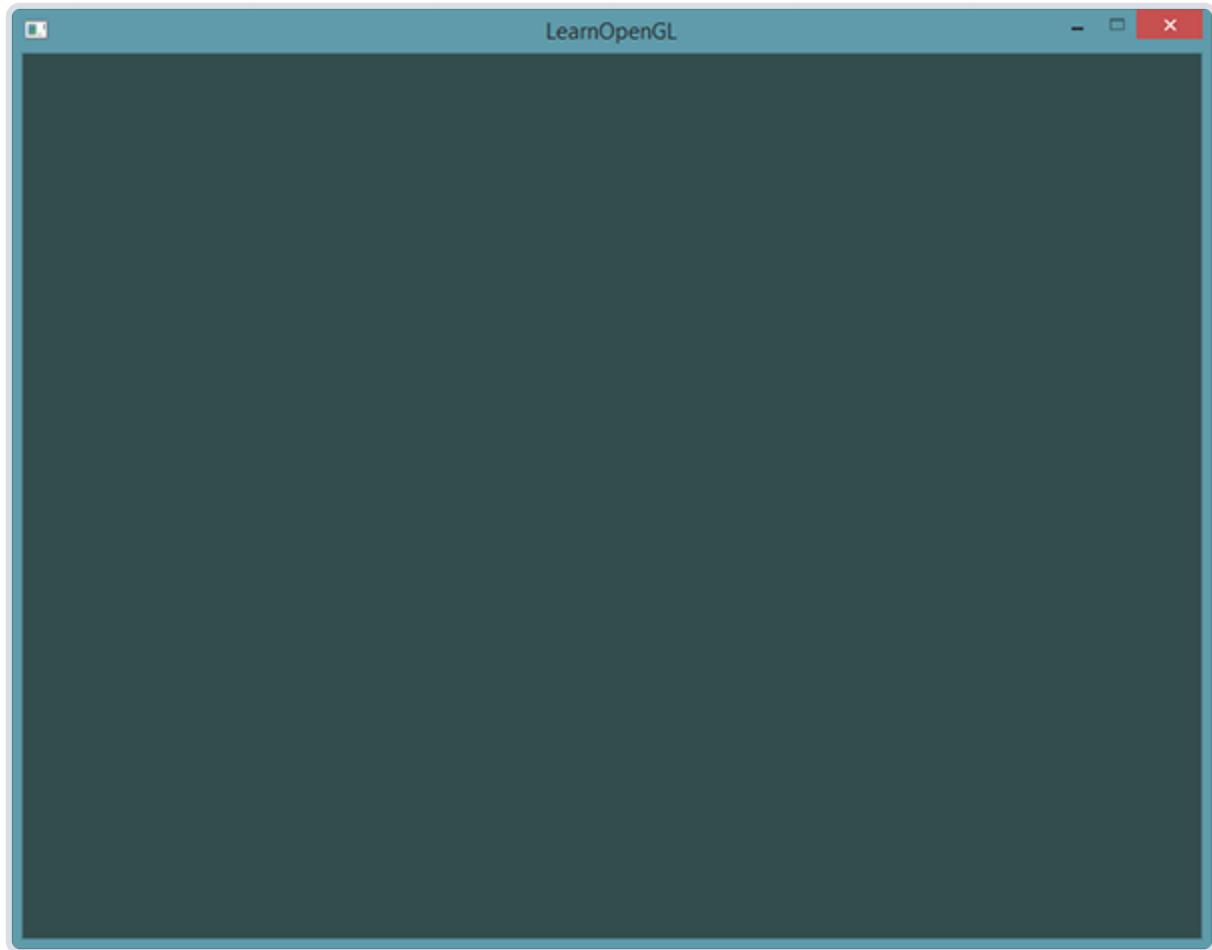
    // 检查并调用事件，交换缓冲
    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

为了测试一切都正常工作，我们使用一个自定义的颜色清空屏幕。在每个新的渲染迭代开始的时候我们总是希望清屏，否则我们仍能看见上一次迭代的渲染结果（这可能是你想要的效果，但通常这不是）。我们可以通过调用`glClear`函数来清空屏幕的颜色缓冲，它接受一个缓冲位(Buffer Bit)来指定要清空的缓冲，可能的缓冲位有`GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`和`GL_STENCIL_BUFFER_BIT`。由于现在我们只关心颜色值，所以我们只清空颜色缓冲。

```
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

注意，除了`glClear`之外，我们还调用了`glClearColor`来设置清空屏幕所用的颜色。当调用`glClear`函数，清除颜色缓冲之后，整个颜色缓冲都会被填充为`glClearColor`里所设置的颜色。在这里，我们将屏幕设置为了类似黑板的深蓝绿色。

你应该能够回忆起来我们在 *OpenGL* 这节教程的内容，`glClearColor`函数是一个状态设置函数，而`glClear`函数则是一个状态使用的函数，它使用了当前的状态来获取应该清除为的颜色。



这个程序的完整源代码可以在[这里](#)找到。

好了，现在我们已经做好开始在渲染循环中添加许多渲染调用的准备了，但这是[下一节](#)教程了，这一节的内容已经太多了。

2.2.5 你好，三角形

原文 [Hello Triangle](#)

作者 JoeyDeVries

翻译 [Django](#), Krasjet, Geequlim

校对 暂未校对

译注

在学习此节之前，建议将这三个单词先记下来：

- 顶点数组对象：Vertex Array Object，VAO
- 顶点缓冲对象：Vertex Buffer Object，VBO
- 索引缓冲对象：Element Buffer Object，EBO或Index Buffer Object，IBO

当指代这三个东西的时候，可能使用的是全称，也可能用的是英文缩写，翻译的时候和原文保持的一致。由于没有英文那样的分词间隔，中文全称的部分可能不太容易注意。但请记住，缩写和中文全称指代的是一个东西。

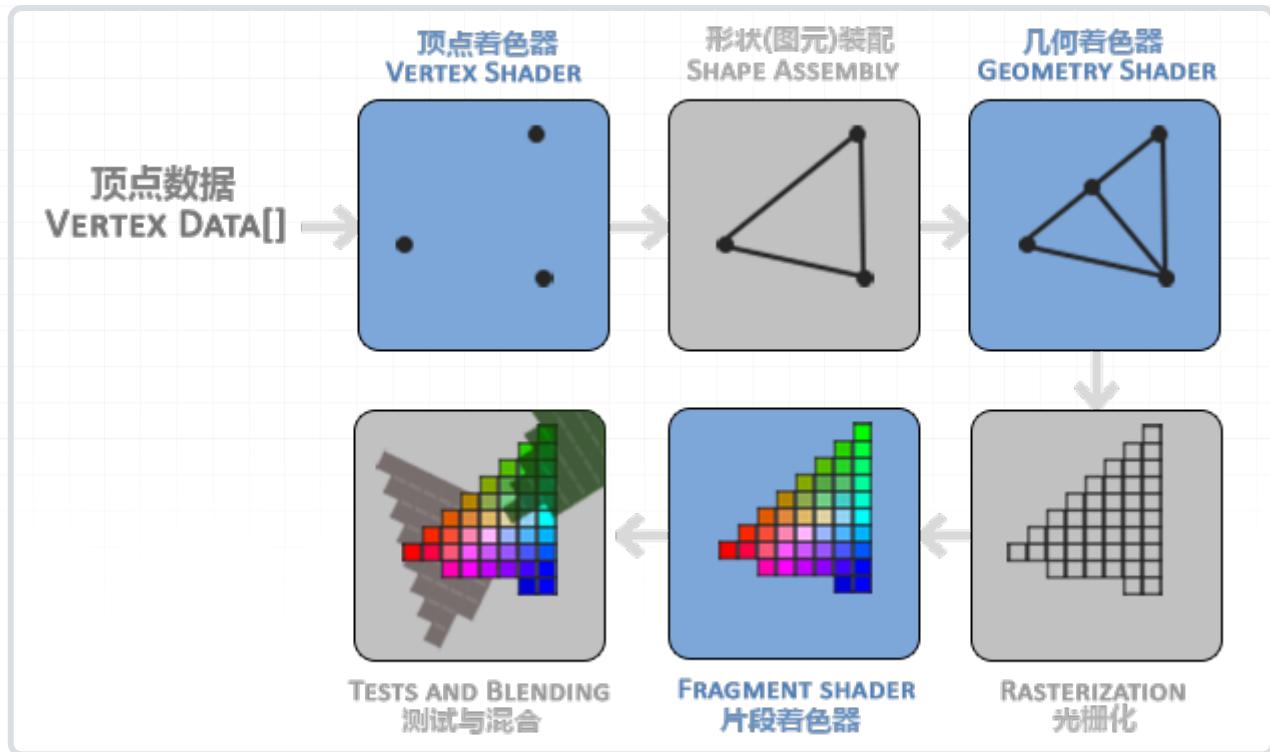
在OpenGL中，任何事物都在3D空间中，而屏幕和窗口却是2D像素数组，这导致OpenGL的大部分工作都是关于把3D坐标转变为适应你屏幕的2D像素。3D坐标转为2D坐标的处理过程是由OpenGL的**图形渲染管线**（Graphics Pipeline，大多译为管线，实际上指的是一堆原始图形数据途经一个输送管道，期间经过各种变化处理最终出现在屏幕的过程）管理的。图形渲染管线可以被划分为两个主要部分：第一部分把你的3D坐标转换为2D坐标，第二部分是把2D坐标转变为实际的有颜色的像素。这个教程里，我们会简单地讨论一下图形渲染管线，以及如何利用它创建一些漂亮的像素。

2D坐标和像素也是不同的，2D坐标精确表示一个点在2D空间中的位置，而2D像素是这个点的近似值，2D像素受到你的屏幕/窗口分辨率的限制。

图形渲染管线接受一组3D坐标，然后把它们转变为你屏幕上的有色2D像素输出。图形渲染管线可以被划分为几个阶段，每个阶段将会把前一个阶段的输出作为输入。所有这些阶段都是高度专门化的（它们都有一个特定的函数），并且很容易并行执行。正是由于它们具有并行执行的特性，当今大多数显卡都有成千上万的小处理核心，它们在GPU上为每一个（渲染管线）阶段运行各自的小程序，从而在图形渲染管线中快速处理你的数据。这些小程序叫做**着色器**(Shader)。

有些着色器允许开发者自己配置，这就允许我们用自己写的着色器来替换默认的。这样我们就可以更细致地控制图形渲染管线中的特定部分了，而且因为它们运行在GPU上，所以它们可以给我们节约宝贵的CPU时间。OpenGL着色器是用**OpenGL着色器语言**(OpenGL Shading Language, [GLSL](#))写成的，在下一节中我们再花更多时间研究它。

下面，你会看到一个图形渲染管线的每个阶段的抽象展示。要注意蓝色部分代表的是我们可以注入自定义的着色器的部分。



如你所见，图形渲染管线包含很多部分，每个部分都将在转换顶点数据到最终像素这一过程中处理各自特定的阶段。我们会概括性地解释一下渲染管线的每个部分，让你对图形渲染管线的工作方式有个大概了解。

首先，我们以数组的形式传递3个3D坐标作为图形渲染管线的输入，用来表示一个三角形，这个数组叫做顶点数据(Vertex Data)；顶点数据是一系列顶点的集合。一个**顶点**(Vertex)是一个3D坐标的数据的集合。而顶点数据是用**顶点属性**(Vertex Attribute)表示的，它可以包含任何我们想用的数据，但是简单起见，我们还是假定每个顶点只由一个3D位置(译注1)和一些颜色值组成的吧。

译注1

当我们谈论一个“位置”的时候，它代表在一个“空间”中所处地点的这个特殊属性；同时“空间”代表着任何一种坐标系，比如x、y、z三维坐标系，x、y二维坐标系，或者一条直线上的x和y的线性关系，只不过二维坐标系是一个扁扁的平面空间，而一条直线是一个很瘦的长长的空间。

为了让OpenGL知道我们的坐标和颜色值构成到底是什么，OpenGL需要你去指定这些数据所表示的渲染类型。我们是希望把这些数据渲染成一系列的点？一系列的三角形？还是仅仅是一个长长的线？做出的这些提示叫做**图元**(Primitive)，任何一个绘制指令的调用都将把图元传递给OpenGL。这是其中的几个：`GL_POINTS`、`GL_TRIANGLES`、`GL_LINE_STRIP`。

图形渲染管线的第一个部分是**顶点着色器**(Vertex Shader)，它把一个单独的顶点作为输入。顶点着色器主要的目的是把3D坐标转为另一种3D坐标（后面会解释），同时顶点着色器允许我们对顶点属性进行一些基本处理。

图元装配(Primitive Assembly)阶段将顶点着色器输出的所有顶点作为输入（如果是`GL_POINTS`，那么就是一个顶点），并所有的点装配成指定图元的形状；本节例子中是一个三角形。

图元装配阶段的输出会传递给**几何着色器**(Geometry Shader)。几何着色器把图元形式的一系列顶点的集合作为输入，它可以通过产生新顶点构造出新的（或是其它的）图元来生成其他形状。例子中，它生成了另一个三角形。

几何着色器的输出会被传入光栅化阶段(Rasterization Stage)，这里它会把图元映射为最终屏幕上相应的像素，生成供片段着色器(Fragment Shader)使用的片段(Fragment)。在片段着色器运行之前会执行裁切(Clipping)。裁切会丢弃超出你的视图以外的所有像素，用来提升执行效率。

OpenGL中的一个片段是OpenGL渲染一个像素所需的所有数据。

片段着色器的主要目的是计算一个像素的最终颜色，这也是所有OpenGL高级效果产生的地方。通常，片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

在所有对应颜色值确定以后，最终的对象将会被传到最后一个阶段，我们叫做Alpha测试和混合(Blending)阶段。这个阶段检测片段的对应的深度（和模板(Stencil)）值（后面会讲），用它们来判断这个像素是其它物体的前面还是后面，决定是否应该丢弃。这个阶段也会检查alpha值（alpha值定义了一个物体的透明度）并对物体进行混合(Blend)。所以，即使在片段着色器中计算出来了一个像素输出的颜色，在渲染多个三角形的时候最后的像素颜色也可能完全不同。

可以看到，图形渲染管线非常复杂，它包含很多可配置的部分。然而，对于大多数场合，我们只需要配置顶点和片段着色器就行了。几何着色器是可选的，通常使用它默认的着色器就行了。

在现代OpenGL中，我们必须定义至少一个顶点着色器和一个片段着色器（因为GPU中没有默认的顶点/片段着色器）。出于这个原因，刚开始学习现代OpenGL的时候可能会非常困难，因为在你能够渲染自己的第一个三角形之前已经需要了解一大堆知识了。在本节结束你最终渲染出你的三角形的时候，你也会了解到非常多的图形编程知识。

顶点输入

开始绘制图形之前，我们必须先给OpenGL输入一些顶点数据。OpenGL是一个3D图形库，所以我们在OpenGL中指定的所有坐标都是3D坐标（x、y和z）。OpenGL不是简单地把所有的3D坐标变换为屏幕上的2D像素；OpenGL仅当3D坐标在3个轴（x、y和z）上都为-1.0到1.0的范围内时才处理它。所有在所谓的标准化设备坐标(Normalized Device Coordinates)范围内的坐标才会最终呈现在屏幕上（在这个范围以外的坐标都不会显示）。

由于我们希望渲染一个三角形，我们一共要指定三个顶点，每个顶点都有一个3D位置。我们会将它们以标准化设备坐标的形式（OpenGL的可见区域）定义为一个float数组。

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    0.0f, 0.5f, 0.0f
};
```

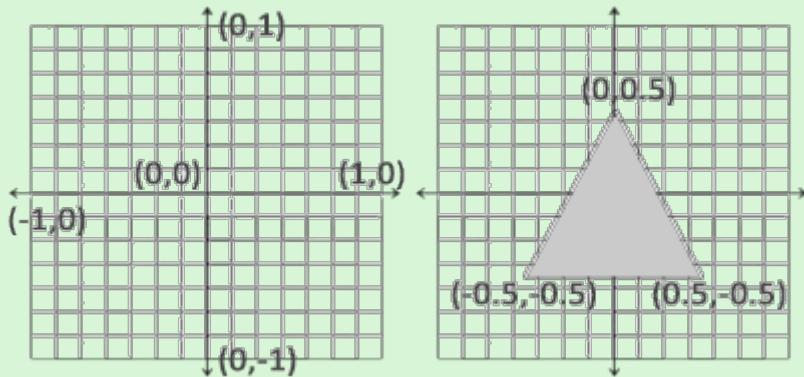
由于OpenGL是在3D空间中工作的，而我们渲染的是一个2D三角形，我们将它顶点的z坐标设置为0.0。这样子的话三角形每一点的深度(Depth，译注2)都是一样的，从而使它看上去像是2D的。

译注2

通常深度可以理解为z坐标，它代表一个像素在空间中和你的距离，如果离你远就可能被别的像素遮挡，你就看不到它了，它会被丢弃，以节省资源。

标准化设备坐标(Normalized Device Coordinates, NDC)

一旦你的顶点坐标已经在顶点着色器中处理过，它们就应该是标准化设备坐标了，标准化设备坐标是一个x、y和z值在-1.0到1.0的一小段空间。任何落在范围外的坐标都会被丢弃/裁剪，不会显示在你的屏幕上。下面你会看到我们定义的在标准化设备坐标中的三角形(忽略z轴)：



与通常的屏幕坐标不同，y轴正方向为向上，(0, 0)坐标是这个图像的中心，而不是左上角。最终你希望所有(变换过的)坐标都在这个坐标空间中，否则它们就不可见了。

你的标准化设备坐标接着会变换为屏幕空间坐标(Screen-space Coordinates)，这是使用你通过`glViewport`函数提供的数据，进行视口变换(Viewport Transform)完成的。所得的屏幕空间坐标又会被变换为片段输入到片段着色器中。

定义这样的顶点数据以后，我们会把它作为输入发送给图形渲染管线的第一个处理阶段：顶点着色器。它会在GPU上创建内存用于储存我们的顶点数据，还要配置OpenGL如何解释这些内存，并且指定其如何发送给显卡。顶点着色器接着会处理我们在内存中指定数量的顶点。

我们通过**顶点缓冲对象**(Vertex Buffer Objects, VBO)管理这个内存，它会在GPU内存（通常被称为显存）中储存大量顶点。使用这些缓冲对象的好处是我们可以一次性的发送一大批数据到显卡上，而不是每个顶点发送一次。从CPU把数据发送到显卡相对较慢，所以只要可能我们都要尝试尽量一次性发送尽可能多的数据。当数据发送至显卡的内存中后，顶点着色器几乎能立即访问顶点，这是个非常快的过程。

顶点缓冲对象是我们在[OpenGL](#)教程中第一个出现的OpenGL对象。就像OpenGL中的其它对象一样，这个缓冲有一个独一无二的ID，所以我们使用`glGenBuffers`函数和一个缓冲ID生成一个VBO对象：

```
unsigned int VBO;
 glGenBuffers(1, &VBO);
```

OpenGL有很多缓冲对象类型，顶点缓冲对象的缓冲类型是`GL_ARRAY_BUFFER`。OpenGL允许我们同时绑定多个缓冲，只要它们是不同的缓冲类型。我们可以使用`glBindBuffer`函数把新创建的缓冲绑定到`GL_ARRAY_BUFFER`目标上：

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

从这一刻起，我们使用的任何（在`GL_ARRAY_BUFFER`目标上的）缓冲调用都会用来配置当前绑定的缓冲(`VBO`)。然后我们可以调用`glBufferData`函数，它会把之前定义的顶点数据复制到缓冲的内存中：

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

`glBufferData`是一个专门用来把用户定义的数据复制到当前绑定缓冲的函数。它的第一个参数是目标缓冲的类型：顶点缓冲对象当前绑定到`GL_ARRAY_BUFFER`目标上。第二个参数指定传输数据的大小(以字节为单位)；用一个简单的`sizeof`计算出顶点数据大小就行。第三个参数是我们希望发送的实际数据。

第四个参数指定了我们希望显卡如何管理给定的数据。它有三种形式：

- `GL_STATIC_DRAW`：数据不会或几乎不会改变。
- `GL_DYNAMIC_DRAW`：数据会被改变很多。
- `GL_STREAM_DRAW`：数据每次绘制时都会改变。

三角形的位置数据不会改变，每次渲染调用时都保持原样，所以它的使用类型最好是`GL_STATIC_DRAW`。如果，比如说一个缓冲中的数据将频繁被改变，那么使用的类型就是`GL_DYNAMIC_DRAW`或`GL_STREAM_DRAW`，这样就能确保显卡把数据放在能够高速写入的内存部分。

现在我们已经把顶点数据储存在显卡的内存中，用`VBO`这个顶点缓冲对象管理。下面我们会创建一个顶点和片段着色器来真正处理这些数据。现在我们开始着手创建它们吧。

顶点着色器

顶点着色器(Vertex Shader)是几个可编程着色器中的一个。如果我们打算做渲染的话，现代OpenGL需要我们至少设置一个顶点和一个片段着色器。我们会简要介绍一下着色器以及配置两个非常简单的着色器来绘制我们第一个三角形。下一节中我们会更详细的讨论着色器。

我们需要做的第一件事是用着色器语言GLSL(OpenGL Shading Language)编写顶点着色器，然后编译这个着色器，这样我们就可以在程序中使用它了。下面你会看到一个非常基础的GLSL顶点着色器的源代码：

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

可以看到，GLSL看起来很像C语言。每个着色器都起始于一个版本声明。OpenGL 3.3以及和更高版本中，GLSL版本号和OpenGL的版本是匹配的（比如说GLSL 420版本对应于OpenGL 4.2）。我们同样明确表示我们会使用核心模式。

下一步，使用`in`关键字，在顶点着色器中声明所有的输入顶点属性(Input Vertex Attribute)。现在我们只关心位置(Position)数据，所以我们只需要一个顶点属性。GLSL有一个向量数据类型，它包含1到4个`float`分量，包含的数量可以从它的后缀数字看出来。由于每个顶点都有一个3D坐标，我们就创建一个`vec3`输入变量`aPos`。我们同样也通过`layout (location = 0)`设定了输入变量的位置值(Location)你后面会看到为什么我们会需要这个位置值。

向量(Vector)

在图形编程中我们经常会使用向量这个数学概念，因为它简明地表达了任意空间中的位置和方向，并且它有非常有用的数学属性。在GLSL中一个向量有最多4个分量，每个分量值都代表空间中的一个坐标，它们可以通过`vec.x`、`vec.y`、`vec.z`和`vec.w`来获取。注意`vec.w`分量不是用作表达空间中的位置的（我们处理的是3D不是4D），而是用在所谓透视除法(Perspective Division)上。我们会在后面的教程中更详细地讨论向量。

为了设置顶点着色器的输出，我们必须把位置数据赋值给预定义的`gl_Position`变量，它在幕后是`vec4`类型的。在`main`函数的最后，我们将`gl_Position`设置的值会成为该顶点着色器的输出。由于我们的输入是一个3分量的向量，我们必须把它转换为4分量的。我们可以把`vec3`的数据作为`vec4`构造器的参数，同时把`w`分量设置为`1.0f`（我们会在后面解释为什么）来完成这一任务。

当前这个顶点着色器可能是我们能想到的最简单的顶点着色器了，因为我们对输入数据什么都没有处理就把它传到着色器的输出了。在真实的程序里输入数据通常都不是标准化设备坐标，所以我们首先必须先把它们转换至OpenGL的可视区域内。

编译着色器

现在，我们暂时将顶点着色器的源代码硬编码在代码文件顶部的C风格字符串中：

```
const char *vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\0";
```

为了能够让OpenGL使用它，我们必须在运行时动态编译它的源代码。

我们首先要做的是创建一个着色器对象，注意还是用ID来引用的。所以我们储存这个顶点着色器为 `unsigned int`，然后用 `glCreateShader` 创建这个着色器：

```
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

我们把需要创建的着色器类型以参数形式提供给 `glCreateShader`。由于我们正在创建一个顶点着色器，传递的参数是 `GL_VERTEX_SHADER`。

下一步我们把这个着色器源码附加到着色器对象上，然后编译它：

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

`glShaderSource` 函数把要编译的着色器对象作为第一个参数。第二参数指定了传递的源码字符串数量，这里只有一个。第三个参数是顶点着色器真正的源码，第四个参数我们先设置为 `NULL`。

你可能会希望检测在调用 `glCompileShader` 后编译是否成功了，如果没成功的话，你还会希望知道错误是什么，这样你才能修复它们。检测编译时错误可以通过以下代码来实现：

```
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

首先我们定义一个整型变量来表示是否成功编译，还定义了一个储存错误消息（如果有的话）的容器。然后我们用 `glGetShaderiv` 检查是否编译成功。如果编译失败，我们会用 `glGetShaderInfoLog` 获取错误消息，然后打印它。

```
if(!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILE_FAILED\n" << infoLog << std::endl;
}
```

如果编译的时候没有检测到任何错误，顶点着色器就被编译成功了。

片段着色器

片段着色器(Fragment Shader)是第二个也是最后一个我们打算创建的用于渲染三角形的着色器。片段着色器所做的是计算像素最后的颜色输出。为了让事情更简单，我们的片段着色器将会一直输出橘黄色。

在计算机图形中颜色被表示为有4个元素的数组：红色、绿色、蓝色和alpha(透明度)分量，通常缩写为RGBA。当在OpenGL或GLSL中定义一个颜色的时候，我们把颜色每个分量的强度设置在0.0到1.0之间。比如说我们设置红为1.0f，绿为1.0f，我们会得到两个颜色的混合色，即黄色。这三种颜色分量的不同调配可以生成超过1600万种不同的颜色！

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

片段着色器只需要一个输出变量，这个变量是一个4分量向量，它表示的是最终的输出颜色，我们应该自己将其计算出来。声明输出变量可以使用 `out` 关键字，这里我们命名为 `FragColor`。下面，我们将一个Alpha值为1.0(1.0代表完全不透明)的橘黄色的 `vec4` 赋值给颜色输出。

编译片段着色器的过程与顶点着色器类似，只不过我们使用 `GL_FRAGMENT_SHADER` 常量作为着色器类型：

```
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

两个着色器现在都编译了，剩下的事情是把两个着色器对象链接到一个用来渲染的 **着色器程序**(Shader Program)中。

着色器程序

着色器程序对象(Shader Program Object)是多个着色器合并之后并最终链接完成的版本。如果要使用刚才编译的着色器我们必须把它们 **链接**(Link)为一个着色器程序对象，然后在渲染对象的时候激活这个着色器程序。已激活着色器程序的着色器将在我们发送渲染调用的时候被使用。

当链接着色器至一个程序的时候，它会把每个着色器的输出链接到下个着色器的输入。当输出和输入不匹配的时候，你会得到一个连接错误。

创建一个程序对象很简单：

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

`glCreateProgram`函数创建一个程序，并返回新创建程序对象的ID引用。现在我们需要把之前编译的着色器附加到程序对象上，然后用`glLinkProgram`链接它们：

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

代码应该很清楚，我们把着色器附加到了程序上，然后用`glLinkProgram`链接。

就像着色器的编译一样，我们也可以检测链接着色器程序是否失败，并获取相应的日志。与上面不同，我们不会调用`glGetShaderiv`和`glGetShaderInfoLog`，现在我们使用：

```
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if(!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    ...
}
```

得到的结果就是一个程序对象，我们可以调用`glUseProgram`函数，用刚创建的程序对象作为它的参数，以激活这个程序对象：

```
glUseProgram(shaderProgram);
```

在`glUseProgram`函数调用之后，每个着色器调用和渲染调用都会使用这个程序对象（也就是之前写的着色器）了。

对了，在把着色器对象链接到程序对象以后，记得删除着色器对象，我们不再需要它们了：

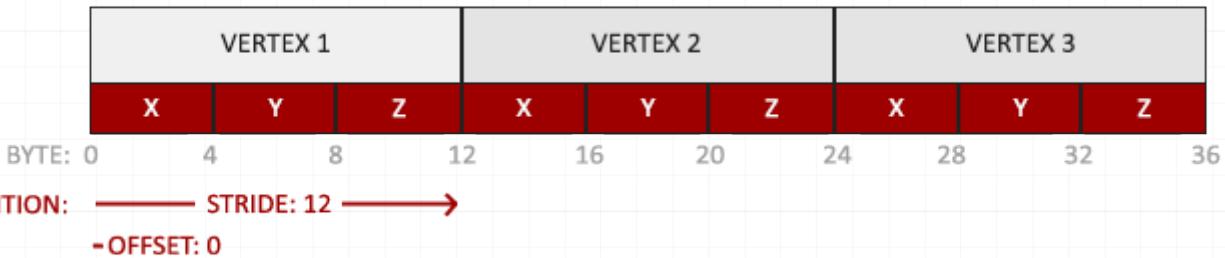
```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

现在，我们已经把输入顶点数据发送给了GPU，并指示了GPU如何在顶点和片段着色器中处理它。就快要完成了，但还没结束，OpenGL还不知道它该如何解释内存中的顶点数据，以及它该如何将顶点数据链接到顶点着色器的属性上。我们需要告诉OpenGL怎么做。

链接顶点属性

顶点着色器允许我们指定任何以顶点属性为形式的输入。这使其具有很强的灵活性的同时，它还的确意味着我们必须手动指定输入数据的哪一个部分对应顶点着色器的哪一个顶点属性。所以，我们必须在渲染前指定OpenGL该如何解释顶点数据。

我们的顶点缓冲数据会被解析为下面这样子：



- 位置数据被储存为32位（4字节）浮点值。
- 每个位置包含3个这样的值。
- 在这3个值之间没有空隙（或其他值）。这几个值在数组中紧密排列(Tightly Packed)。
- 数据中第一个值在缓冲开始的位置。

有了这些信息我们就可以使用`glVertexAttribPointer`函数告诉OpenGL该如何解析顶点数据（应用到逐个顶点属性上）了：

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);
```

`glVertexAttribPointer`函数的参数非常多，所以我会逐一介绍它们：

- 第一个参数指定我们要配置的顶点属性。还记得我们在顶点着色器中使用`layout(location = 0)`定义了`position`顶点属性的位置值(Location)吗？它可以把顶点属性的位置值设置为`0`。因为我们希望把数据传递到这一个顶点属性中，所以这里我们传入`0`。
- 第二个参数指定顶点属性的大小。顶点属性是一个`vec3`，它由3个值组成，所以大小是3。
- 第三个参数指定数据的类型，这里是`GL_FLOAT`(GLSL中`vec*`都是由浮点数值组成的)。
- 下个参数定义我们是否希望数据被标准化(Normalize)。如果我们设置为`GL_TRUE`，所有数据都会被映射到0（对于有符号型signed数据是-1）到1之间。我们把它设置为`GL_FALSE`。
- 第五个参数叫做步长(Stride)，它告诉我们在连续的顶点属性组之间的间隔。由于下个组位置数据在3个`float`之后，我们把步长设置为`3 * sizeof(float)`。要注意的是由于我们知道这个数组是紧密排列的（在两个顶点属性之间没有空隙）我们也可以设置为0来让OpenGL决定具体步长是多少（只有当数值是紧密排列时才可用）。一旦我们有更多的顶点属性，我们就必须更小心地定义每个顶点属性之间的间隔，我们在后面会看到更多的例子（译注：这个参数的意思简单说就是从这个属性第二次出现的地方到整个数组0位置之间有多少字节）。
- 最后一个参数的类型是`void*`，所以需要我们进行这个奇怪的强制类型转换。它表示位置数据在缓冲中起始位置的偏移量(Offset)。由于位置数据在数组的开头，所以这里是0。我们会在后面详细解释这个参数。

每个顶点属性从一个VBO管理的内存中获得它的数据，而具体是从哪个VBO（程序中可以有多个VBO）获取则是通过在调用`glVertexAttribPointer`时绑定到`GL_ARRAY_BUFFER`的VBO决定的。由于在调用`glVertexAttribPointer`之前绑定的是先前定义的`VBO`对象，顶点属性`0`现在会链接到它的顶点数据。

现在我们已经定义了OpenGL该如何解释顶点数据，我们现在应该使用`glEnableVertexAttribArray`，以顶点属性位置值作为参数，启用顶点属性；顶点属性默认是禁用的。自此，所有东西都已经设置好了：我们使用一个顶点缓冲对象将顶点数据初始化至缓冲中，建立了一个顶点和一个片段着色器，并告诉了OpenGL如何把顶点数据链接到顶点着色器的顶点属性上。在OpenGL中绘制一个物体，代码会像是这样：

```
// 0. 复制顶点数组到缓冲中供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. 设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 2. 当我们渲染一个物体时要使用着色器程序
glUseProgram(shaderProgram);
// 3. 绘制物体
someOpenGLFunctionThatDrawsOurTriangle();
```

每当我们绘制一个物体的时候都必须重复这一过程。这看起来可能不多，但是如果超过5个顶点属性，上百个不同物体呢（这其实并不罕见）。绑定正确的缓冲对象，为每个物体配置所有顶点属性很快就变成一件麻烦事。有没有一些方法可以使我们把所有这些状态配置储存在一个对象中，并且可以通过绑定这个对象来恢复状态呢？

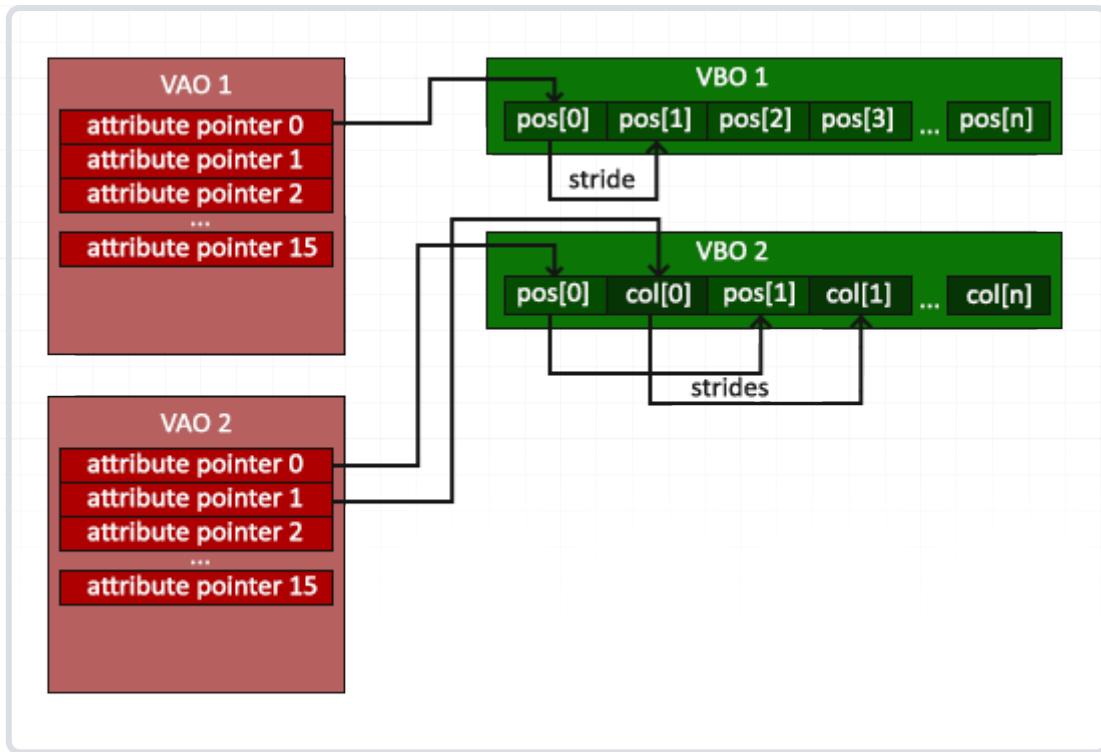
顶点数组对象

顶点数组对象(Vertex Array Object, **VAO**)可以像顶点缓冲对象那样被绑定，任何随后的顶点属性调用都会储存在这个VAO中。这样的好处就是，当配置顶点属性指针时，你只需要将那些调用执行一次，之后再绘制物体的时候只需要绑定相应的VAO就行了。这使在不同顶点数据和属性配置之间切换变得非常简单，只需要绑定不同的VAO就行了。刚刚设置的所有状态都将存储在VAO中

OpenGL的核心模式要求我们使用VAO，所以它知道该如何处理我们的顶点输入。如果我们绑定VAO失败，OpenGL会拒绝绘制任何东西。

一个顶点数组对象会储存以下这些内容：

- `glEnableVertexAttribArray`和`glDisableVertexAttribArray`的调用。
- 通过`glVertexAttribPointer`设置的顶点属性配置。
- 通过`glVertexAttribPointer`调用与顶点属性关联的顶点缓冲对象。



创建一个VAO和创建一个VBO很类似：

```
unsigned int VAO;
 glGenVertexArrays(1, &VAO);
```

要想使用VAO，要做的只是使用`glBindVertexArray`绑定VAO。从绑定之后起，我们应该绑定和配置对应的VBO和属性指针，之后解绑VAO供之后使用。当我们打算绘制一个物体的时候，我们只要在绘制物体前简单地把VAO绑定到希望使用的设定上就行了。这段代码应该看起来像这样：

```
// ...: 初始化代码（只运行一次（除非你的物体频繁改变）） :: ..
// 1. 绑定VAO
glBindVertexArray(VAO);
// 2. 把顶点数组复制到缓冲中供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. 设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);

[...]

// ...: 绘制代码（渲染循环中） :: ..
// 4. 绘制物体
glUseProgram(shaderProgram);
 glBindVertexArray(VAO);
 someOpenGLFunctionThatDrawsOurTriangle();
```

就这么多了！前面做的一切都是等待这一刻，一个储存了我们顶点属性配置和应使用的VBO的顶点数组对象。一般当你打算绘制多个物体时，你首先要生成/配置所有的VAO（和必须的VBO及属性指针），然后储存它们供后面使用。当我们打算绘制物体的时候就拿出相应的VAO，绑定它，绘制完物体后，再解绑VAO。

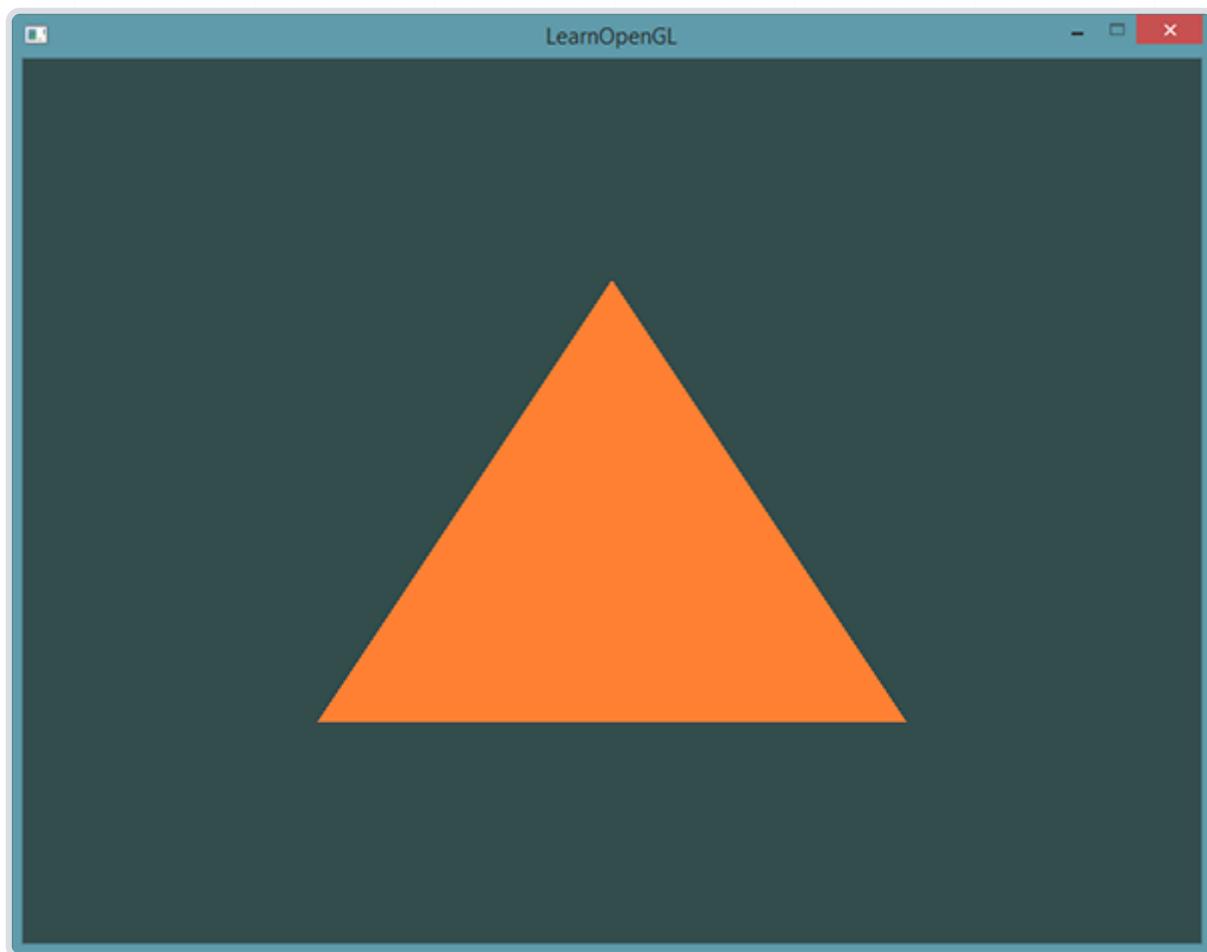
我们一直期待的三角形

要想绘制我们想要的物体，OpenGL给我们提供了`glDrawArrays`函数，它使用当前激活的着色器，之前定义的顶点属性配置，和VBO的顶点数据（通过VAO间接绑定）来绘制图元。

```
glUseProgram(shaderProgram);
 glBindVertexArray(VAO);
 glDrawArrays(GL_TRIANGLES, 0, 3);
```

`glDrawArrays`函数第一个参数是我们打算绘制的OpenGL图元的类型。由于我们在一开始时说过，我们希望绘制的是一个三角形，这里传递`GL_TRIANGLES`给它。第二个参数指定了顶点数组的起始索引，我们这里填`0`。最后一个参数指定我们打算绘制多少个顶点，这里是`3`（我们只从我们的数据中渲染一个三角形，它只有3个顶点长）。

现在尝试编译代码，如果弹出了任何错误，回头检查你的代码。如果你编译通过了，你应该看到下面的结果：



完整的程序源码可以在[这里](#)找到。

如果你的输出和这个看起来不一样，你可能做错了什么。去查看一下源码，检查你是否遗漏了什么东西，或者你也可以在评论区提问。

索引缓冲对象

在渲染顶点这一话题上我们还有最后一个需要讨论的东西——索引缓冲对象(Element Buffer Object, EBO, 也叫Index Buffer Object, IBO)。要解释索引缓冲对象的工作方式最好还是举个例子：假设我们不再绘制一个三角形而是绘制一个矩形。我们可以绘制两个三角形来组成一个矩形(OpenGL主要处理三角形)。这会生成下面的顶点的集合：

```
float vertices[] = {
    // 第一个三角形
    0.5f, 0.5f, 0.0f,    // 右上角
    0.5f, -0.5f, 0.0f,   // 右下角
    -0.5f, 0.5f, 0.0f,   // 左上角
    // 第二个三角形
    0.5f, -0.5f, 0.0f,   // 右下角
    -0.5f, -0.5f, 0.0f,  // 左下角
    -0.5f, 0.5f, 0.0f    // 左上角
};
```

可以看到，有几个顶点叠加了。我们指定了右下角和左上角两次！一个矩形只有4个而不是6个顶点，这样就产生50%的额外开销。当我们有包括上千个三角形的模型之后这个问题会更糟糕，这会产生一大堆浪费。更好的解决方案是只储存不同的顶点，并设定绘制这些顶点的顺序。这样子我们只要储存4个顶点就能绘制矩形了，之后只要指定绘制的顺序就行了。如果OpenGL提供这个功能就好了，对吧？

很幸运，索引缓冲对象的工作方式正是这样的。和顶点缓冲对象一样，EBO也是一个缓冲，它专门储存索引，OpenGL调用这些顶点的索引来决定该绘制哪个顶点。所谓的索引绘制(Indexed Drawing)正是我们问题的解决方案。首先，我们先要定义(不重复的)顶点，和绘制出矩形所需的索引：

```
float vertices[] = {
    0.5f, 0.5f, 0.0f,    // 右上角
    0.5f, -0.5f, 0.0f,   // 右下角
    -0.5f, -0.5f, 0.0f,  // 左下角
    -0.5f, 0.5f, 0.0f    // 左上角
};

unsigned int indices[] = { // 注意索引从0开始!
    0, 1, 3, // 第一个三角形
    1, 2, 3  // 第二个三角形
};
```

你可以看到，当使用索引的时候，我们只定义了4个顶点，而不是6个。下一步我们需要创建索引缓冲对象：

```
unsigned int EBO;
glGenBuffers(1, &EBO);
```

与VBO类似，我们先绑定EBO然后用`glBufferData`把索引复制到缓冲里。同样，和VBO类似，我们会把这些函数调用放在绑定和解绑函数调用之间，只不过这次我们把缓冲的类型定义为`GL_ELEMENT_ARRAY_BUFFER`。

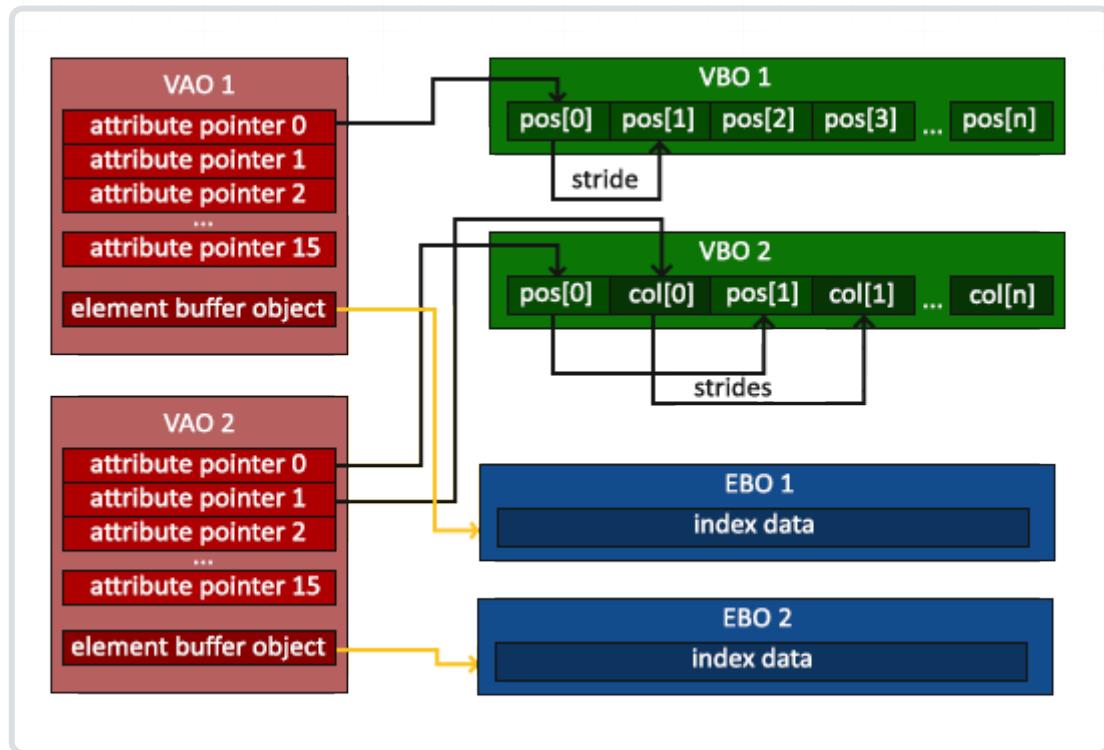
```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

要注意的是，我们传递了`GL_ELEMENT_ARRAY_BUFFER`当作缓冲目标。最后一件要做的事是用`glDrawElements`来替换`glDrawArrays`函数，来指明我们从索引缓冲渲染。使用`glDrawElements`时，我们会使用当前绑定的索引缓冲对象中的索引进行绘制：

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

第一个参数指定了我们绘制的模式，这个和`glDrawArrays`的一样。第二个参数是我们打算绘制顶点的个数，这里填6，也就是说我们一共需要绘制6个顶点。第三个参数是索引的类型，这里是`GL_UNSIGNED_INT`。最后一个参数里我们可以指定EBO中的偏移量（或者传递一个索引数组，但是这是当你不在使用索引缓冲对象的时候），但是我们会在这里填写0。

`glDrawElements`函数从当前绑定到`GL_ELEMENT_ARRAY_BUFFER`目标的EBO中获取索引。这意味着我们必须在每次要用索引渲染一个物体时绑定相应的EBO，这还是有点麻烦。不过顶点数组对象同样可以保存索引缓冲对象的绑定状态。VAO绑定时正在绑定的索引缓冲对象会被保存为VAO的元素缓冲对象。绑定VAO的同时也会自动绑定EBO。



当目标是`GL_ELEMENT_ARRAY_BUFFER`的时候，VAO会储存`glBindBuffer`的函数调用。这也意味着它也会储存解绑调用，所以确保你没有在解绑VAO之前解绑索引数组缓冲，否则它就没有这个EBO配置了。

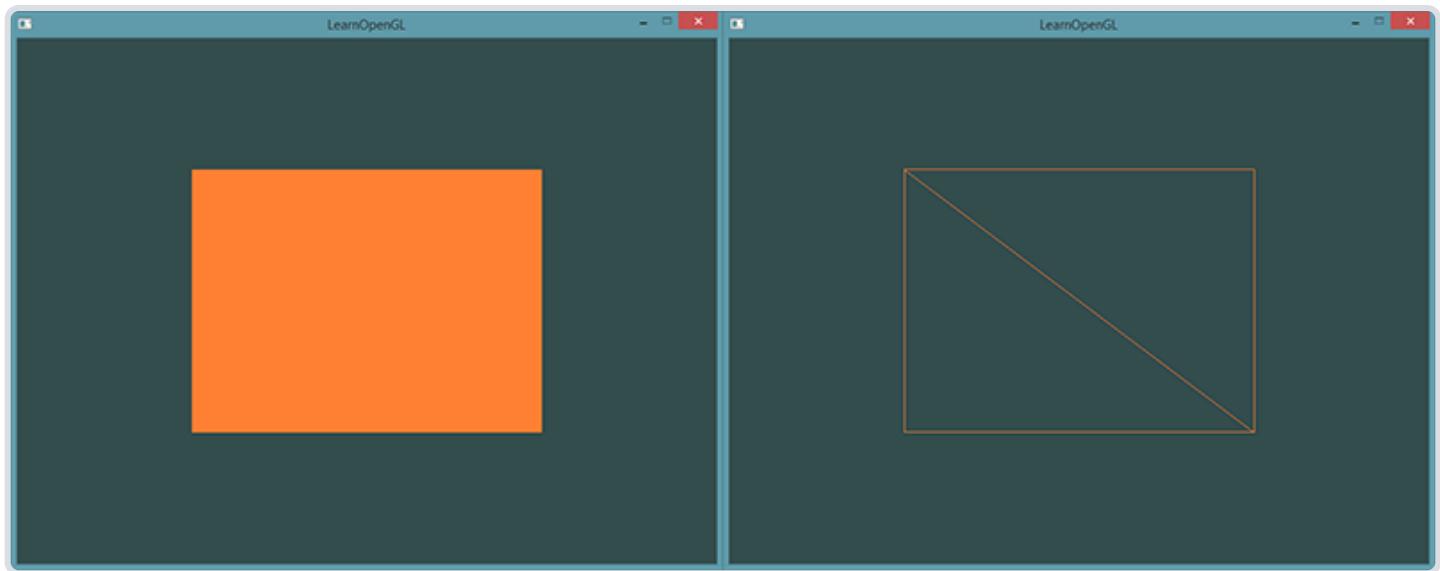
最后的初始化和绘制代码现在看起来像这样：

```
// ...::: 初始化代码 ::...
// 1. 绑定顶点数组对象
glBindVertexArray(VAO);
// 2. 把我们的顶点数组复制到一个顶点缓冲中, 供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. 复制我们的索引数组到一个索引缓冲中, 供OpenGL使用
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. 设定顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);

[...]

// ...::: 绘制代码 (渲染循环中) ::...
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
```

运行程序会获得下面这样的图片的结果。左侧图片看应该起来很熟悉，而右侧的则是使用**线框模式**(Wireframe Mode)绘制的。线框矩形可以显示出矩形的确是由两个三角形组成的。



线框模式(Wireframe Mode)

要想用线框模式绘制你的三角形, 你可以通过 `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` 函数配置OpenGL如何绘制图元。第一个参数表示我们打算将其应用到所有的三角形的正面和背面, 第二个参数告诉我们用线来绘制。之后的绘制调用会一直以线框模式绘制三角形, 直到我们用 `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)` 将其设置回默认模式。

如果你遇到任何错误, 回头检查代码, 看看是否遗漏了什么。同时, 你可以在[这里](#)找到全部源码, 你也可以在评论区自由提问。

如果你像我这样成功绘制出了这个三角形或矩形，那么恭喜你，你成功地通过了现代OpenGL最难部分之一：绘制你自己的第一个三角形。这部分很难，因为在可以绘制第一个三角形之前你需要了解很多知识。幸运的是我们现在已经越过了这个障碍，接下来的教程会比较容易理解一些。

附加资源

- antongerdelan.net/hellotriangle : Anton Gerdelan的渲染第一个三角形教程。
- open.gl/drawing : Alexander Overvoorde的渲染第一个三角形教程。
- antongerdelan.net/vertexbuffers : 顶点缓冲对象的一些深入探讨。
- **调试**: 这个教程中涉及到了很多步骤，如果你在哪卡住了，阅读一点调试的教程是非常值得的（只需要阅读到调试输出部分）。

2.2.6 练习

为了更好的掌握上述概念，我准备了一些练习。建议在继续下一节的学习之前先做完这些练习，确保你对这些知识有比较好的理解。

1. 添加更多顶点到数据中，使用`glDrawArrays`，尝试绘制两个彼此相连的三角形：[参考解答](#)
2. 创建相同的两个三角形，但对它们的数据使用不同的VAO和VBO：[参考解答](#)
3. 创建两个着色器程序，第二个程序使用一个不同的片段着色器，输出黄色；再次绘制这两个三角形，让其中一个输出为黄色：[参考解答](#)

2.2.7 着色器

原文 [Shaders](#)

作者 JoeyDeVries

翻译 [Django](#), Krasjet, Geequlim

校对 暂未校对

在[Hello Triangle](#)教程中提到，着色器(Shader)是运行在GPU上的小程序。这些小程序为图形渲染管线的某个特定部分而运行。从基本意义上来说，着色器只是一种把输入转化为输出的程序。着色器也是一种非常独立的程序，因为它们之间不能相互通信；它们之间唯一的沟通只有通过输入和输出。

前面的教程里我们简要地触及了一点着色器的皮毛，并了解了如何恰当地使用它们。现在我们会用一种更加广泛的形式详细解释着色器，特别是OpenGL着色器语言(GLSL)。

2.2.8 GLSL

着色器是使用一种叫GLSL的类C语言写成的。GLSL是为图形计算量身定制的，它包含一些针对向量和矩阵操作的有用特性。

着色器的开头总是要声明版本，接着是输入和输出变量、uniform和main函数。每个着色器的入口点都是main函数，在这个函数中我们处理所有的输入变量，并将结果输出到输出变量中。如果你不知道什么是uniform也不用担心，我们后面会进行讲解。

一个典型的着色器有下面的结构：

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

int main()
{
    // 处理输入并进行一些图形操作
    ...
    // 输出处理过的结果到输出变量
    out_variable_name = weird_stuff_we_processed;
}
```

当我们特别谈论到顶点着色器的时候，每个输入变量也叫**顶点属性**(Vertex Attribute)。我们能声明的顶点属性是有上限的，它一般由硬件来决定。OpenGL确保至少有16个包含4分量的顶点属性可用，但是有些硬件或许允许更多的顶点属性，你可以查询[GL_MAX_VERTEX_ATTRIBS](#)来获取具体的上限：

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " << nrAttributes << std::endl;
```

通常情况下它至少会返回16个，大部分情况下是够用了。

数据类型

和其他编程语言一样，GLSL有数据类型可以来指定变量的种类。GLSL中包含C等其它语言大部分的默认基础数据类型：`int`、`float`、`double`、`uint`和`bool`。GLSL也有两种容器类型，它们会在这个教程中使用很多，分别是向量(Vector)和矩阵(Matrix)，其中矩阵我们会在之后的教程里再讨论。

向量

GLSL中的向量是一个可以包含有2、3或者4个分量的容器，分量的类型可以是前面默认基础类型的任意一个。它们可以是下面的形式（`n`代表分量的数量）：

类型 含义

<code>vecn</code>	包含 <code>n</code> 个 <code>float</code> 分量的默认向量
<code>bvecn</code>	包含 <code>n</code> 个 <code>bool</code> 分量的向量
<code>ivecn</code>	包含 <code>n</code> 个 <code>int</code> 分量的向量
<code>uvecn</code>	包含 <code>n</code> 个 <code>unsigned int</code> 分量的向量
<code>dvecn</code>	包含 <code>n</code> 个 <code>double</code> 分量的向量

大多数时候我们使用`vecn`，因为`float`足够满足大多数要求了。

一个向量的分量可以通过`vec.x`这种方式获取，这里`x`是指这个向量的第一个分量。你可以分别使用`.x`、`.y`、`.z`和`.w`来获取它们的第1、2、3、4个分量。GLSL也允许你对颜色使用`rgba`，或是对纹理坐标使用`stpq`访问相同的分量。

向量这一数据类型也允许一些有趣而灵活的分量选择方式，叫做**重组**(Swizzling)。重组允许这样的语法：

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

你可以使用上面4个字母任意组合来创建一个和原来向量一样长的（同类型）新向量，只要原来向量有那些分量即可；然而，你不允许在一个`vec2`向量中去获取`.z`元素。我们也可以把一个向量作为一个参数传给不同的向量构造函数，以减少需求参数的数量：

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

向量是一种灵活的数据类型，我们可以把它用在各种输入和输出上。学完教程你会看到很多新颖的管理向量的例子。

输入与输出

虽然着色器是各自独立的小程序，但是它们都是一个整体的一部分，出于这样的原因，我们希望每个着色器都有输入和输出，这样才能进行数据交流和传递。GLSL定义了`in`和`out`关键字专门来实现这个目的。每个着色器使用这两个关键字设定输入和输出，只要一个输出变量与下一个着色器阶段的输入匹配，它就会传递下去。但在顶点和片段着色器中会有点不同。

顶点着色器应该接收的是一种特殊形式的输入，否则就会效率低下。顶点着色器的输入特殊在，它从顶点数据中直接接收输入。为了定义顶点数据该如何管理，我们使用`location`这一元数据指定输入变量，这样我们才可以在CPU上配置顶点属性。我们已经在前面的教程看过这个了，`layout (location = 0)`。顶点着色器需要为它的输入提供一个额外的`layout`标识，这样我们才能把它链接到顶点数据。

你也可以忽略 `layout (location = 0)` 标识符，通过在OpenGL代码中使用`glGetAttribLocation`查询属性位置值（Location），但是我更喜欢在着色器中设置它们，这样会更容易理解而且节省你（和OpenGL）的工作量。

另一个例外是片段着色器，它需要一个 `vec4` 颜色输出变量，因为片段着色器需要生成一个最终输出的颜色。如果你在片段着色器没有定义输出颜色，OpenGL会把你的物体渲染为黑色（或白色）。

所以，如果我们打算从一个着色器向另一个着色器发送数据，我们必须在发送方着色器中声明一个输出，在接收方着色器中声明一个类似的输入。当类型和名字都一样的时候，OpenGL就会把两个变量链接到一起，它们之间就能发送数据了（这是在链接程序对象时完成的）。为了展示这是如何工作的，我们会稍微改动一下之前教程里的那个着色器，让顶点着色器为片段着色器决定颜色。

顶点着色器

```
#version 330 core
layout (location = 0) in vec3 aPos; // 位置变量的属性位置值为0

out vec4 vertexColor; // 为片段着色器指定一个颜色输出

void main()
{
    gl_Position = vec4(aPos, 1.0); // 注意我们如何把一个vec3作为vec4的构造器的参数
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // 把输出变量设置为暗红色
}
```

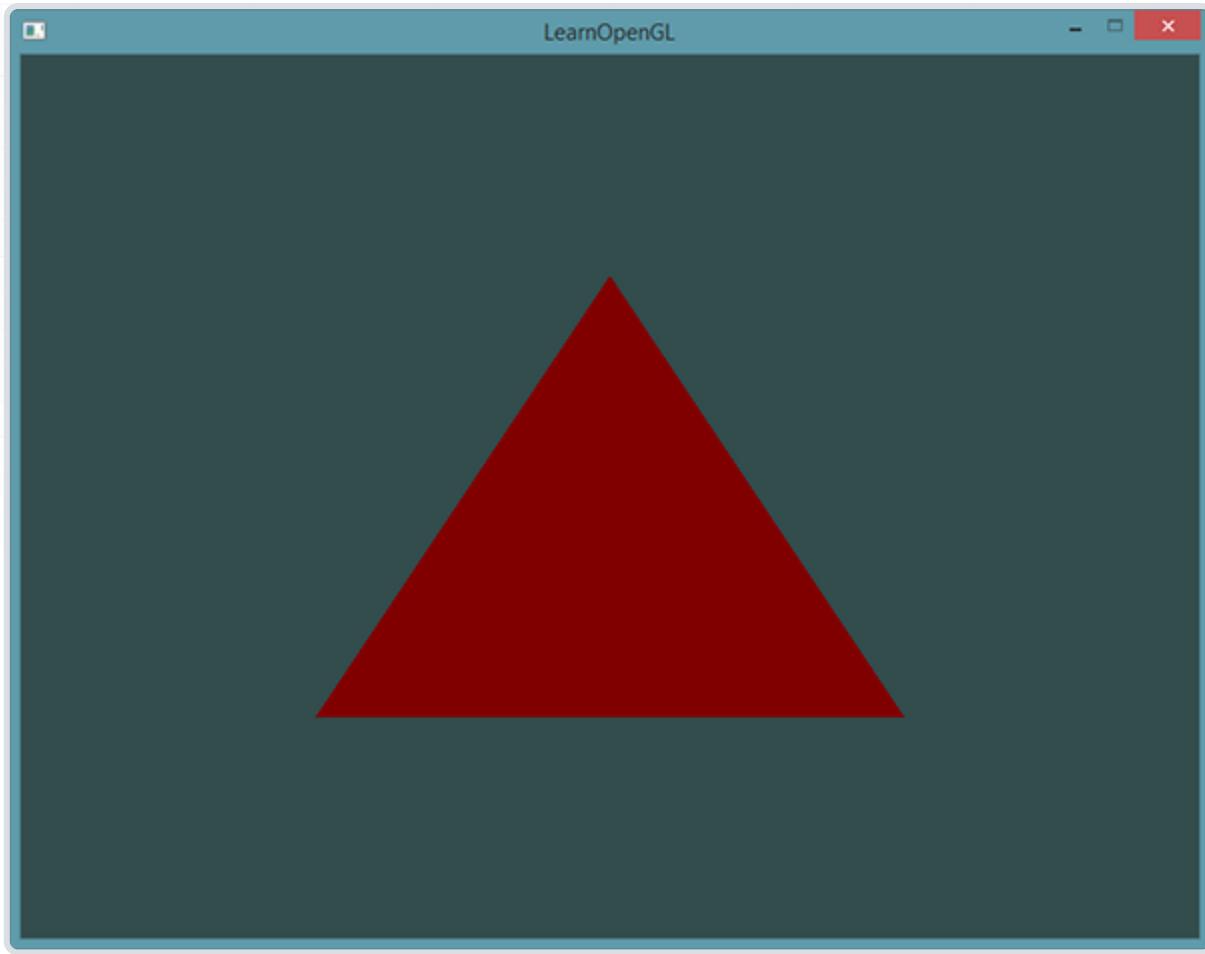
片段着色器

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // 从顶点着色器传来的输入变量（名称相同、类型相同）

void main()
{
    FragColor = vertexColor;
}
```

你可以看到我们在顶点着色器中声明了一个 `vertexColor` 变量作为 `vec4` 输出，并在片段着色器中声明了一个类似的 `vertexColor`。由于它们名字相同且类型相同，片段着色器中的 `vertexColor` 就和顶点着色器中的 `vertexColor` 链接了。由于我们在顶点着色器中将颜色设置为深红色，最终的片段也是深红色的。下面的图片展示了输出结果：



完成了！我们成功地从顶点着色器向片段着色器发送数据。让我们更上一层楼，看看能否从应用程序中直接给片段着色器发送一个颜色！

Uniform

Uniform是一种从CPU中的应用向GPU中的着色器发送数据的方式，但uniform和顶点属性有些不同。首先，uniform是全局的(Global)。全局意味着uniform变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问。第二，无论你把uniform值设置成什么，uniform会一直保存它们的数据，直到它们被重置或更新。

我们可以在一个着色器中添加 `uniform` 关键字至类型和变量名前来声明一个GLSL的uniform。从此处开始我们就可以在着色器中使用新声明的uniform了。我们来看看这次是否能通过uniform设置三角形的颜色：

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // 在OpenGL程序代码中设定这个变量

void main()
{
    FragColor = ourColor;
}
```

我们在片段着色器中声明了一个uniform `vec4` 的`ourColor`，并把片段着色器的输出颜色设置为uniform值的内容。因为uniform是全局变量，我们可以在任何着色器中定义它们，而无需通过顶点着色器作为中介。顶点着色器中不需要这个uniform，所以我们不用在那里定义它。

如果你声明了一个uniform却在GLSL代码中没用过，编译器会静默移除这个变量，导致最后编译出的版本中并不会包含它，这可能导致几个非常麻烦的错误，记住这点！

这个uniform现在还是空的；我们还没有给它添加任何数据，所以下面我们就做这件事。我们首先需要找到着色器中uniform属性的索引/位置值。当我们得到uniform的索引/位置值后，我们就可以更新它的值了。这次我们不去给像素传递单独一个颜色，而是让它随着时间改变颜色：

```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
 glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

首先我们通过`glfwGetTime()`获取运行的秒数。然后我们使用`sin`函数让颜色在0.0到1.0之间改变，最后将结果储存到`greenValue`里。

接着，我们用`glGetUniformLocation`查询uniform `ourColor`的位置值。我们为查询函数提供着色器程序和uniform的名字（这是我们希望获得的位置值的来源）。如果`glGetUniformLocation`返回`-1`就代表没有找到这个位置值。最后，我们可以通过`glUniform4f`函数设置uniform值。注意，查询uniform地址不要求你之前使用过着色器程序，但是更新一个uniform之前你必须先使用程序（调用`glUseProgram`），因为它是在当前激活的着色器程序中设置uniform的。

因为OpenGL在其核心是一个C库，所以它不支持类型重载，在函数参数不同的时候就要为其定义新的函数；`glUniform`是一个典型例子。这个函数有一个特定的后缀，标识设定的uniform的类型。可能的后缀有：

后缀含义

- `f` 函数需要一个float作为它的值
- `i` 函数需要一个int作为它的值
- `ui` 函数需要一个unsigned int作为它的值
- `3f` 函数需要3个float作为它的值
- `fv` 函数需要一个float向量/数组作为它的值

每当你打算配置一个OpenGL的选项时就可以简单地根据这些规则选择适合你的数据类型的重载函数。在我们的例子里，我们希望分别设定uniform的4个float值，所以我们通过`glUniform4f`传递我们的数据（注意，我们也可以使用`fv`版本）。

现在你知道如何设置uniform变量的值了，我们可以使用它们来渲染了。如果我们打算让颜色慢慢变化，我们就要在游戏循环的每一次迭代中（所以他会逐帧改变）更新这个uniform，否则三角形就不会改变颜色。下面我们就计算`greenValue`然后每个渲染迭代都更新这个uniform：

```
while(!glfwWindowShouldClose(window))
{
    // 输入
    processInput(window);

    // 渲染
    // 清除颜色缓冲
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // 记得激活着色器
    glUseProgram(shaderProgram);

    // 更新uniform颜色
    float timeValue = glfwGetTime();
    float greenValue = sin(timeValue) / 2.0f + 0.5f;
    int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);

    // 绘制三角形
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // 交换缓冲并查询IO事件
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

这里的代码对之前代码是一次非常直接的修改。这次，我们在每次迭代绘制三角形前先更新uniform值。如果你正确更新了uniform，你会看到你的三角形逐渐由绿变黑再变回绿色。

如果你在哪儿卡住了，可以到[这里](#)查看源码。

可以看到，uniform对于设置一个在渲染迭代中会改变的属性是一个非常有用的工具，它也是一个在程序和着色器间数据交互的很好工具，但假如我们打算为每个顶点设置一个颜色的时候该怎么办？这种情况下，我们就不得不声明和顶点数目一样多的uniform了。在这一问题上更好的解决方案是在顶点属性中包含更多的数据，这是我们接下来要做的事情。

更多属性！

在前面的教程中，我们了解了如何填充VBO、配置顶点属性指针以及如何把它们都储存到一个VAO里。这次，我们同样打算把颜色数据加进顶点数据中。我们将把颜色数据添加为3个float值至`vertices`数组。我们将把三角形的三个角分别指定为红色、绿色和蓝色：

```
float vertices[] = {
    // 位置          // 颜色
    0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // 右下
    -0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, // 左下
    0.0f, 0.5f, 0.0f, 0.0f, 0.0f, 1.0f    // 顶部
};
```

由于现在有更多的数据要发送到顶点着色器，我们有必要去调整一下顶点着色器，使它能够接收颜色值作为一个顶点属性输入。需要注意的是我们用 `layout` 标识符来把 `aColor` 属性的位置值设置为1：

```
#version 330 core
layout (location = 0) in vec3 aPos; // 位置变量的属性位置值为 0
layout (location = 1) in vec3 aColor; // 颜色变量的属性位置值为 1

out vec3 ourColor; // 向片段着色器输出一个颜色

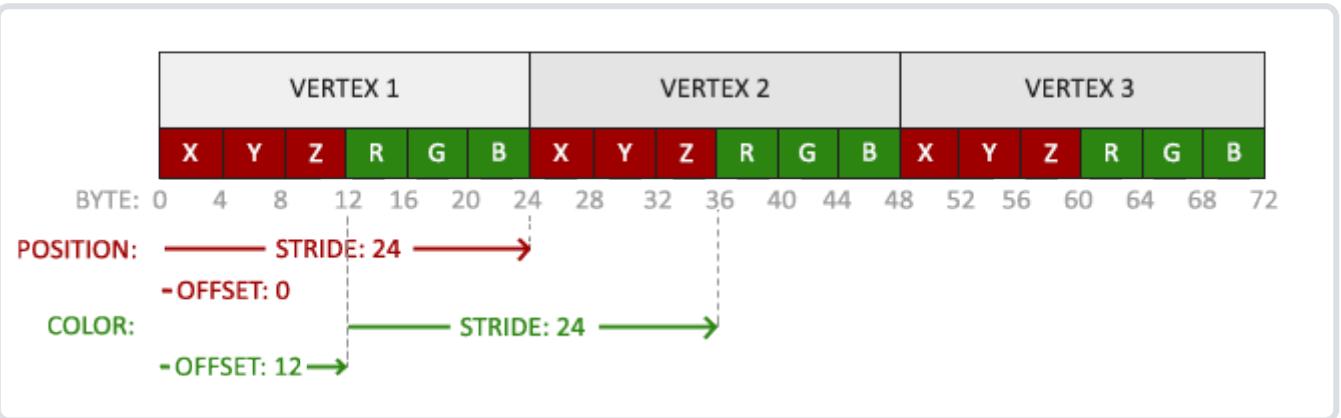
void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // 将ourColor设置为我们从顶点数据那里得到的输入颜色
}
```

由于我们不再使用 `uniform` 来传递片段的颜色了，现在使用 `ourColor` 输出变量，我们必须再修改一下片段着色器：

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

因为我们添加了另一个顶点属性，并且更新了VBO的内存，我们就必须重新配置顶点属性指针。更新后的VBO内存中的数据现在看起来像这样：



知道了现在使用的布局，我们就可以使用 `glVertexAttribPointer` 函数更新顶点格式，

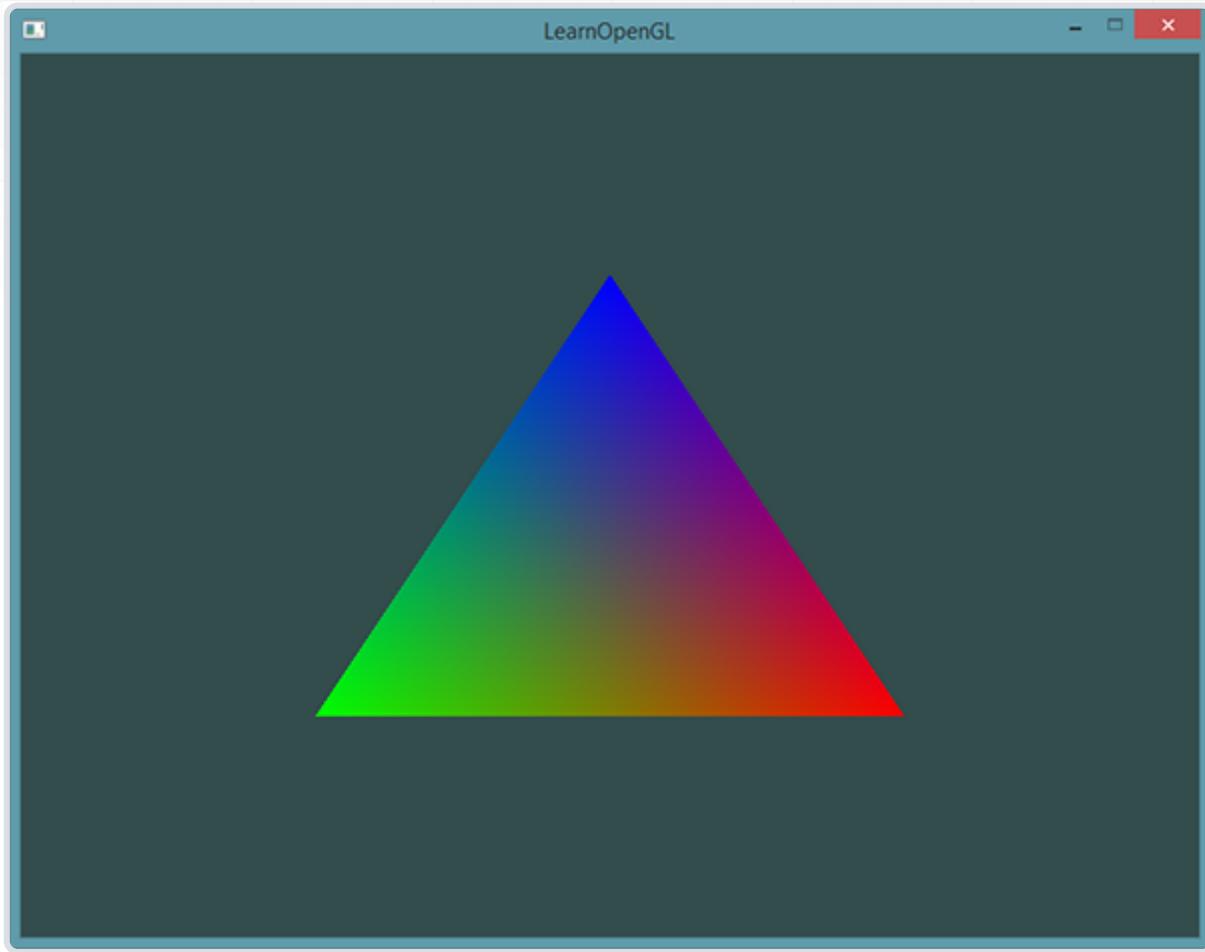
```
// 位置属性
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);
// 颜色属性
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
 glEnableVertexAttribArray(1);
```

`glVertexAttribPointer` 函数的前几个参数比较明了。这次我们配置属性位置值为1的顶点属性。颜色值有3个float那么大，我们不去标准化这些值。

由于我们现在有了两个顶点属性，我们不得不重新计算步长值。为获得数据队列中下一个属性值（比如位置向量的下个 `x` 分量）我们必须向右移动6个float，其中3个是位置值，另外3个是颜色值。这使我们的步长值为6乘以float的字节数（=24字节）。

同样，这次我们必须指定一个偏移量。对于每个顶点来说，位置顶点属性在前，所以它的偏移量是0。颜色属性紧随位置数据之后，所以偏移量就是 `3 * sizeof(float)`，用字节来计算就是12字节。

运行程序你应该会看到如下结果：



如果你在哪卡住了，可以在[这里](#)查看源码。

这个图片可能不是你所期望的那种，因为我们只提供了3个颜色，而不是我们现在看到的大调色板。这是在片段着色器中进行的所谓**片段插值**(Fragment Interpolation)的结果。当渲染一个三角形时，光栅化(Rasterization)阶段通常会造成比原指定顶点更多的片段。光栅会根据每个片段在三角形形状上所处相对位置决定这些片段的位置。

基于这些位置，它会**插值**(Interpolate)所有片段着色器的输入变量。比如说，我们有一个线段，上面的端点是绿色的，下面的端点是蓝色的。如果一个片段着色器在线段的70%的位置运行，它的颜色输入属性就会是一个绿色和蓝色的线性结合；更精确地说就是30%蓝 + 70%绿。

这正是在这个三角形中发生了什么。我们有3个顶点，和相应的3个颜色，从这个三角形的像素来看它可能包含50000左右的片段，片段着色器为这些像素进行插值颜色。如果你仔细看这些颜色就应该能明白了：红首先变成到紫再变为蓝色。片段插值会被应用到片段着色器的所有输入属性上。

2.2.9 我们自己的着色器类

编写、编译、管理着色器是件麻烦事。在着色器主题的最后，我们会写一个类来让我们的生活轻松一点，它可以从硬盘读取着色器，然后编译并链接它们，并对它们进行错误检测，这就变得很好用了。这也会让你了解该如何封装目前所学的知识到一个抽象对象中。

我们会把着色器类全部放在在头文件里，主要是为了学习用途，当然也方便移植。我们先来添加必要的include，并定义类结构：

```
#ifndef SHADER_H
#define SHADER_H

#include <glad/glad.h>; // 包含glad来获取所有的必须OpenGL头文件

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader
{
public:
    // 程序ID
    unsigned int ID;

    // 构造器读取并构建着色器
    Shader(const char* vertexPath, const char* fragmentPath);
    // 使用/激活程序
    void use();
    // uniform工具函数
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
};

#endif
```

在上面，我们在头文件顶部使用了几个预处理指令(Preprocessor Directives)。这些预处理指令会告知你的编译器只在它没被包含过的情况下才包含和编译这个头文件，即使多个文件都包含了这个着色器头文件。它是用来防止链接冲突的。

着色器类储存了着色器程序的ID。它的构造器需要顶点和片段着色器源代码的文件路径，这样我们就可以把源码的文本文件储存在硬盘上了。除此之外，为了让我们的生活更轻松一点，还加入了一些工具函数：use用来激活着色器程序，所有的set...函数能够查询一个uniform的位置值并设置它的值。

从文件读取

我们使用C++文件流读取着色器内容，储存到几个 `string` 对象里：

```
Shader(const char* vertexPath, const char* fragmentPath)
{
    // 1. 从文件路径中获取顶点/片段着色器
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // 保证ifstream对象可以抛出异常：
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        // 打开文件
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // 读取文件的缓冲内容到数据流中
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // 关闭文件处理器
        vShaderFile.close();
        fShaderFile.close();
        // 转换数据流到string
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch(std::ifstream::failure e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char* fShaderCode = fragmentCode.c_str();
    [...]
```

下一步，我们需要编译和链接着色器。注意，我们也将检查编译/链接是否失败，如果失败则打印编译时错误，调试的时候这些错误输出会及其重要（你总会需要这些错误日志的）：

```
// 2. 编译着色器
unsigned int vertex, fragment;
int success;
char infoLog[512];

// 顶点着色器
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
// 打印编译错误 (如果有的话)
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILE_FAILED\n" << infoLog << std::endl;
};

// 片段着色器也类似
[...]

// 着色器程序
ID = glCreateProgram();
glAttachShader(ID, vertex);
glAttachShader(ID, fragment);
glLinkProgram(ID);
// 打印连接错误 (如果有的话)
glGetProgramiv(ID, GL_LINK_STATUS, &success);
if(!success)
{
    glGetProgramInfoLog(ID, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}

// 删除着色器，它们已经链接到我们的程序中了，已经不再需要了
glDeleteShader(vertex);
glDeleteShader(fragment);
```

`use`函数非常简单：

```
void use()
{
    glUseProgram(ID);
}
```

uniform的setter函数也很类似：

```
void setBool(const std::string &name, bool value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}
void setInt(const std::string &name, int value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}
void setFloat(const std::string &name, float value) const
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}
```

现在我们就写完了一个完整的[着色器类](#)。使用这个着色器类很简单；只要创建一个着色器对象，从那一点开始我们就可以开始使用了：

```
Shader ourShader("path/to/shaders/shader.vs", "path/to/shaders/shader.fs");
...
while(...)
{
    ourShader.use();
    ourShader.setFloat("someUniform", 1.0f);
    DrawStuff();
}
```

我们把顶点和片段着色器储存为两个叫做 `shader.vs` 和 `shader.fs` 的文件。你可以使用自己喜欢的名字命名着色器文件；我自己觉得用 `.vs` 和 `.fs` 作为扩展名很直观。

你可以在[这里](#)找到使用[新着色器类](#)的源代码。注意你可以点击源码中的着色器文件路径来查看每一个着色器的源代码。

2.2.10 练习

1. 修改顶点着色器让三角形上下颠倒：[参考解答](#)
2. 使用uniform定义一个水平偏移量，在顶点着色器中使用这个偏移量把三角形移动到屏幕右侧：[参考解答](#)
3. 使用`out`关键字把顶点位置输出到片段着色器，并将片段的颜色设置为与顶点位置相等（来看看连顶点位置值都在三角形中被插值的结果）。做完这些后，尝试回答下面的问题：为什么在三角形的左下角是黑的？：[参考解答](#)

2.2.11 纹理

原文 [Textures](#)

作者 JoeyDeVries

翻译 [Django](#), [Krasjet](#), [Geequlim](#), [BLumia](#)

校对 暂未校对

译注

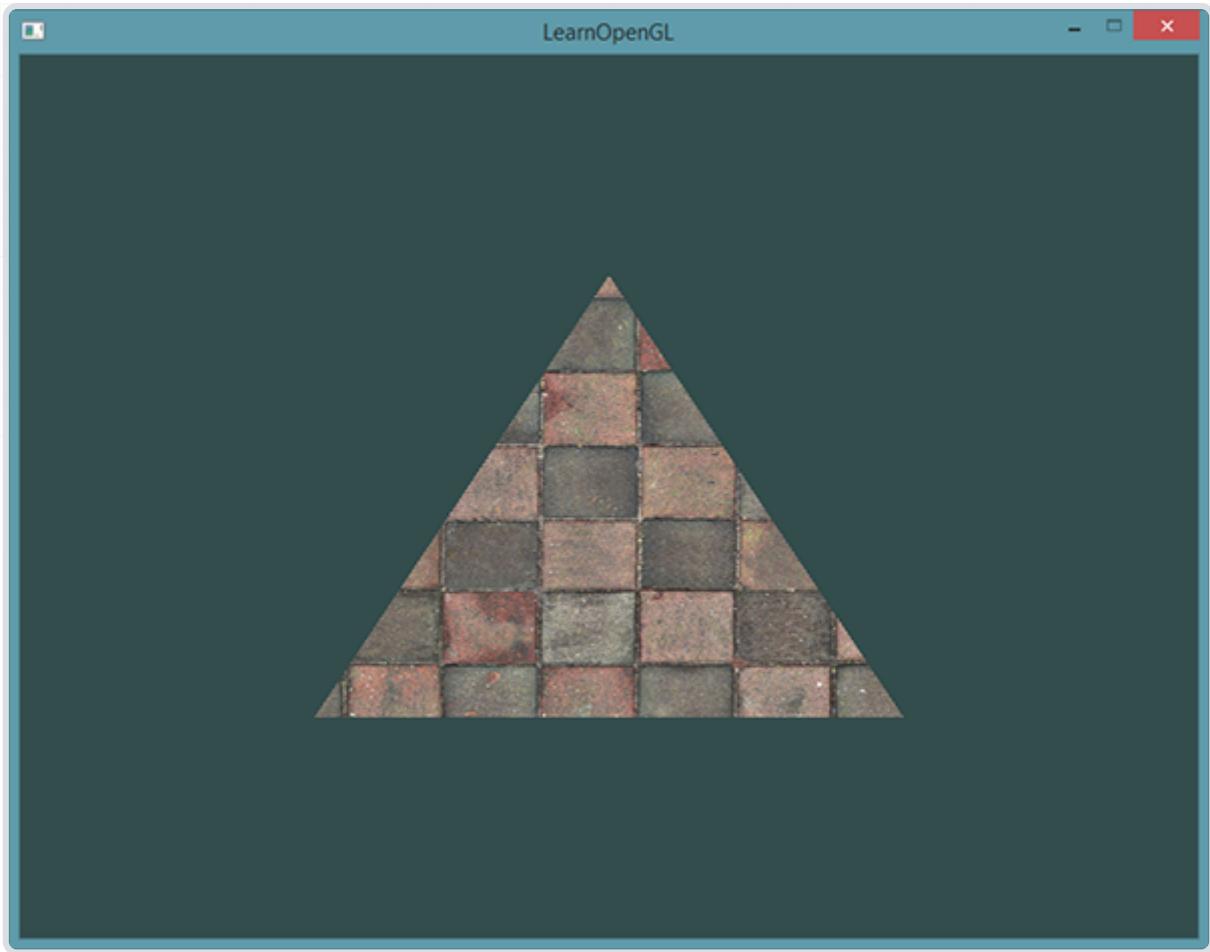
注意，由于作者对教程做出了更新，之前本节使用的是SOIL库，但现在改为了使用 `stb_image.h` 库，关于SOIL配置的部分现在已经被修改，但我仍决定将这部分教程保留起来，放到一个历史存档中，如果有需要的话可以到[这里](#)来查看。

我们已经了解到，我们可以为每个顶点添加颜色来增加图形的细节，从而创建出有趣的图像。但是，如果想让图形看起来更真实，我们就必须有足够多的顶点，从而指定足够多的颜色。这将会产生很多额外开销，因为每个模型都会需求更多的顶点，每个顶点又需求一个颜色属性。

艺术家和程序员更喜欢使用纹理(Texture)。纹理是一个2D图片（甚至也有1D和3D的纹理），它可以用来添加物体的细节；你可以想象纹理是一张绘有砖块的纸，无缝折叠贴合到你的3D的房子上，这样你的房子看起来就像有砖墙外表了。因为我们可以在一张图片上插入非常多的细节，这样就可以让物体非常精细而不用指定额外的顶点。

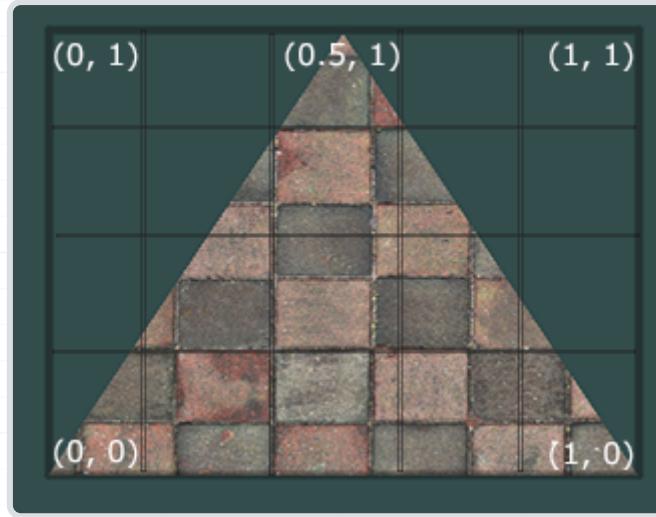
除了图像以外，纹理也可以被用来储存大量的数据，这些数据可以发送到着色器上，但是这不是我们现在的主题。

下面你会看到之前教程的那个三角形贴上了一张[砖墙](#)图片。



为了能够把纹理映射(Map)到三角形上，我们需要指定三角形的每个顶点各自对应纹理的哪个部分。这样每个顶点就会关联着一个纹理坐标(Texture Coordinate)，用来标明该从纹理图像的哪个部分采样(译注：采集片段颜色)。之后在图形的其它片段上进行片段插值(Fragment Interpolation)。

纹理坐标在x和y轴上，范围为0到1之间(注意我们使用的是2D纹理图像)。使用纹理坐标获取纹理颜色叫做采样(Sampling)。纹理坐标起始于(0, 0)，也就是纹理图片的左下角，终始于(1, 1)，即纹理图片的右上角。下面的图片展示了我们是如何把纹理坐标映射到三角形上的。



我们为三角形指定了3个纹理坐标点。如上图所示，我们希望三角形的左下角对应纹理的左下角，因此我们把三角形左下角顶点的纹理坐标设置为(0, 0)；三角形的上顶点对应于图片的上中位置所以我们把它的纹理坐标设置为(0.5, 1.0)；同理右下方的顶点设置为(1, 0)。我们只要给顶点着色器传递这三个纹理坐标就行了，接下来它们会被传片段着色器中，它会为每个片段进行纹理坐标的插值。

纹理坐标看起来就像这样：

```
float texCoords[] = {
    0.0f, 0.0f, // 左下角
    1.0f, 0.0f, // 右下角
    0.5f, 1.0f // 上中
};
```

对纹理采样的解释非常宽松，它可以采用几种不同的插值方式。所以我们需要自己告诉OpenGL该怎样对纹理采样。

纹理环绕方式

纹理坐标的范围通常是从(0, 0)到(1, 1)，那如果我们把纹理坐标设置在范围之外会发生什么？OpenGL默认的行为是重复这个纹理图像（我们基本上忽略浮点纹理坐标的整数部分），但OpenGL提供了更多的选择：

环绕方式	描述
GL_REPEAT	对纹理的默认行为。重复纹理图像。
GL_MIRRORED_REPEAT	和GL_REPEAT一样，但每次重复图片是镜像放置的。
GL_CLAMP_TO_EDGE	纹理坐标会被约束在0到1之间，超出的部分会重复纹理坐标的边缘，产生一种边缘被拉伸的效果。
GL_CLAMP_TO_BORDER	超出的坐标为用户指定的边缘颜色。

当纹理坐标超出默认范围时，每个选项都有不同的视觉效果输出。我们来看看这些纹理图像的例子：



前面提到的每个选项都可以使用`glTexParameter*`函数对单独的一个坐标轴设置 (`s`、`t` (如果是使用3D纹理那么还有一个`r`) 它们和 `x`、`y`、`z` 是等价的) :

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

第一个参数指定了纹理目标; 我们使用的是2D纹理, 因此纹理目标是`GL_TEXTURE_2D`。第二个参数需要我们指定设置的选项与应用的纹理轴。我们打算配置的是`WRAP` 选项, 并且指定 `s` 和 `t` 轴。最后一个参数需要我们传递一个环绕方式(Wrapping), 在这个例子中 OpenGL会给当前激活的纹理设定纹理环绕方式为`GL_MIRRORED_REPEAT`。

如果我们选择`GL_CLAMP_TO_BORDER`选项, 我们还需要指定一个边缘的颜色。这需要使用`glTexParameter`函数的`fv` 后缀形式, 用`GL_TEXTURE_BORDER_COLOR`作为它的选项, 并且传递一个float数组作为边缘的颜色值:

```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

纹理过滤

纹理坐标不依赖于分辨率(Resolution), 它可以是任意浮点值, 所以OpenGL需要知道怎样将纹理像素(Texture Pixel, 也叫Texel, 译注1) 映射到纹理坐标。当你有一个很大的物体但是纹理的分辨率很低的时候这就变得很重要了。你可能已经猜到了, OpenGL也有对于纹理过滤(Texture Filtering)的选项。纹理过滤有很多个选项, 但是现在我们只讨论最重要的两种: `GL_NEAREST`和`GL_LINEAR`。

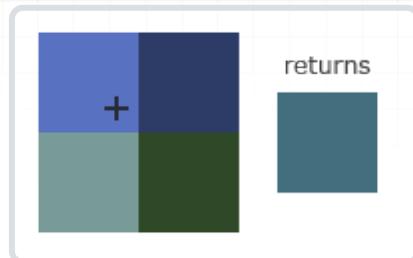
译注1

Texture Pixel也叫Texel, 你可以想象你打开一张 `.jpg` 格式图片, 不断放大你会发现它是由无数像素点组成的, 这个点就是纹理像素; 注意不要和纹理坐标搞混, 纹理坐标是你给模型顶点设置的那个数组, OpenGL以这个顶点的纹理坐标数据去查找纹理图像上的像素, 然后进行采样提取纹理像素的颜色。

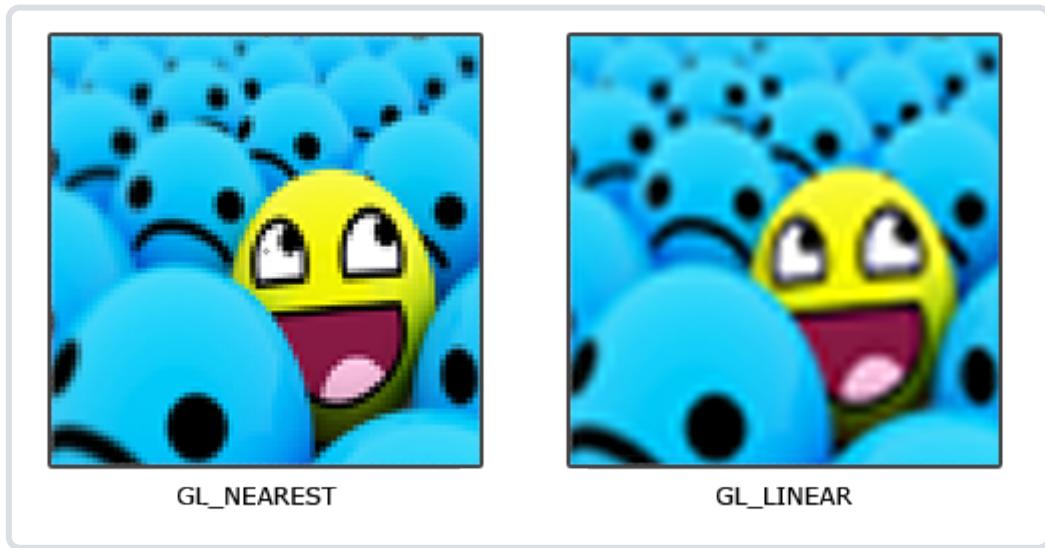
`GL_NEAREST` (也叫邻近过滤, Nearest Neighbor Filtering) 是OpenGL默认的纹理过滤方式。当设置为`GL_NEAREST`的时候, OpenGL会选择中心点最接近纹理坐标的那个像素。下图中你可以看到四个像素, 加号代表纹理坐标。左上角那个纹理像素的中心距离纹理坐标最近, 所以它会被选择为样本颜色:



`GL_NEAREST` (也叫线性过滤, (Bi)linear Filtering) 它会基于纹理坐标附近的纹理像素, 计算出一个插值, 近似出这些纹理像素之间的颜色。一个纹理像素的中心距离纹理坐标越近, 那么这个纹理像素的颜色对最终的样本颜色的贡献越大。下图中你可以看到返回的颜色是邻近像素的混合色:



那么这两种纹理过滤方式有怎样的视觉效果呢? 让我们看看在一个很大的物体上应用一张低分辨率的纹理会发生什么吧 (纹理被放大了, 每个纹理像素都能看到) :



`GL_NEAREST`产生了颗粒状的图案, 我们能够清晰看到组成纹理的像素, 而`GL_LINEAR`能够产生更平滑的图案, 很难看出单个的纹理像素。`GL_LINEAR`可以产生更真实的输出, 但有些开发者更喜欢8-bit风格, 所以他们会用`GL_NEAREST`选项。

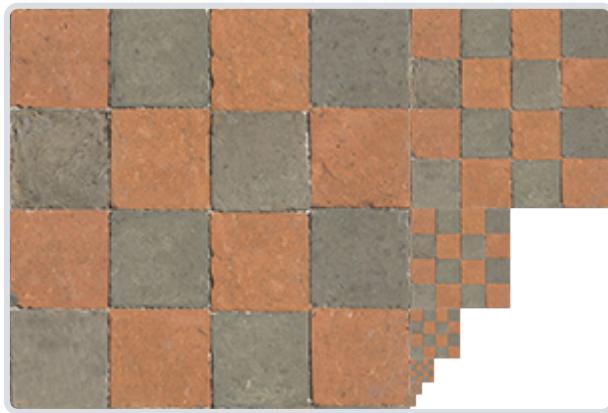
当进行放大(Magnify)和缩小(Minify)操作的时候可以设置纹理过滤的选项, 比如你可以在纹理被缩小的时候使用邻近过滤, 被放大时使用线性过滤。我们需要使用`glTexParameter*`函数为放大和缩小指定过滤方式。这段代码看起来会和纹理环绕方式的设置很相似:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

多级渐远纹理

想象一下，假设我们有一个包含着上千物体的大房间，每个物体上都有纹理。有些物体会很远，但其纹理会拥有与近处物体同样高的分辨率。由于远处的物体可能只产生很少的片段，OpenGL从高分辨率纹理中为这些片段获取正确的颜色值就很困难，因为它需要对一个跨过纹理很大部分的片段只拾取一个纹理颜色。在小物体上这会产生不真实的感觉，更不用说对它们使用高分辨率纹理浪费内存的问题了。

OpenGL使用一种叫做**多级渐远纹理**(Mipmap)的概念来解决这个问题，它简单来说就是一系列的纹理图像，后一个纹理图像是前一个的二分之一。多级渐远纹理背后的理念很简单：距观察者的距离超过一定的阈值，OpenGL会使用不同的多级渐远纹理，即最适合物体的距离的那个。由于距离远，解析度不高也不会被用户注意到。同时，多级渐远纹理另一加分之处是它的性能非常好。让我们看一下多级渐远纹理是什么样子的：



手工为每个纹理图像创建一系列多级渐远纹理很麻烦，幸好OpenGL有一个`glGenerateMipmaps`函数，在创建完一个纹理后调用它OpenGL就会承担接下来的所有工作了。后面的教程中你会看到该如何使用它。

在渲染中切换多级渐远纹理级别(Level)时，OpenGL在两个不同级别的多级渐远纹理层之间会产生不真实的生硬边界。就像普通的纹理过滤一样，切换多级渐远纹理级别时你也可以在两个不同多级渐远纹理级别之间使用`NEAREST`和`LINEAR`过滤。为了指定不同多级渐远纹理级别之间的过滤方式，你可以使用下面四个选项中的一个代替原有的过滤方式：

过滤方式	描述
<code>GL_NEAREST_MIPMAP_NEAREST</code>	使用最邻近的多级渐远纹理来匹配像素大小，并使用邻近插值进行纹理采样
<code>GL_LINEAR_MIPMAP_NEAREST</code>	使用最邻近的多级渐远纹理级别，并使用线性插值进行采样
<code>GL_NEAREST_MIPMAP_LINEAR</code>	在两个最匹配像素大小的多级渐远纹理之间进行线性插值，使用邻近插值进行采样
<code>GL_LINEAR_MIPMAP_LINEAR</code>	在两个邻近的多级渐远纹理之间使用线性插值，并使用线性插值进行采样

就像纹理过滤一样，我们可以使用`glTexParameter`i将过滤方式设置为前面四种提到的方法之一：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

一个常见的错误是，将放大过滤的选项设置为多级渐远纹理过滤选项之一。这样没有任何效果，因为多级渐远纹理主要是使用在纹理被缩小的情况下：纹理放大不会使用多级渐远纹理，为放大过滤设置多级渐远纹理的选项会产生一个`GL_INVALID_ENUM`错误代码。

2.2.12 加载与创建纹理

使用纹理之前要做的第一件事是把它们加载到我们的应用中。纹理图像可能被储存为各种各样的格式，每种都有自己的数据结构和排列，所以我们如何才能把这些图像加载到应用中呢？一个解决方案是选一个需要的文件格式，比如 `.PNG`，然后自己写一个图像加载器，把图像转化为字节序列。写自己的图像加载器虽然不难，但仍然挺麻烦的，而且如果要支持更多文件格式呢？你就不得不为每种你希望支持的格式写加载器了。

另一个解决方案也许是一种更好的选择，使用一个支持多种流行格式的图像加载库来为我们解决这个问题。比如说我们要用的 `stb_image.h` 库。

stb_image.h

`stb_image.h` 是 [Sean Barrett](#) 的一个非常流行的单头文件图像加载库，它能够加载大部分流行的文件格式，并且能够很简单得整合到你的工程之中。`stb_image.h` 可以在[这里](#) 下载。下载这一个头文件，将它以 `stb_image.h` 的名字加入你的工程，并另创建一个新的C++文件，输入以下代码：

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

通过定义 `STB_IMAGE_IMPLEMENTATION`，预处理器会修改头文件，让其只包含相关的函数定义源码，等于是将这个头文件变为一个 `.cpp` 文件了。现在只需要在你的程序中包含 `stb_image.h` 并编译就可以了。

下面的教程中，我们会使用一张 [木箱](#) 的图片。要使用 `stb_image.h` 加载图片，我们需要使用它的 `stbi_load` 函数：

```
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
```

这个函数首先接受一个图像文件的位置作为输入。接下来它需要三个 `int` 作为它的第二、第三和第四个参数，`stb_image.h` 将会用图像的宽度、高度和颜色通道的个数填充这三个变量。我们之后生成纹理的时候会用到的图像的宽度和高度的。

生成纹理

和之前生成的OpenGL对象一样，纹理也是使用ID引用的。让我们来创建一个：

```
unsigned int texture;
 glGenTextures(1, &texture);
```

`glGenTextures` 函数首先需要输入生成纹理的数量，然后把它们储存在第二个参数的 `unsigned int` 数组中（我们的例子中只是单独的一个 `unsigned int`），就像其他对象一样，我们需要绑定它，让之后任何的纹理指令都可以配置当前绑定的纹理：

```
 glBindTexture(GL_TEXTURE_2D, texture);
```

现在纹理已经绑定了，我们可以使用前面载入的图片数据生成一个纹理了。纹理可以通过 `glTexImage2D` 来生成：

```
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
 glGenerateMipmap(GL_TEXTURE_2D);
```

函数很长，参数也不少，所以我们一个一个地讲解：

- 第一个参数指定了纹理目标(Target)。设置为`GL_TEXTURE_2D`意味着会生成与当前绑定的纹理对象在同一个目标上的纹理（任何绑定到`GL_TEXTURE_1D`和`GL_TEXTURE_3D`的纹理不会受到影响）。
- 第二个参数为纹理指定多级渐远纹理的级别，如果你希望单独手动设置每个多级渐远纹理的级别的话。这里我们填0，也就是基本级别。
- 第三个参数告诉OpenGL我们希望把纹理储存为哪种格式。我们的图像只有`RGB`值，因此我们也把纹理储存为`RGB`值。
- 第四个和第五个参数设置最终的纹理的宽度和高度。我们之前加载图像的时候储存了它们，所以我们使用对应的变量。
- 下个参数应该总是被设为`0`（历史遗留的问题）。
- 第七第八个参数定义了源图的格式和数据类型。我们使用RGB值加载这个图像，并把它们储存为`char`(byte)数组，我们将会传入对应值。
- 最后一个参数是真正的图像数据。

当调用`glTexImage2D`时，当前绑定的纹理对象就会被附加上纹理图像。然而，目前只有基本级别(Base-level)的纹理图像被加载了，如果要使用多级渐远纹理，我们必须手动设置所有不同的图像（不断递增第二个参数）。或者，直接在生成纹理之后调用`glGenerateMipmap`。这会为当前绑定的纹理自动生成所有需要的多级渐远纹理。

生成了纹理和相应的多级渐远纹理后，释放图像的内存是一个很好的习惯。

```
stbi_image_free(data);
```

生成一个纹理的过程应该看起来像这样：

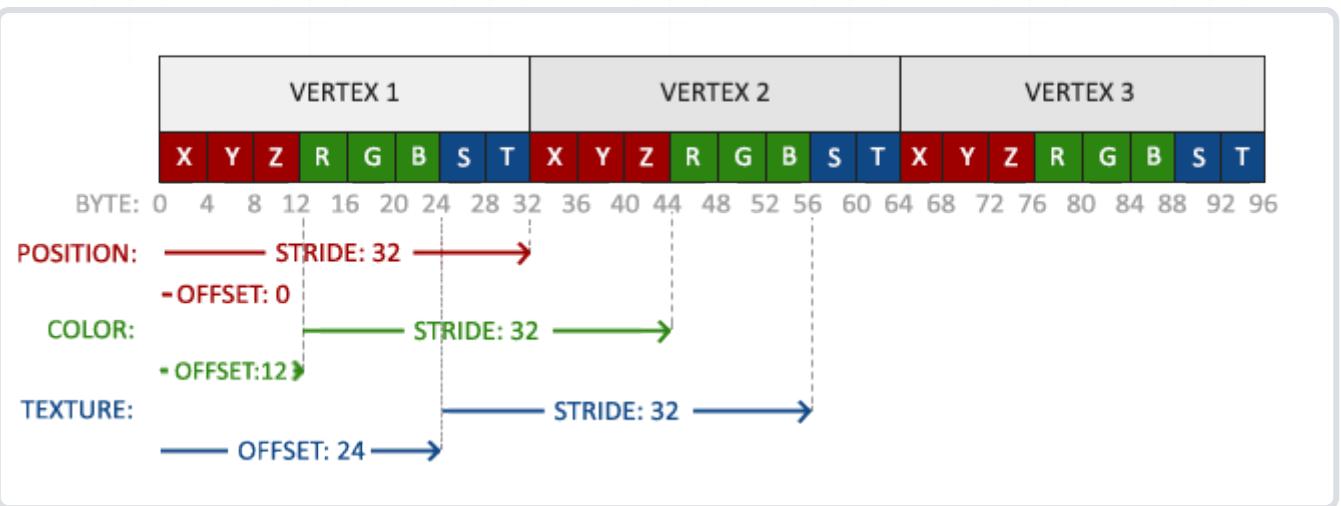
```
unsigned int texture;
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
 // 为当前绑定的纹理对象设置环绕、过滤方式
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 // 加载并生成纹理
 int width, height, nrChannels;
 unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
 if (data)
 {
     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
     glGenerateMipmap(GL_TEXTURE_2D);
 }
 else
 {
     std::cout << "Failed to load texture" << std::endl;
 }
 stbi_image_free(data);
```

应用纹理

后面的这部分我们会使用`glDrawElements`绘制「你好，三角形」教程最后一部分的矩形。我们需要告知OpenGL如何采样纹理，所以我们必须使用纹理坐标更新顶点数据：

```
float vertices[] = {
//    ---- 位置 ----- 颜色 ----- - 纹理坐标 -
    0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // 右上
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // 右下
   -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // 左下
   -0.5f, 0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, // 左上
};
```

由于我们添加了一个额外的顶点属性，我们必须告诉OpenGL我们新的顶点格式：



```
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
 glEnableVertexAttribArray(2);
```

注意，我们同样需要调整前面两个顶点属性的步长参数为`8 * sizeof(float)`。

接着我们需要调整顶点着色器使其能够接受顶点坐标作为一个顶点属性，并把坐标传给片段着色器：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
    TexCoord = aTexCoord;
}
```

片段着色器应该接下来会把输出变量`TexCoord`作为输入变量。

片段着色器也应该能访问纹理对象，但是我们怎样能把纹理对象传给片段着色器呢？GLSL有一个供纹理对象使用的内建数据类型，叫作**采样器**(Sampler)，它以纹理类型作为后缀，比如 `sampler1D`、`sampler3D`，或在我们的例子中的 `sampler2D`。我们可以简单声明一个 `uniform sampler2D` 把一个纹理添加到片段着色器中，稍后我们会把纹理赋值给这个uniform。

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

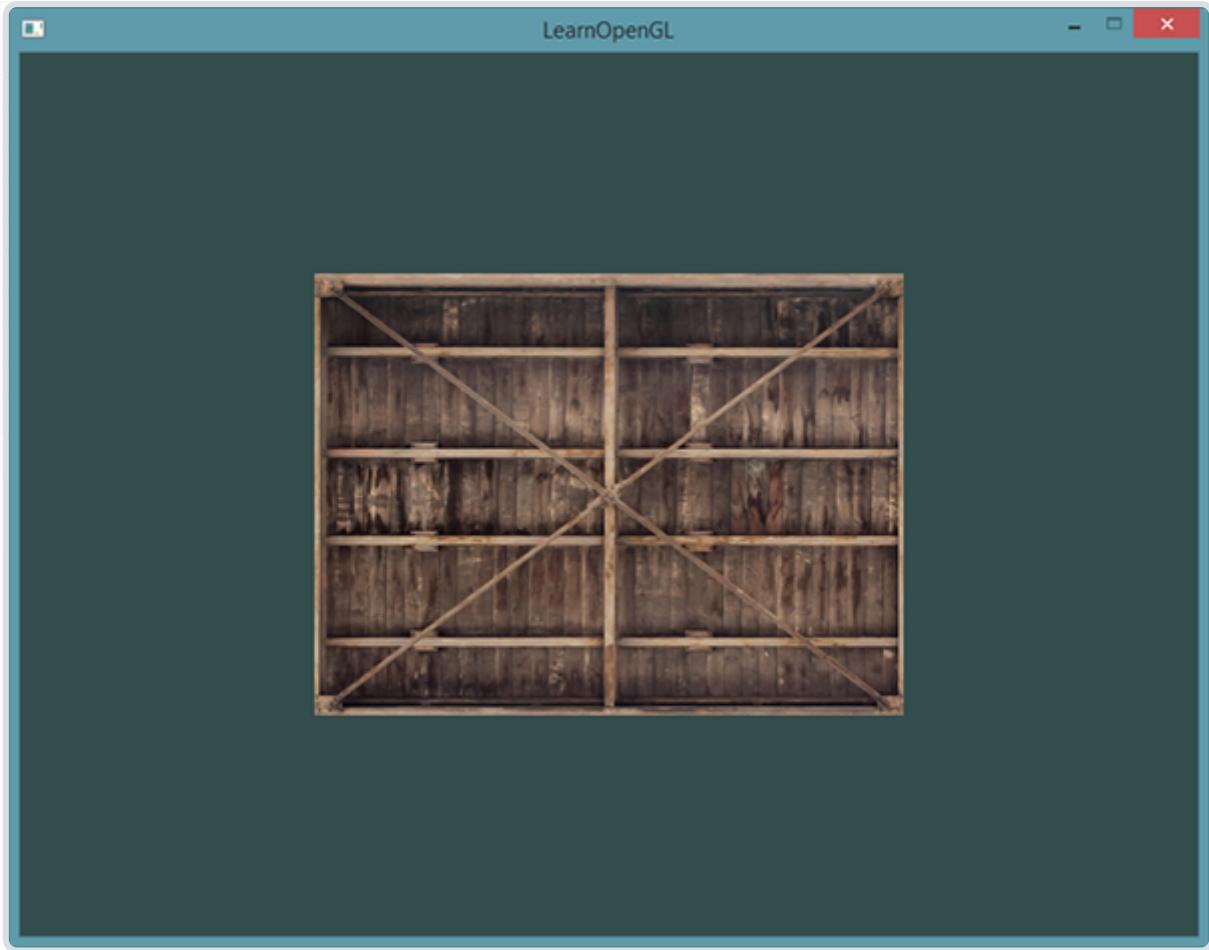
void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
```

我们使用GLSL内建的`texture`函数来采样纹理的颜色，它第一个参数是纹理采样器，第二个参数是对应的纹理坐标。`texture`函数会使用之前设置的纹理参数对相应的颜色值进行采样。这个片段着色器的输出就是纹理的（插值）纹理坐标上的(过滤后的)颜色。

现在只剩下在调用`glDrawElements`之前绑定纹理了，它会自动把纹理赋值给片段着色器的采样器：

```
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

如果你跟着这个教程正确地做完了，你会看到下面的图像：



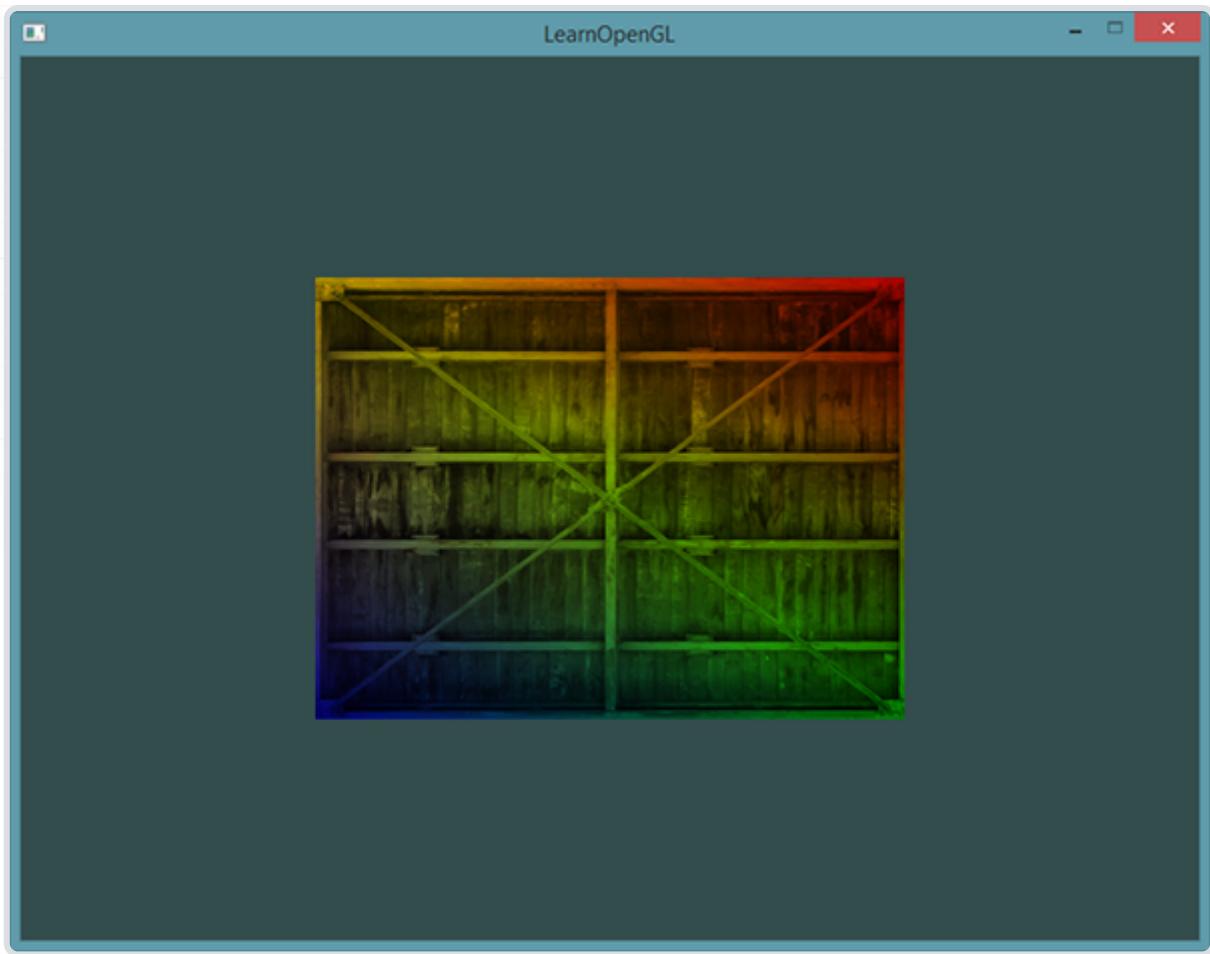
如果你的矩形是全黑或全白的你可能在哪儿做错了什么。检查你的着色器日志，并尝试对比一下[源码](#)。

如果你的纹理代码不能正常工作或者显示是全黑，请继续阅读，并一直跟进我们的代码到最后的例子，它是应该能够工作的。在一些驱动中，必须要对每个采样器uniform都附加上纹理单元才可以，这个会在下面介绍。

我们还可以把得到的纹理颜色与顶点颜色混合，来获得更有趣的效果。我们只需把纹理颜色与顶点颜色在片段着色器中相乘来混合二者颜色：

```
FragColor = texture(ourTexture, TexCoord) * vec4(ourColor, 1.0);
```

最终的效果应该是顶点颜色和纹理颜色的混合色：



我猜你会说我们的箱子喜欢跳70年代的迪斯科。

纹理单元

你可能会奇怪为什么 `sampler2D` 变量是个uniform，我们却不用 `glUniform` 给它赋值。使用 `glUniform1i`，我们可以给纹理采样器分配一个位置值，这样的话我们能够在一个片段着色器中设置多个纹理。一个纹理的位置值通常称为一个**纹理单元**(Texture Unit)。一个纹理的默认纹理单元是0，它是默认的激活纹理单元，所以教程前面部分我们没有分配一个位置值。

纹理单元的主要目的是让我们在着色器中可以使用多于一个的纹理。通过把纹理单元赋值给采样器，我们可以一次绑定多个纹理，只要我们首先激活对应的纹理单元。就像 `glBindTexture`一样，我们可以使用 `glActiveTexture` 激活纹理单元，传入我们需要使用的纹理单元：

```
glActiveTexture(GL_TEXTURE0); // 在绑定纹理之前先激活纹理单元
glBindTexture(GL_TEXTURE_2D, texture);
```

激活纹理单元之后，接下来的 `glBindTexture` 函数调用会绑定这个纹理到当前激活的纹理单元，纹理单元 `GL_TEXTURE0` 默认总是被激活，所以我们在前面的例子里当我们使用 `glBindTexture` 的时候，无需激活任何纹理单元。

OpenGL至少保证有16个纹理单元供你使用，也就是说你可以激活从`GL_TEXTURE0`到`GL_TEXTURE15`。它们都是按顺序定义的，所以我们也可以通过`GL_TEXTURE0 + 8`的方式获得`GL_TEXTURE8`，这在当我们需要循环一些纹理单元的时候会很有用。

我们仍然需要编辑片段着色器来接收另一个采样器。这应该相对来说非常直接了：

```
#version 330 core
...
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```

最终输出颜色现在是两个纹理的结合。GLSL内建的`mix`函数需要接受两个值作为参数，并对它们根据第三个参数进行线性插值。如果第三个值是`0.0`，它会返回第一个输入；如果是`1.0`，会返回第二个输入值。`0.2`会返回`80%`的第一个输入颜色和`20%`的第二个输入颜色，即返回两个纹理的混合色。

我们现在需要载入并创建另一个纹理；你应该对这些步骤很熟悉了。记得创建另一个纹理对象，载入图片，使用`glTexImage2D`生成最终纹理。对于第二个纹理我们使用一张[你学习OpenGL时的面部表情](#)图片。

为了使用第二个纹理（以及第一个），我们必须改变一点渲染流程，先绑定两个纹理到对应的纹理单元，然后定义哪个uniform采样器对应哪个纹理单元：

```
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texture1);
 glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, texture2);

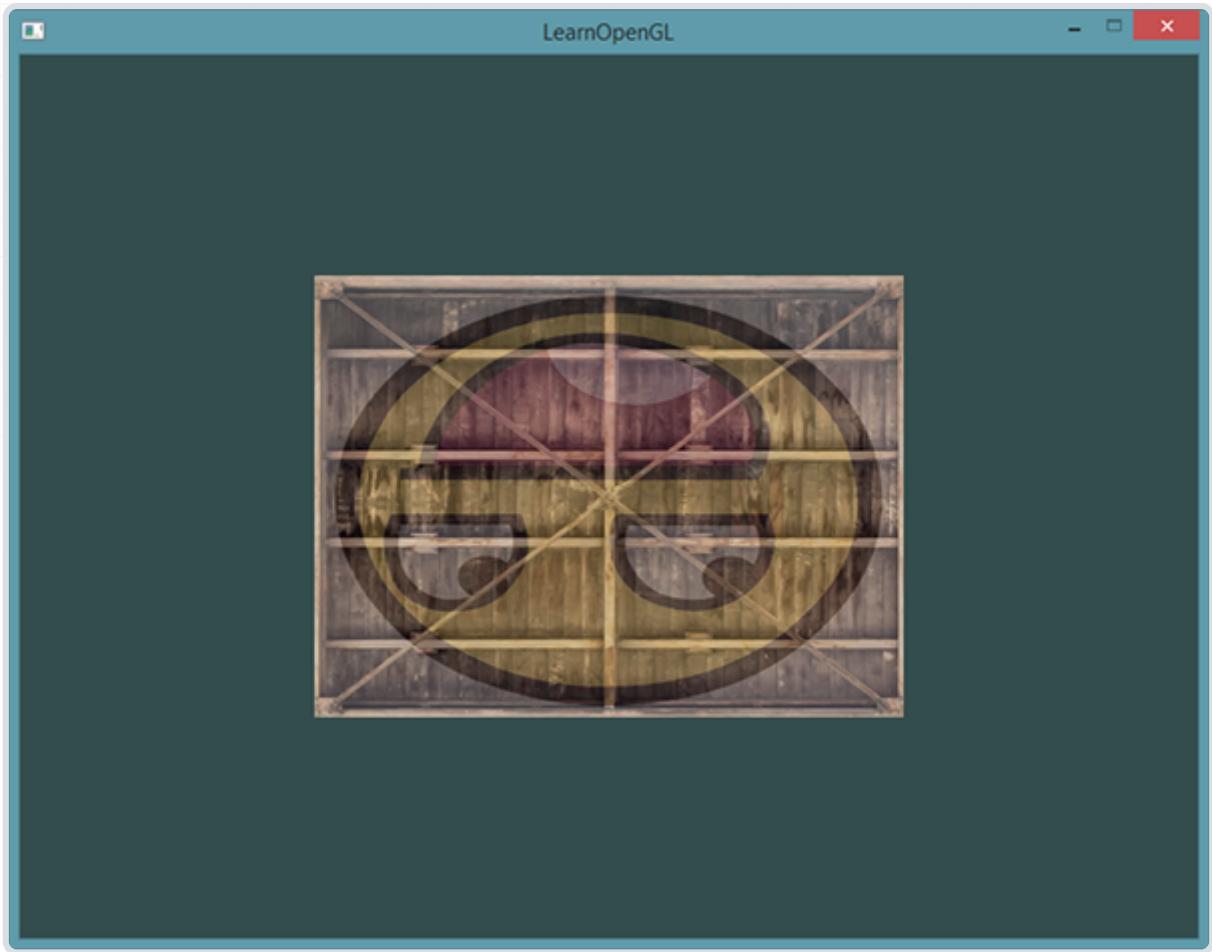
 glBindVertexArray(VAO);
 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

我们还要通过使用`glUniform1i`设置每个采样器的方式告诉OpenGL每个着色器采样器属于哪个纹理单元。我们只需要设置一次即可，所以这个会放在渲染循环的前面：

```
ourShader.use(); // 不要忘记在设置uniform变量之前激活着色器程序!
glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0); // 手动设置
ourShader.setInt("texture2", 1); // 或者使用着色器类设置

while(...)
{
    [...]
}
```

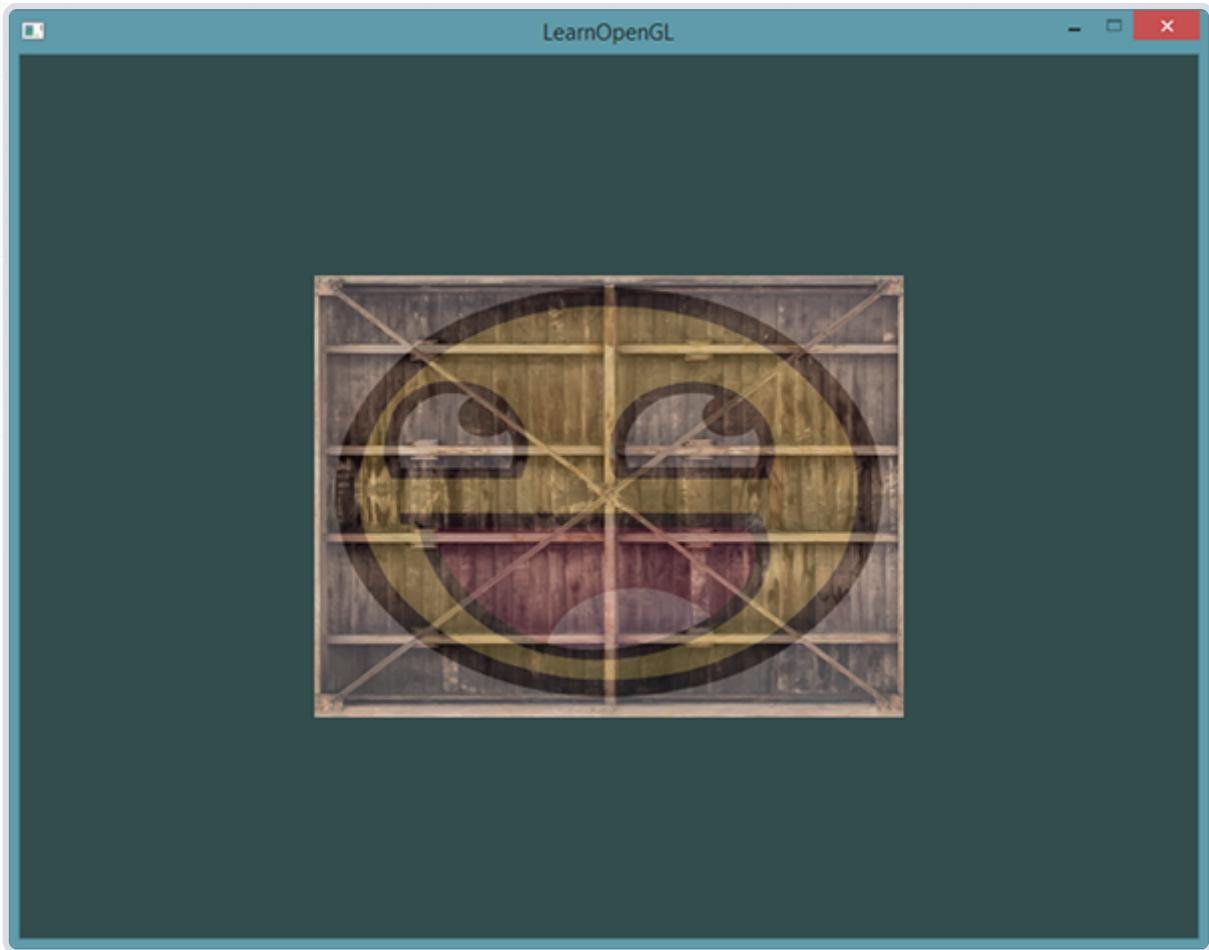
通过使用`glUniform1i`设置采样器，我们保证了每个uniform采样器对应着正确的纹理单元。你应该能得到下面的结果：



你可能注意到纹理上下颠倒了！这是因为OpenGL要求y轴`0.0`坐标是在图片的底部的，但是图片的y轴`0.0`坐标通常在顶部。很幸运，`stb_image.h`能够在图像加载时帮助我们翻转y轴，只需要在加载任何图像前加入以下语句即可：

```
stbi_set_flip_vertically_on_load(true);
```

在让`stb_image.h`在加载图片时翻转y轴之后你就应该能够获得下面的结果了：



如果你看到了一个开心的箱子，你就做对了。你可以对比一下[源代码](#)。

练习

为了更熟练地使用纹理，建议在继续之后的学习之前做完这些练习：

- 修改片段着色器，仅让笑脸图案朝另一个方向看，[参考解答](#)
- 尝试用不同的纹理环绕方式，设定一个从 `0.0f` 到 `2.0f` 范围内的（而不是原来的 `0.0f` 到 `1.0f`）纹理坐标。试试看能不能在箱子的角落放置4个笑脸：[参考解答](#), [结果](#)。记得一定要试试其它的环绕方式。
- 尝试在矩形上只显示纹理图像的中间一部分，修改纹理坐标，达到能看见单个的像素的效果。尝试使用`GL_NEAREST`的纹理过滤方式让像素显示得更清晰：[参考解答](#)
- 使用一个uniform变量作为`mix`函数的第三个参数来改变两个纹理可见度，使用上和下键来改变箱子或笑脸的可见度：[参考解答](#)。

2.2.13 变换

原文 [Transformations](#)

作者 JoeyDeVries

翻译 Django, Krasjet, [BLumia](#)

校对 暂未校对

尽管我们现在已经知道了如何创建一个物体、着色、加入纹理，给它们一些细节的表现，但因为它们都还是静态的物体，仍是不够有趣。我们可以尝试着在每一帧改变物体的顶点并且重配置缓冲区从而使它们移动，但这太繁琐了，而且会消耗很多的处理时间。我们现在有一个更好的解决方案，使用（多个）**矩阵**(Matrix)对象可以更好的**变换**(Transform)一个物体。当然，这并不是说我们会去讨论武术和数字虚拟世界（译注：Matrix同样也是电影「黑客帝国」的英文名，电影中人类生活在数字虚拟世界，主角会武术）。

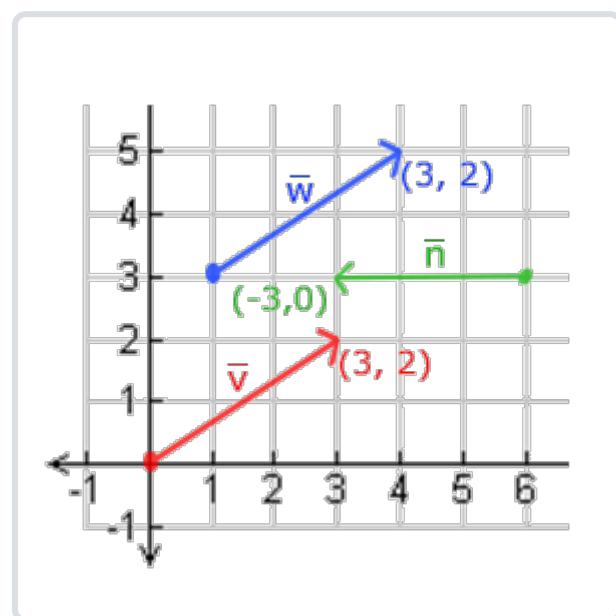
矩阵是一种非常有用的数学工具，尽管听起来可能有些吓人，不过一旦你理解了它们后，它们会变得非常有用。在讨论矩阵的过程中，我们需要使用到一些数学知识。对于一些愿意多了解这些知识的读者，我会附加一些资源给你们阅读。

为了深入了解变换，我们首先要在讨论矩阵之前进一步了解一下向量。这一节的目标是让你拥有将来需要的最基础的数学背景知识。如果你发现这节十分困难，尽量尝试去理解它们，当你以后需要它们的时候回过头来复习这些概念。

2.2.14 向量

向量最基本的定义就是一个方向。或者更正式的说，向量有一个**方向**(Direction)和**大小**(Magnitude，也叫做强度或长度)。你可以把向量想像成一个藏宝图上的指示：“向左走10步，向北走3步，然后向右走5步”；“左”就是方向，“10步”就是向量的长度。那么这个藏宝图的指示一共有3个向量。向量可以在任意维度(Dimension)上，但是我们通常只使用2至4维。如果一个向量有2个维度，它表示一个平面的方向(想象一下2D的图像)，当它有3个维度的时候它可以表达一个3D世界的方向。

下面你会看到3个向量，每个向量在2D图像中都用一个箭头(x, y)表示。我们在2D图片中展示这些向量，因为这样子会更直观一点。你可以把这些2D向量当做z坐标为0的3D向量。由于向量表示的是方向，起始于何处并不会改变它的值。下图我们可以看到向量和是相等的，尽管他们的起始点不同：



数学家喜欢在字母上面加一横表示向量，比如说。当用在公式中时它们通常是这样的：

由于向量是一个方向，所以有些时候会很难形象地将它们用位置(Position)表示出来。为了让其更为直观，我们通常设定这个方向的原点为(0, 0, 0)，然后指向一个方向，对应一个点，使其变为位置向量(Position Vector)（你也可以把起点设置为其他的点，然后说：这个向量从这个点起始指向另一个点）。比如说位置向量(3, 5)在图像中的起点会是(0, 0)，并会指向(3, 5)。我们可以使用向量在2D或3D空间中表示方向与位置。

和普通数字一样，我们也可以用向量进行多种运算（其中一些你可能已经看到过了）。

向量与标量运算

标量(Scalar)只是一个数字（或者说是仅有一个分量的向量）。当把一个向量加/减/乘/除一个标量，我们可以简单的把向量的每个分量分别进行该运算。对于加法来说会像这样：

其中的+可以是+, -, ·或÷，其中·是乘号。注意-和÷运算时不能颠倒（标量-/÷向量），因为颠倒的运算是没有定义的。

译注

注意，数学上是没有向量与标量相加这个运算的，但是很多线性代数的库都对它有支持（比如说我们用的GLM）。如果你使用过numpy的话，可以把它理解为[Broadcasting](#)。

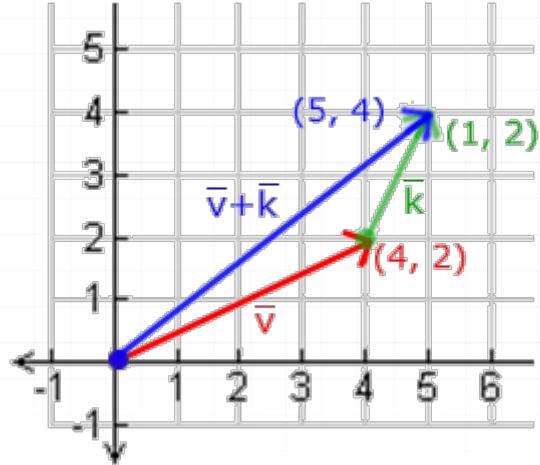
向量取反

对一个向量取反(Negate)会将其方向逆转。一个指向东北的向量取反后就指向西南方向了。我们在一个向量的每个分量前加负号就可以实现取反了（或者说用-1数乘该向量）：

向量加减

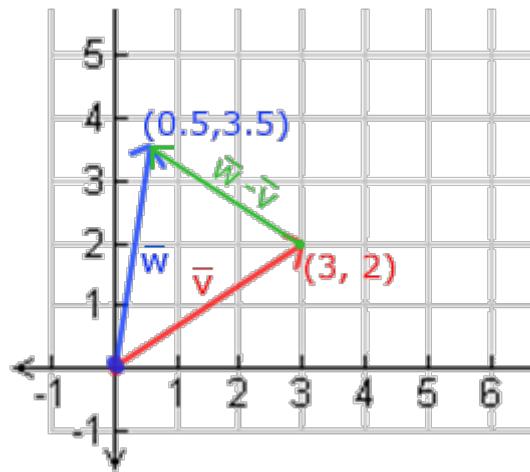
向量的加法可以被定义为是分量的(Component-wise)相加，即将一个向量中的每一个分量加上另一个向量的对应分量：

向量 $v = (4, 2)$ 和 $k = (1, 2)$ 可以直观地表示为：



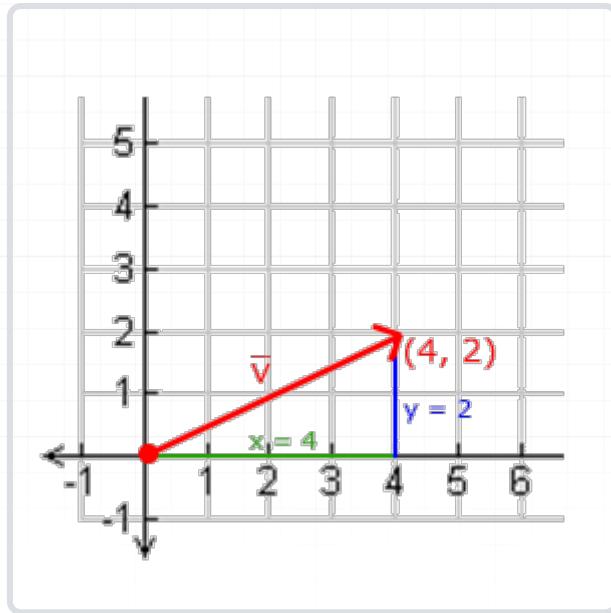
就像普通数字的加减一样，向量的减法等于加上第二个向量的相反向量：

两个向量的相减会得到这两个向量指向位置的差。这在我们想要获取两点的差会非常有用。



长度

我们使用勾股定理(Pythagoras Theorem)来获取向量的长度(Length)/大小(Magnitude)。如果你把向量的x与y分量画出来，该向量会和x与y分量为边形成一个三角形：



因为两条边 (x 和 y) 是已知的，如果希望知道斜边的长度，我们可以直接通过勾股定理来计算：

表示向量的长度，我们也可以加上把这个公式拓展到三维空间。

例子中向量(4, 2)的长度等于：

结果是4.47。

有一个特殊类型的向量叫做**单位向量**(Unit Vector)。单位向量有一个特别的性质——它的长度是1。我们可以用任意向量的每个分量除以向量的长度得到它的单位向量：

我们把这种方法叫做一个向量的**标准化**(Normalizing)。单位向量头上有一个^样子的记号。通常单位向量会变得很有用，特别是在我们只关心方向不关心长度的时候（如果改变向量的长度，它的方向并不会改变）。

向量相乘

两个向量相乘是一种很奇怪的情况。普通的乘法在向量上是没有定义的，因为它在视觉上是没有意义的。但是在相乘的时候我们有两种特定情况可以选择：一个是**点乘**(Dot Product)，记作，另一个是**叉乘**(Cross Product)，记作。

点乘

两个向量的点乘等于它们的数乘结果乘以两个向量之间夹角的余弦值。可能听起来有点费解，我们来看一下公式：

它们之间的夹角记作。为什么这很有用？想象如果和都是单位向量，它们的长度会等于1。这样公式会有效简化成：

现在点积只定义了两个向量的夹角。你也许记得90度的余弦值是0，0度的余弦值是1。使用点乘可以很容易测试两个向量是否**正交**(Orthogonal)或平行（正交意味着两个向量互为**直角**）。如果你想要了解更多关于正弦或余弦函数的知识，我推荐你看[可汗学院](#)的基础三角学视频。

你也可以通过点乘的结果计算两个非单位向量的夹角，点乘的结果除以两个向量的长度之积，得到的结果就是夹角的余弦值，即。

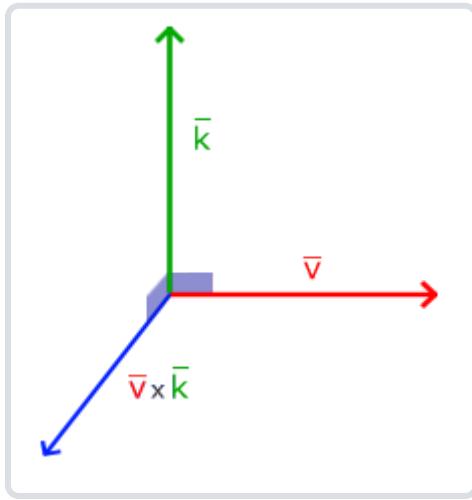
译注：通过上面点乘定义式可推出：

所以，我们该如何计算点乘呢？点乘是通过将对应分量逐个相乘，然后再把所得积相加来计算的。两个单位向量的（你可以验证它们的长度都为1）点乘会像是这样：

要计算两个单位向量间的夹角，我们可以使用反余弦函数，可得结果是143.1度。现在我们很快就计算出了这两个向量的夹角。点乘会在计算光照的时候非常有用。

叉乘

叉乘只在3D空间中有定义，它需要两个不平行向量作为输入，生成一个正交于两个输入向量的第三个向量。如果输入的两个向量也是正交的，那么叉乘之后将会产生3个互相正交的向量。接下来的教程中这会非常有用。下面的图片展示了3D空间中叉乘的样子：



不同于其他运算，如果你没有钻研过线性代数，可能会觉得叉乘很反直觉，所以只记住公式就没问题啦（记不住也没问题）。下面你会看到两个正交向量A和B叉积：

是不是看起来毫无头绪？不过只要你按照步骤来了，你就能得到一个正交于两个输入向量的第三个向量。

2.2.15 矩阵

现在我们已经讨论了向量的全部内容，是时候看看矩阵了！简单来说矩阵就是一个矩形的数字、符号或表达式数组。矩阵中每一项叫做矩阵的元素(Element)。下面是一个 2×3 矩阵的例子：

矩阵可以通过 (i, j) 进行索引， i 是行， j 是列，这就是上面的矩阵叫做 2×3 矩阵的原因（3列2行，也叫做矩阵的维度(Dimension)）。这与你在索引2D图像时的 (x, y) 相反，获取4的索引是 $(2, 1)$ （第二行，第一列）（译注：如果是图像索引应该是 $(1, 2)$ ，先算列，再算行）。

矩阵基本也就是这些了，它就是一个矩形的数学表达式阵列。和向量一样，矩阵也有非常漂亮的数学属性。矩阵有几个运算，分别是：矩阵加法、减法和乘法。

矩阵的加减

矩阵与标量之间的加减定义如下：

标量值要加到矩阵的每一个元素上。矩阵与标量的减法也相似：

译注

注意，数学上是没有矩阵与标量相加减的运算的，但是很多线性代数的库都对它有支持（比如说我们用的GLM）。如果你使用过numpy的话，可以把它理解为[Broadcasting](#)。

矩阵与矩阵之间的加减就是两个矩阵对应元素的加减运算，所以总体的规则和与标量运算是差不多的，只不过在相同索引下的元素才能进行运算。这也就是说加法和减法只对同维度的矩阵才是有定义的。一个 3×2 矩阵和一个 2×3 矩阵（或一个 3×3 矩阵与 4×4 矩阵）是不能进行加减的。我们看看两个 2×2 矩阵是怎样相加的：

同样的法则也适用于减法：

矩阵的数乘

和矩阵与标量的加减一样，矩阵与标量之间的乘法也是矩阵的每一个元素分别乘以该标量。下面的例子展示了乘法的过程：

现在我们也能明白为什么这些单独的数字要叫做标量(Scalar)了。简单来说，标量就是用它的值缩放(Scale)矩阵的所有元素（译注：注意Scalar是由Scale + -ar演变过来的）。前面那个例子中，所有的元素都被放大了2倍。

到目前为止都还好，我们的例子都不复杂。不过矩阵与矩阵的乘法就不一样了。

矩阵相乘

矩阵之间的乘法不见得有多复杂，但的确很难让人适应。矩阵乘法基本上意味着遵照规定好的法则进行相乘。当然，相乘还有一些限制：

1. 只有当左侧矩阵的列数与右侧矩阵的行数相等，两个矩阵才能相乘。
2. 矩阵相乘不遵守[交换律](#)(Commutative)，也就是说。

我们先看一个两个 2×2 矩阵相乘的例子：

现在你可能会想：天哪，刚刚到底发生了什么？矩阵的乘法是一系列乘法和加法组合的结果，它使用到了左侧矩阵的行和右侧矩阵的列。我们可以看下面的图片：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

我们首先把左侧矩阵的行和右侧矩阵的列拿出来。这些挑出来行和列将决定我们该计算结果 2×2 矩阵的哪个输出值。如果取的是左矩阵的第一行，输出值就会出现在结果矩阵的第一行。接下来再取一列，如果我们取的是右矩阵的第一列，最终值则会出现在结果矩阵的第一列。这正是红框里的情况。如果想计算结果矩阵右下角的值，我们要用第一个矩阵的第二行和第二个矩阵的第二列（译注：简单来说就是结果矩阵的元素的行取决于第一个矩阵，列取决于第二个矩阵）。

计算一项的结果值的方式是先计算左侧矩阵对应行和右侧矩阵对应列的第一个元素之积，然后是第二个，第三个，第四个等等，然后把所有的乘积相加，这就是结果了。现在我们就能解释为什么左侧矩阵的列数必须和右侧矩阵的行数相等了，如果不相等这一步的运算就无法完成了！

结果矩阵的维度是 (n, m) ， n 等于左侧矩阵的行数， m 等于右侧矩阵的列数。

如果在脑子里想象出这一乘法有些困难，别担心。不断地动手计算，如果遇到困难再回头看这页的内容。随着时间流逝，矩阵乘法对你来说会变成很自然的事。

我们用一个更大的例子来结束对矩阵相乘的讨论。试着使用颜色来寻找规律。作为一个有用的练习，你可以试着自己解答一下这个乘法问题，再将你的结果和图中的这个进行对比（如果用笔计算，你很快就能掌握它们）。

可以看到，矩阵相乘非常繁琐而容易出错（这也是我们通常让计算机做这件事的原因），而且当矩阵变大以后很快就会出现问题。如果你仍然希望了解更多，或对矩阵的数学性质感到好奇，我强烈推荐你看看[可汗学院](#)的矩阵教程。

不管怎样，现在我们知道如何进行矩阵相乘了，我们可以开始学习好东西了。

2.2.16 矩阵与向量相乘

目前为止，通过这些教程我们已经相当了解向量了。我们用向量来表示位置，表示颜色，甚至是纹理坐标。让我们更深入了解一下向量，它其实就是一个 $N \times 1$ 矩阵， N 表示向量分量的个数（也叫N维(N-dimensional)向量）。如果你仔细思考一下就会明白。向量和矩阵一样都是一个数字序列，但它只有1列。那么，这个新的定义对我们有什么帮助呢？如果我们有一个 $M \times N$ 矩阵，我们可以用这个矩阵乘以我们的 $N \times 1$ 向量，因为这个矩阵的列数等于向量的行数，所以它们就能相乘。

但是为什么我们会关心矩阵能否乘以一个向量？好吧，正巧，很多有趣的2D/3D变换都可以放在一个矩阵中，用这个矩阵乘以我们的向量将变换(Transform)这个向量。如果你仍然有些困惑，我们来看一些例子，你很快就能明白了。

单位矩阵

在OpenGL中，由于某些原因我们通常使用 4×4 的变换矩阵，而其中最重要的原因就是大部分的向量都是4分量的。我们能想到的最简单的变换矩阵就是[单位矩阵](#)(Identity Matrix)。单位矩阵是一个除了对角线以外都是0的 $N \times N$ 矩阵。在下式中可以看到，这种变换矩阵使一个向量完全不变：

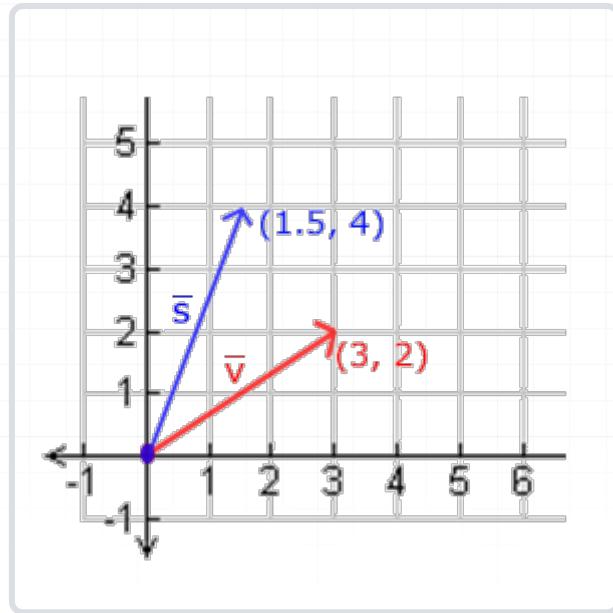
向量看起来完全没变。从乘法法则来看就很容易理解来：第一个结果元素是矩阵的第一行的每个元素乘以向量的每个对应元素。因为每行的元素除了第一个都是0，可得：，向量的其他3个元素同理。

你可能会奇怪一个没变换的变换矩阵有什么用？单位矩阵通常是生成其他变换矩阵的起点，如果我们深挖线性代数，这还是一个对证明定理、解线性方程非常有用矩阵。

缩放

对一个向量进行缩放(Scaling)就是对向量的长度进行缩放，而保持它的方向不变。由于我们进行的是2维或3维操作，我们可以分别定义一个有2或3个缩放变量的向量，每个变量缩放一个轴(x、y或z)。

我们先来尝试缩放向量。我们可以把向量沿着x轴缩放0.5，使它的宽度缩小为原来的二分之一；我们将沿着y轴把向量的高度缩放为原来的两倍。我们看看把向量缩放(0.5, 2)倍所获得的是什么样的：



记住，OpenGL通常是在3D空间进行操作的，对于2D的情况我们可以把z轴缩放1倍，这样z轴的值就不变了。我们刚刚的缩放操作是**不均匀**(Non-uniform)缩放，因为每个轴的缩放因子(Scaling Factor)都不一样。如果每个轴的缩放因子都一样那么就叫**均匀缩放**(Uniform Scale)。

我们下面会构造一个变换矩阵来为我们提供缩放功能。我们从单位矩阵了解到，每个对角线元素会分别与向量的对应元素相乘。如果我们把1变为3会怎样？这样子的话，我们就把向量的每个元素乘以3了，这事实上就把向量缩放3倍。如果我们把缩放变量表示为我们可以任意向量定义一个缩放矩阵：

注意，第四个缩放向量仍然是1，因为在3D空间中缩放w分量是无意义的。w分量另有其他用途，在后面我们会看到。

位移

位移(Translation)是在原始向量的基础上加上另一个向量从而获得一个在不同位置的新向量的过程，从而在位移向量基础上移动了原始向量。我们已经讨论了向量加法，所以这应该不会太陌生。

和缩放矩阵一样，在 4×4 矩阵上有几个特别的位置用来执行特定的操作，对于位移来说它们是第四列最上面的3个值。如果我们把位移向量表示为，我们就能把位移矩阵定义为：

这样是能工作的，因为所有的位移值都要乘以向量的w行，所以位移值会加到向量的原始值上（想想矩阵乘法法则）。而如果你用 3×3 矩阵我们的位移值就没地方放也没地方乘了，所以是不行的。

齐次坐标(Homogeneous Coordinates)

向量的w分量也叫**齐次坐标**。想要从齐次向量得到3D向量，我们可以把x、y和z坐标分别除以w坐标。我们通常不会注意这个问题，因为w分量通常是1.0。使用齐次坐标有几点好处：它允许我们在3D向量上进行位移（如果没有w分量我们是不能位移向量的），而且下一章我们会用w值创建3D视觉效果。

如果一个向量的齐次坐标是0，这个坐标就是**方向向量**(Direction Vector)，因为w坐标是0，这个向量就不能位移（译注：这也就是我们说的不能位移一个方向）。

有了位移矩阵我们就可以在3个方向(x、y、z)上移动物体，它是我们的变换工具箱中非常有用的一个变换矩阵。

旋转

上面几个的变换内容相对容易理解，在2D或3D空间中也容易表示出来，但旋转(Rotation)稍复杂些。如果你想知道旋转矩阵是如何构造出来的，我推荐你去看可汗学院[线性代数](#)的视频。

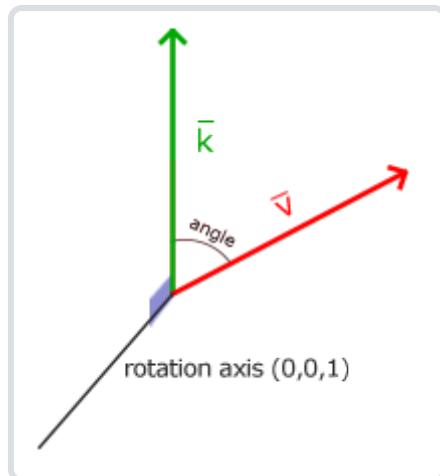
首先我们来定义一个向量的旋转到底是什么。2D或3D空间中的旋转用角(Angle)来表示。角可以是角度制或弧度制的，周角是360度或 2π 弧度。我个人更喜欢用角度，因为它们看起来更直观。

大多数旋转函数需要用弧度制的角，但幸运的是角度制的角也可以很容易地转化为弧度制的：

- 弧度转角度： $\text{角度} = \text{弧度} * (180.0f / \pi)$
- 角度转弧度： $\text{弧度} = \text{角度} * (\pi / 180.0f)$

π 约等于3.14159265359。

转半圈会旋转 $360/2 = 180$ 度，向右旋转 $1/5$ 圈表示向右旋转 $360/5 = 72$ 度。下图中展示的2D向量是由向右旋转72度所得的：



在3D空间中旋转需要定义一个角和一个**旋转轴**(Rotation Axis)。物体会沿着给定的旋转轴旋转特定角度。如果你想要更形象化的感受，可以试试向下看着一个特定的旋转轴，同时将你的头部旋转一定角度。当2D向量在3D空间中旋转时，我们把旋转轴设为z轴（尝试想象这种情况）。

使用三角学，给定一个角度，可以把一个向量变换为一个经过旋转的新向量。这通常是使用一系列正弦和余弦函数（一般简称sin和cos）各种巧妙的组合得到的。当然，讨论如何生成变换矩阵超出了这个教程的范围。

旋转矩阵在3D空间中每个单位轴都有不同定义，旋转角度用表示：

沿x轴旋转：

沿y轴旋转：

沿z轴旋转：

利用旋转矩阵我们可以把任意位置向量沿一个单位旋转轴进行旋转。也可以将多个矩阵复合，比如先沿着x轴旋转再沿着y轴旋转。但是这会很快导致一个问题——**万向节死锁**（Gimbal Lock，可以看看[这个视频（优酷）](#)来了解）。在这里我们不会讨论它的细节，但是对于3D空间中的旋转，一个更好的模型是沿着任意的一个轴，比如单位向量\$(0.662, 0.2, 0.7222)\$旋转，而不是对一系列旋转矩阵进行复合。这样的一个（超级麻烦的）矩阵是存在的，见下面这个公式，其中代表任意旋转轴：

在数学上讨论如何生成这样的矩阵仍然超出了本节内容。但是记住，即使这样一个矩阵也不能完全解决万向节死锁问题（尽管会极大地避免）。避免万向节死锁的真正解决方案是使用**四元数**(Quaternion)，它不仅更安全，而且计算会更有效率。四元数可能会在后面的教程中讨论。

译注

对四元数的理解会用到非常多的数学知识。如果你想了解四元数与3D旋转之间的关系，可以来阅读我的[教程](#)。如果你对万向节死锁的概念仍不是那么清楚，可以来阅读我教程的[Bonus章节](#)。

现在3Blue1Brown也已经开始了一个四元数的视频系列，他采用球极平面投影(Stereographic Projection)的方式将四元数投影到3D空间，同样有助于理解四元数的概念（仍在更新中）：<https://www.youtube.com/watch?v=d4EgbgTm0Bg>

矩阵的组合

使用矩阵进行变换的真正力量在于，根据矩阵之间的乘法，我们可以把多个变换组合到一个矩阵中。让我们看看我们是否能生成一个变换矩阵，让它组合多个变换。假设我们有一个顶点\$(x, y, z)\$，我们希望将其缩放2倍，然后位移\$(1, 2, 3)\$个单位。我们需要一个位移和缩放矩阵来完成这些变换。结果的变换矩阵看起来像这样：

注意，当矩阵相乘时我们先写位移再写缩放变换的。矩阵乘法是不遵守交换律的，这意味着它们的顺序很重要。当矩阵相乘时，在最右边的矩阵是第一个与向量相乘的，所以你应该从右向左读这个乘法。建议您在组合矩阵时，先进行缩放操作，然后是旋转，最后才是位移，否则它们会（消极地）互相影响。比如，如果你先位移再缩放，位移的向量也会同样被缩放（译注：比如向某方向移动2米，2米也许会被缩放成1米）！

用最终的变换矩阵左乘我们的向量会得到以下结果：

不错！向量先缩放2倍，然后位移了\$(1, 2, 3)\$个单位。

2.2.17 实践

现在我们已经解释了变换背后的所有理论，是时候将这些知识利用起来了。OpenGL没有自带任何的矩阵和向量知识，所以我们必须定义自己的数学类和函数。在教程中我们更希望抽象所有的数学细节，使用已经做好了的数学库。幸运的是，有个易于使用，专门为OpenGL量身定做的数学库，那就是GLM。

GLM

GLM是OpenGL Mathematics的缩写，它是一个只有头文件的库，也就是说我们只需包含对应的头文件就行了，不用链接和编译。GLM可以在它们的[网站](#)上下载。把头文件的根目录复制到你的includes文件夹，然后你就可以使用这个库了。



GLM库从0.9.9版本起，默认会将矩阵类型初始化为一个零矩阵（所有元素均为0），而不是单位矩阵（对角元素为1，其它元素为0）。如果你使用的是0.9.9或0.9.9以上的版本，你需要将所有的矩阵初始化改为

`glm::mat4 mat = glm::mat4(1.0f)`。如果你想与本教程的代码保持一致，请使用低于0.9.9版本的GLM，或者改用上述代码初始化所有的矩阵。

我们需要的GLM的大多数功能都可以从下面这3个头文件中找到：

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

我们来看看是否可以利用我们刚学的变换知识把一个向量 $(1, 0, 0)$ 位移 $(1, 1, 0)$ 个单位（注意，我们把它定义为一个`glm::vec4`类型的值，齐次坐标设定为1.0）：

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
// 译注：下面就是矩阵初始化的一个例子，如果使用的是0.9.9及以上版本
// 下面这行代码就需要改为：
// glm::mat4 trans = glm::mat4(1.0f)
// 之后将不再进行提示
glm::mat4 trans;
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

我们先用GLM内建的向量类定义一个叫做`vec`的向量。接下来定义一个`mat4`类型的`trans`，默认是一个 4×4 单位矩阵。下一步是创建一个变换矩阵，我们是把单位矩阵和一个位移向量传递给`glm::translate`函数来完成这个工作的（然后用给定的矩阵乘以位移矩阵就能获得最后需要的矩阵）。之后我们把向量乘以位移矩阵并且输出最后的结果。如果你仍记得位移矩阵是如何工作的话，得到的向量应该是 $(1 + 1, 0 + 1, 0 + 0)$ ，也就是 $(2, 1, 0)$ 。这个代码片段将会输出`210`，所以这个位移矩阵是正确的。

我们来做些更有意思的事情，让我们来旋转和缩放之前教程中的那个箱子。首先我们把箱子逆时针旋转90度。然后缩放0.5倍，使它变成原来的一半大。我们先来创建变换矩阵：

```
glm::mat4 trans;
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

首先，我们把箱子在每个轴都缩放到0.5倍，然后沿z轴旋转90度。GLM希望它的角度是弧度制的(Radian)，所以我们使用`glm::radians`将角度转化为弧度。注意有纹理的那面矩形是在XY平面上的，所以我们需要把它绕着z轴旋转。因为我们把这个矩阵传递给了GLM的每个函数，GLM会自动将矩阵相乘，返回的结果是一个包括了多个变换的变换矩阵。

下一个大问题是：如何把矩阵传递给着色器？我们在前面简单提到过GLSL里也有一个`mat4`类型。所以我们将修改顶点着色器让其接收一个`mat4`的uniform变量，然后再用矩阵uniform乘以位置向量：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, 1.0 - aTexCoord.y);
}
```

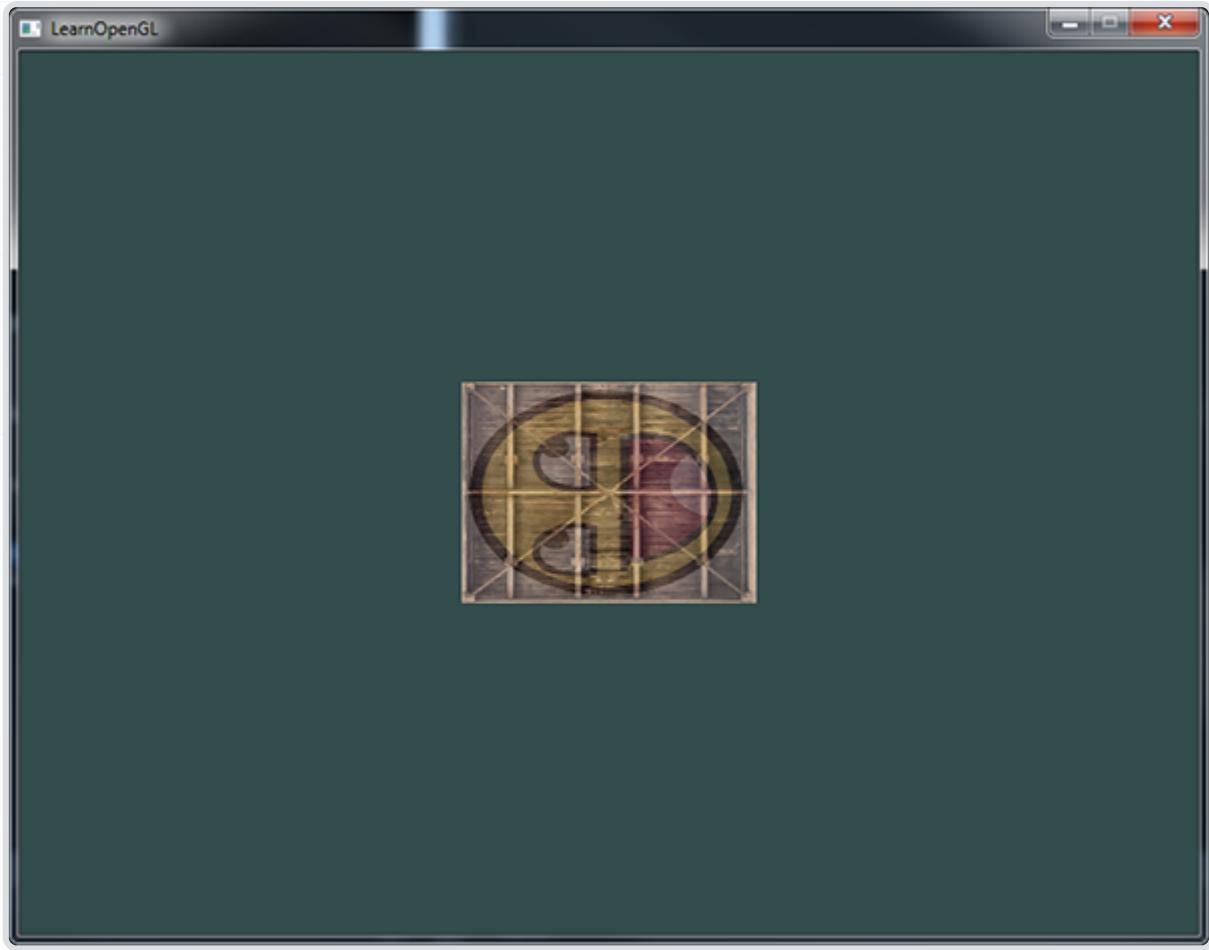
GLSL也有`mat2`和`mat3`类型从而允许了像向量一样的混合运算。前面提到的所有数学运算（像是标量-矩阵相乘，矩阵-向量相乘和矩阵-矩阵相乘）在矩阵类型里都可以使用。当出现特殊的矩阵运算的时候我们会特别说明。

在把位置向量传给`gl_Position`之前，我们先添加一个uniform，并且将其与变换矩阵相乘。我们的箱子现在应该是原来的二分之一大小并（向左）旋转了90度。当然，我们仍需要把变换矩阵传递给着色器：

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

我们首先查询uniform变量的地址，然后用有`Matrix4fv`后缀的`glUniform`函数把矩阵数据发送给着色器。第一个参数你现在应该很熟悉了，它是uniform的位置值。第二个参数告诉OpenGL我们将要发送多少个矩阵，这里是1。第三个参数询问我们是否希望对我们的矩阵进行转置(Transpose)，也就是说交换我们矩阵的行和列。OpenGL开发者通常使用一种内部矩阵布局，叫做列主序(Column-major Ordering)布局。GLM的默认布局就是列主序，所以并不需要转置矩阵，我们填`GL_FALSE`。最后一个参数是真正的矩阵数据，但是GLM并不是把它们的矩阵储存为OpenGL所希望接受的那种，因此我们要先用GLM的自带的函数`value_ptr`来变换这些数据。

我们创建了一个变换矩阵，在顶点着色器中声明了一个uniform，并把矩阵发送给了着色器，着色器会变换我们的顶点坐标。最后的结果应该看起来像这样：



完美！我们的箱子向左侧旋转，并是原来的一半大小，所以变换成功了。我们现在做些更有意思的，看看我们是否可以让箱子随着时间旋转，我们还会重新把箱子放在窗口的右下角。要让箱子随着时间推移旋转，我们必须在游戏循环中更新变换矩阵，因为它在每一次渲染迭代中都要更新。我们使用GLFW的时间函数来获取不同时间的角度：

```
glm::mat4 trans;
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

要记住的是前面的例子中我们可以在任何地方声明变换矩阵，但是现在我们必须在每一次迭代中创建它，从而保证我们能够不断更新旋转角度。这也就意味着我们不得不在每次游戏循环的迭代中重新创建变换矩阵。通常在渲染场景的时候，我们也会有多个需要在每次渲染迭代中都用新值重新创建的变换矩阵

在这里我们先把箱子围绕原点(0, 0, 0)旋转，之后，我们把旋转过后的箱子位移到屏幕的右下角。记住，实际的变换顺序应该与阅读顺序相反：尽管在代码中我们先位移再旋转，实际的变换却是先应用旋转再是位移的。明白所有这些变换的组合，并且知道它们是如何应用到物体上是一件非常困难的事情。只有不断地尝试和实验这些变换你才能快速地掌握它们。

如果你做对了，你将看到下面的结果：

这就是我们刚刚做到的！一个位移过的箱子，它会一直转，一个变换矩阵就做到了！现在你可以明白为什么矩阵在图形领域是一个如此重要的工具了。我们可以定义无限数量的变换，而把它们组合为仅仅一个矩阵，如果愿意的话我们还可以重复使用它。在着色器中使用矩阵可以省去重新定义顶点数据的功夫，它也能够节省处理时间，因为我们没有一直重新发送我们的数据（这是个非常慢的过程）。

如果你没有得到正确的结果，或者你有哪儿不清楚的地方。可以看[源码](#)。

下一节中，我们会讨论怎样使用矩阵为顶点定义不同的坐标空间。这将是我们进入实时3D图像的第一步！

拓展阅读

- [线性代数的本质](#)：Grant Sanderson制作的非常棒的视频教程系列，它讨论了变换和线性代数内在的数学本质（[中文字幕版本](#)）。

练习

- 使用应用在箱子上的最后一个变换，尝试将其改变为先旋转，后位移。看看发生了什么，试着想想为什么会发生这样的事情：
[参考解答](#)
- 尝试再次调用 `glDrawElements` 画出第二个箱子，只使用变换将其摆放在不同的位置。让这个箱子被摆放在窗口的左上角，并且会不断的缩放（而不是旋转）。（`sin` 函数在这里会很有用，不过注意使用 `sin` 函数时应用负值会导致物体被翻转）：
[参考解答](#)

2.2.18 坐标系统

原文 [Coordinate Systems](#)

作者 JoeyDeVries

翻译 linkoln, Geequlim, Krasjet, [BLumia](#)

校对 暂未校对

在上一个教程中，我们学习了如何有效地利用矩阵的变换来对所有顶点进行变换。OpenGL希望在每次顶点着色器运行后，我们可见的所有顶点都为标准化设备坐标(Normalized Device Coordinate, NDC)。也就是说，每个顶点的x, y, z坐标都应该在-1.0到1.0之间，超出这个坐标范围的顶点都将不可见。我们通常会自己设定一个坐标的范围，之后再在顶点着色器中将这些坐标变换为标准化设备坐标。然后将这些标准化设备坐标传入光栅器(Rasterizer)，将它们变换为屏幕上的二维坐标或像素。

将坐标变换为标准化设备坐标，接着再转化为屏幕坐标的过程通常是分步进行的，也就是类似于流水线那样子。在流水线中，物体的顶点在最终转化为屏幕坐标之前还会被变换到多个坐标系统(Coordinate System)。将物体的坐标变换到几个过渡坐标系(Intermediate Coordinate System)的优点在于，在这些特定的坐标系统中，一些操作或运算更加方便和容易，这一点很快就会变得很明显。对我们来说比较重要的总共有5个不同的坐标系统：

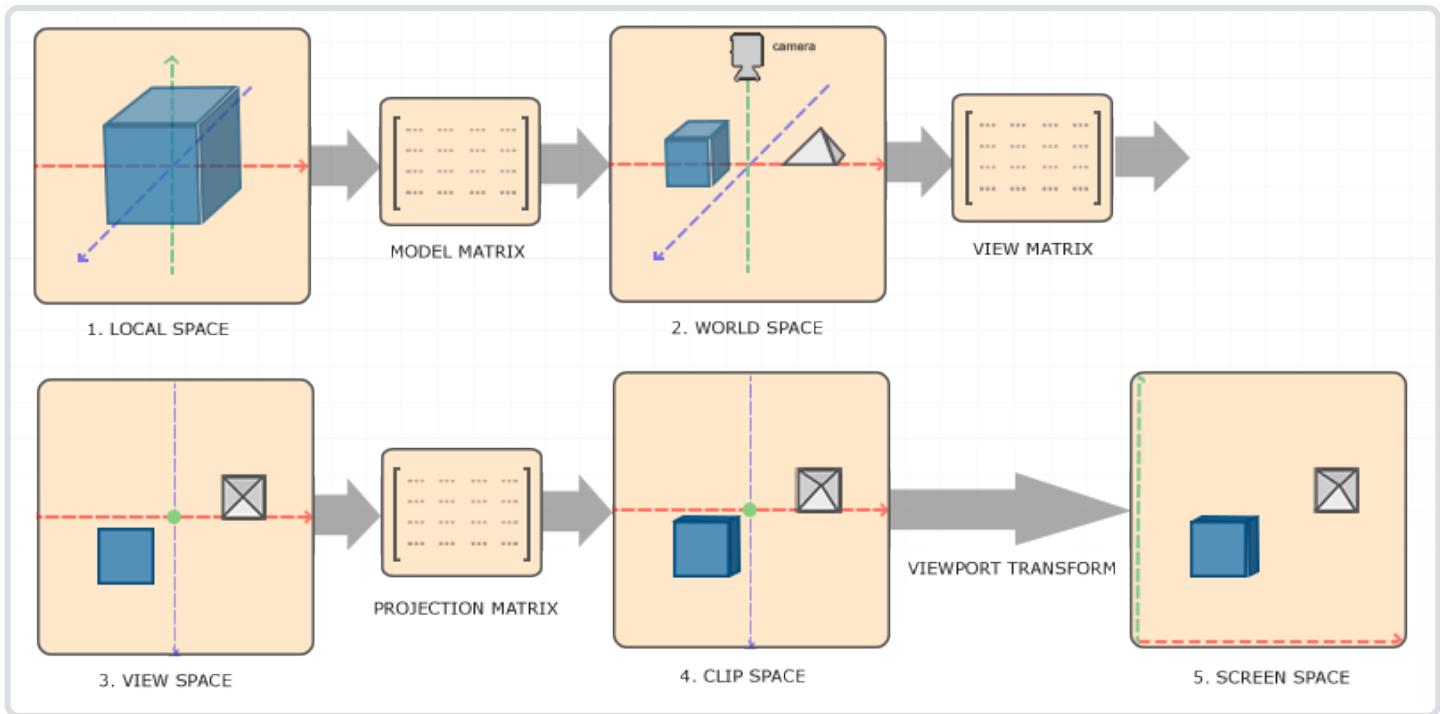
- 局部空间(Local Space，或者称为物体空间(Object Space))
- 世界空间(World Space)
- 观察空间(View Space，或者称为视觉空间(Eye Space))
- 裁剪空间(Clip Space)
- 屏幕空间(Screen Space)

这就是一个顶点在最终被转化为片段之前需要经历的所有不同状态。

你现在可能会对什么是坐标空间，什么是坐标系统感到非常困惑，所以我们将用一种更加通俗的方式来解释它们。下面，我们将显示一个整体的图片，之后我们会讲解每个空间的具体功能。

概述

为了将坐标从一个坐标系变换到另一个坐标系，我们需要用到几个变换矩阵，最重要的几个分别是模型(Model)、观察(View)、投影(Projection)三个矩阵。我们的顶点坐标起始于局部空间(Local Space)，在这里它称为局部坐标(Local Coordinate)，它在之后会变为世界坐标(World Coordinate)，观察坐标(View Coordinate)，裁剪坐标(Clip Coordinate)，并最后以屏幕坐标(Screen Coordinate)的形式结束。下面的这张图展示了整个流程以及各个变换过程做了什么：



1. 局部坐标是对象相对于局部原点的坐标，也是物体起始的坐标。
2. 下一步是将局部坐标变换为世界空间坐标，世界空间坐标是处于一个更大的空间范围的。这些坐标相对于世界的全局原点，它们会和其它物体一起相对于世界的原点进行摆放。
3. 接下来我们将世界坐标变换为观察空间坐标，使得每个坐标都是从摄像机或者说观察者的角度进行观察的。
4. 坐标到达观察空间之后，我们需要将其投影到裁剪坐标。裁剪坐标会被处理至-1.0到1.0的范围内，并判断哪些顶点将会出现在屏幕上。
5. 最后，我们将裁剪坐标变换为屏幕坐标，我们将使用一个叫做**视口变换**(Viewport Transform)的过程。视口变换将位于-1.0到1.0范围的坐标变换到由**glViewport**函数所定义的坐标范围内。最后变换出来的坐标将会送到光栅器，将其转化为片段。

你可能已经大致了解了每个坐标空间的作用。我们之所以将顶点变换到各个不同的空间的原因是有些操作在特定的坐标系统中才有意义且更方便。例如，当需要对物体进行修改的时候，在局部空间中来操作会更说得通；如果要对一个物体做出一个相对于其它物体位置的操作时，在世界坐标系中来做这个才更说得通，等等。如果我们愿意，我们也可以定义一个直接从局部空间变换到裁剪空间的变换矩阵，但那样会失去很多灵活性。

接下来我们将要更仔细地讨论各个坐标系统。

局部空间

局部空间是指物体所在的坐标空间，即对象最开始所在的地方。想象你在一个建模软件（比如说Blender）中创建了一个立方体。你创建的立方体的原点有可能位于(0, 0, 0)，即便它有可能最后在程序中处于完全不同的位置。甚至有可能你创建的所有模型都以(0, 0, 0)为初始位置（译注：然而它们会最终出现在世界的不同位置）。所以，你的模型的所有顶点都是在局部空间中：它们相对于你的物体来说都是局部的。

我们一直使用的那个箱子的顶点是被设定在-0.5到0.5的坐标范围内，(0, 0)是它的原点。这些都是局部坐标。

世界空间

如果我们将我们所有的物体导入到程序当中，它们有可能会全挤在世界的原点(0, 0, 0)上，这并不是我们想要的结果。我们想为每一个物体定义一个位置，从而能在更大的世界当中放置它们。世界空间中的坐标正如其名：是指顶点相对于（游戏）世界的坐标。如果你希望将物体分散在世界上摆放（特别是非常真实的那样），这就是你希望物体变换到的空间。物体的坐标将会从局部变换到世界空间；该变换是由模型矩阵(Model Matrix)实现的。

模型矩阵是一种变换矩阵，它能通过对物体进行位移、缩放、旋转来将它置于它本应该在的位置或朝向。你可以将它想像为变换一个房子，你需要先将它缩小（它在局部空间中太大了），并将其位移至郊区的一个小镇，然后在y轴上往左旋转一点以搭配附近的房子。你也可以把上一节将箱子到处摆放在场景中用的那个矩阵大致看作一个模型矩阵；我们将箱子的局部坐标变换到场景/世界中的不同位置。

观察空间

观察空间经常被人们称之为OpenGL的摄像机(Camera)（所以有时也称为摄像机空间(Camera Space)或视觉空间(Eye Space)）。观察空间是将世界空间坐标转化为用户视野前方的坐标而产生的结果。因此观察空间就是从摄像机的视角所观察到的空间。而这通常是由一系列的位移和旋转的组合来完成，平移/旋转场景从而使得特定的对象被变换到摄像机的前方。这些组合在一起的变换通常存储在一个观察矩阵(View Matrix)里，它被用来将世界坐标变换到观察空间。在下一节中我们将深入讨论如何创建一个这样的观察矩阵来模拟一个摄像机。

裁剪空间

在一个顶点着色器运行的最后，OpenGL期望所有的坐标都能落在一个特定的范围内，且任何在这个范围之外的点都应该被裁剪掉(Clipped)。被裁剪掉的坐标就会被忽略，所以剩下的坐标就将变为屏幕上可见的片段。这也就是裁剪空间(Clip Space)名字的由来。

因为将所有可见的坐标都指定在-1.0到1.0的范围内不是很直观，所以我们会指定自己的坐标集(Coordinate Set)并将它变换回标准化设备坐标系，就像OpenGL期望的那样。

为了将顶点坐标从观察变换到裁剪空间，我们需要定义一个投影矩阵(Projection Matrix)，它指定了一个范围的坐标，比如在每个维度上的-1000到1000。投影矩阵接着会将在这个指定的范围内的坐标变换为标准化设备坐标的范围(-1.0, 1.0)。所有在范围外的坐标不会被映射到在-1.0到1.0的范围之间，所以会被裁剪掉。在上面这个投影矩阵所指定的范围内，坐标(1250, 500, 750)将是不可见的，这是由于它的x坐标超出了范围，它被转化为一个大于1.0的标准化设备坐标，所以被裁剪掉了。

如果只是图元(Primitive)，例如三角形，的一部分超出了裁剪体积(Clipping Volume)，则OpenGL会重新构建这个三角形为一个或多个三角形让其能够适合这个裁剪范围。

由投影矩阵创建的观察箱(Viewing Box)被称为平截头体(Frustum)，每个出现在平截头体范围内的坐标都会最终出现在用户的屏幕上。将特定范围内的坐标转化到标准化设备坐标系的过程（而且它很容易被映射到2D观察空间坐标）被称之为投影(Projection)，因为使用投影矩阵能将3D坐标投影(Project)到很容易映射到2D的标准化设备坐标系中。

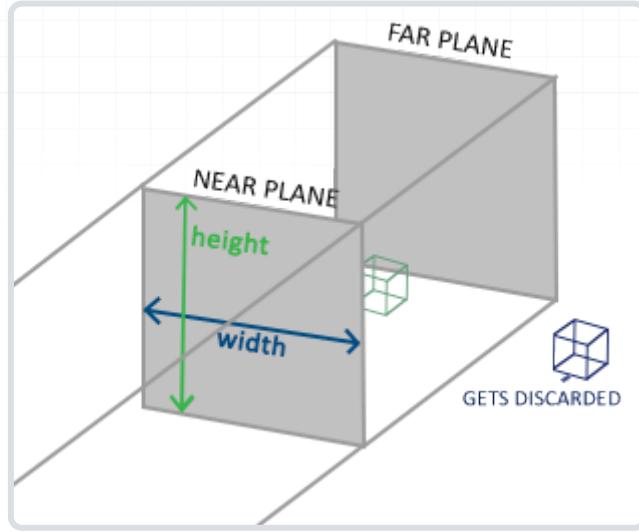
一旦所有顶点被变换到裁剪空间，最终的操作——透视除法(Perspective Division)将会执行，在这个过程中我们将位置向量的x，y，z分量分别除以向量的齐次w分量；透视除法是将4D裁剪空间坐标变换为3D标准化设备坐标的过程。这一步会在每一个顶点着色器运行的最后被自动执行。

在这一阶段之后，最终的坐标将会被映射到屏幕空间中（使用`glViewport`中的设定），并被转换成片段。

将观察坐标变换为裁剪坐标的投影矩阵可以为两种不同的形式，每种形式都定义了不同的平截头体。我们可以选择创建一个正射投影矩阵(Orthographic Projection Matrix)或一个透视投影矩阵(Perspective Projection Matrix)。

正射投影

正射投影矩阵定义了一个类似立方体的平截头箱，它定义了一个裁剪空间，在这空间之外的顶点都会被裁剪掉。创建一个正射投影矩阵需要指定可见平截头体的宽、高和长度。在使用正射投影矩阵变换至裁剪空间之后处于这个平截头体内的所有坐标将不会被裁剪掉。它的平截头体看起来像一个容器：



上面的平截头体定义了可见的坐标，它由由宽、高、**近**(Near)平面和**远**(Far)平面所指定。任何出现在近平面之前或远平面之后的坐标都会被裁剪掉。正射平截头体直接将平截头体内部的所有坐标映射为标准化设备坐标，因为每个向量的w分量都没有进行改变；如果w分量等于1.0，透视除法则不会改变这个坐标。

要创建一个正射投影矩阵，我们可以使用GLM的内置函数 `glm::ortho`：

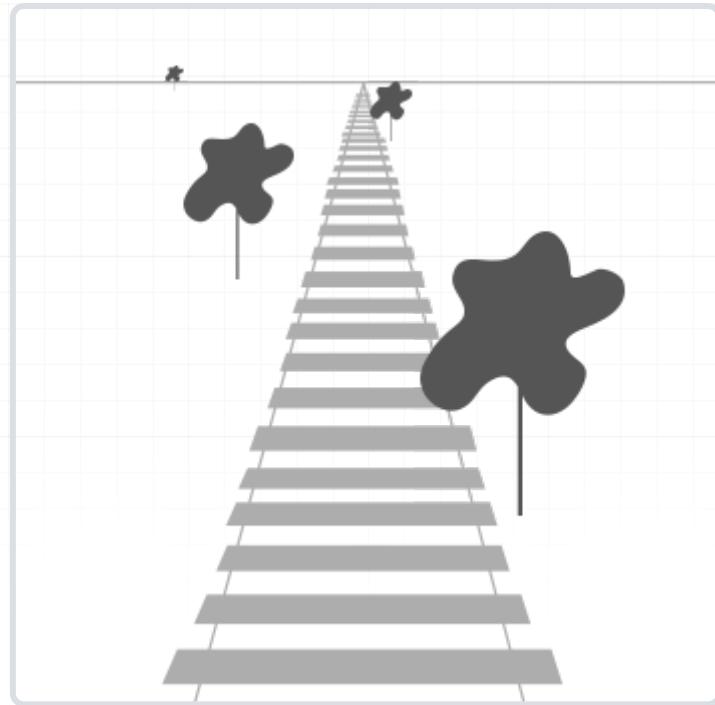
```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

前两个参数指定了平截头体的左右坐标，第三和第四个参数指定了平截头体的底部和顶部。通过这四个参数我们定义了近平面和远平面的大小，然后第五和第六个参数则定义了近平面和远平面的距离。这个投影矩阵会将处于这些x，y，z值范围内的坐标变换为标准化设备坐标。

正射投影矩阵直接将坐标映射到2D平面中，即你的屏幕，但实际上一个直接的投影矩阵会产生不真实的结果，因为这个投影没有将**透视**(Perspective)考虑进去。所以我们需要**透视投影矩阵**来解决这个问题。

透视投影

如果你曾经体验过实际生活给你带来的景象，你就会注意到离你越远的东西看起来更小。这个奇怪的效果称之为**透视**(Perspective)。透视的效果在我们看一条无限长的高速公路或铁路时尤其明显，正如下面图片显示的那样：



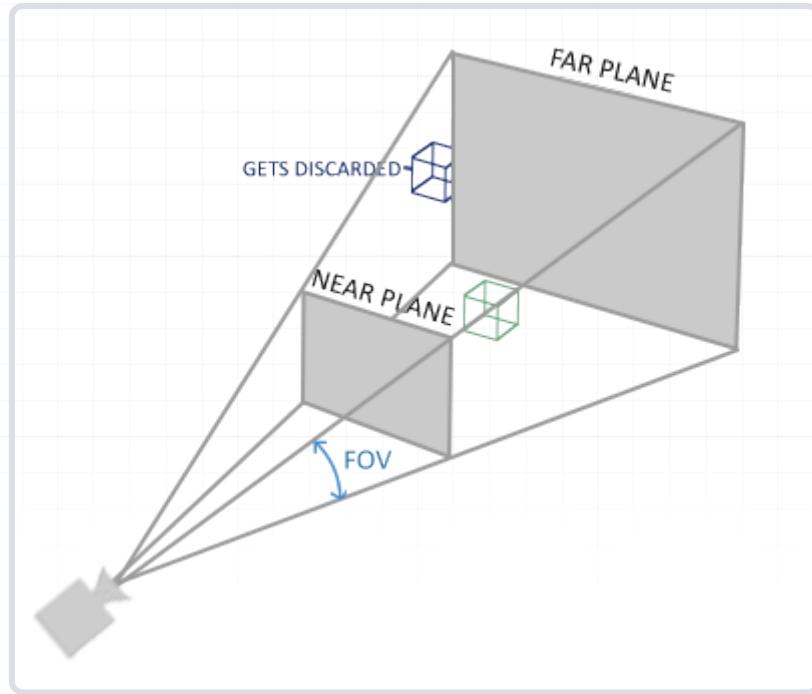
正如你看到的那样，由于透视，这两条线在很远的地方看起来会相交。这正是透视投影想要模仿的效果，它是使用[透视投影矩阵](#)来完成的。这个投影矩阵将给定的平截头体范围映射到裁剪空间，除此之外还修改了每个顶点坐标的w值，从而使得离观察者越远的顶点坐标w分量越大。被变换到裁剪空间的坐标都会在- w 到 w 的范围之间（任何大于这个范围的坐标都会被裁剪掉）。OpenGL要求所有可见的坐标都落在-1.0到1.0范围内，作为顶点着色器最后的输出，因此，一旦坐标在裁剪空间内之后，透视除法就会被应用到裁剪空间坐标上：

顶点坐标的每个分量都会除以它的w分量，距离观察者越远顶点坐标就会越小。这是也是w分量非常重要的另一个原因，它能够帮助我们进行透视投影。最后的结果坐标就是处于标准化设备空间中的。如果你对正射投影矩阵和透视投影矩阵是如何计算的很感兴趣（且不会对数学感到恐惧的话）我推荐这篇由Songho写的[文章](#)。

在GLM中可以这样创建一个透视投影矩阵：

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

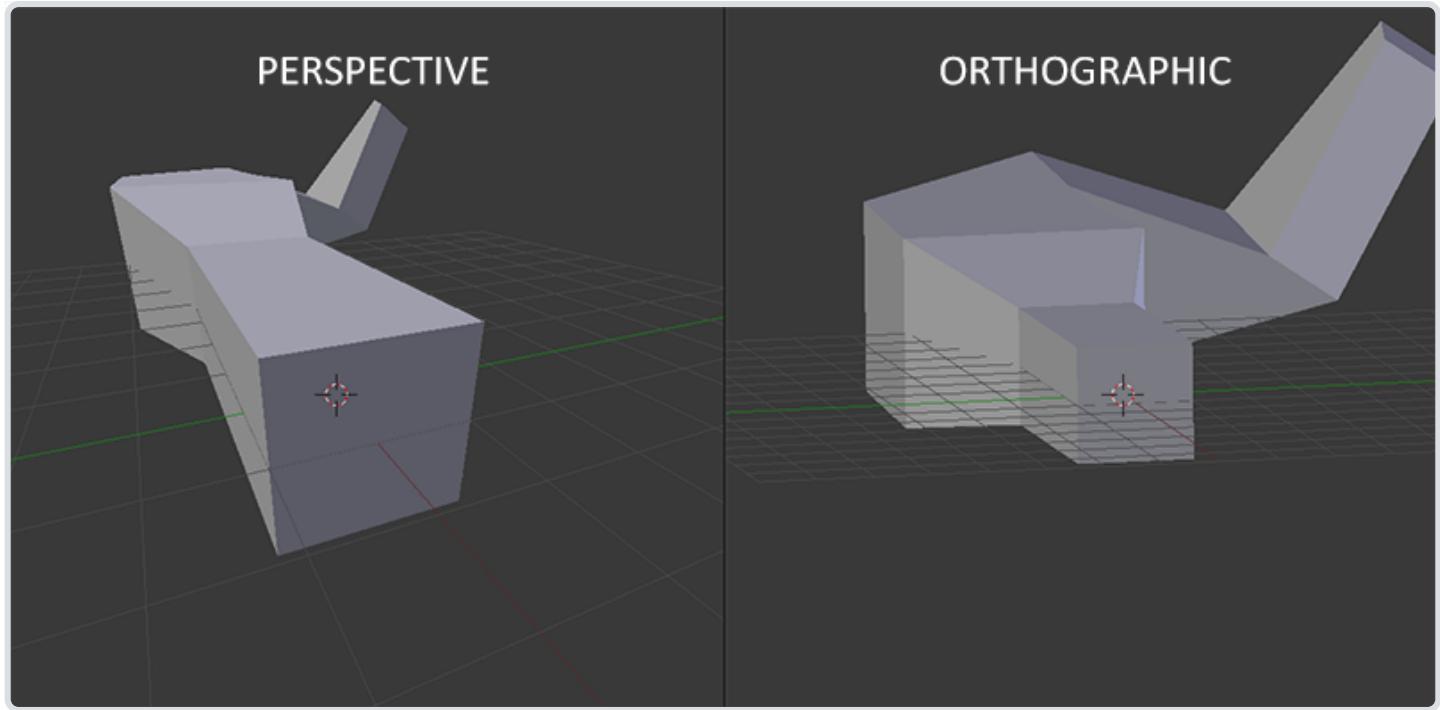
同样，`glm::perspective`所做的其实就是创建了一个定义了可视空间的大平截头体，任何在这个平截头体以外的东西最后都不会出现在裁剪空间体积内，并且将会受到裁剪。一个透视平截头体可以被看作一个不均匀形状的箱子，在这个箱子内部的每个坐标都会被映射到裁剪空间上的一个点。下面是一张透视平截头体的图片：



它的第一个参数定义了`fov`的值，它表示的是**视野**(Field of View)，并且设置了观察空间的大小。如果想要一个真实的观察效果，它的值通常设置为45.0f，但想要一个末日风格的结果你可以将其设置一个更大的值。第二个参数设置了宽高比，由视口的宽除以高所得。第三和第四个参数设置了平截头体的近和远平面。我们通常设置近距离为0.1f，而远距离设为100.0f。所有在近平面和远平面内且处于平截头体内的顶点都会被渲染。

当你把透视矩阵的 `near` 值设置太大时（如10.0f），OpenGL会将靠近摄像机的坐标（在0.0f和10.0f之间）都裁剪掉，这会导致一个你在游戏中很熟悉的视觉效果：在太过靠近一个物体的时候你的视线会直接穿过去。

当使用正射投影时，每一个顶点坐标都会直接映射到裁剪空间中而不经过任何精细的透视除法（它仍然会进行透视除法，只是w分量没有被改变（它保持为1），因此没有起作用）。因为正射投影没有使用透视，远处的物体不会显得更小，所以产生奇怪的视觉效果。由于这个原因，正射投影主要用于二维渲染以及一些建筑或工程的程序，在这些场景中我们更希望顶点不会被透视所干扰。某些如Blender等进行三维建模的软件有时在建模时也会使用正射投影，因为它在各个维度下都更准确地描绘了每个物体。下面你能够看到在Blender里面使用两种投影方式的对比：



你可以看到，使用透视投影的话，远处的顶点看起来比较小，而在正射投影中每个顶点距离观察者的距离都是一样的。

把它们都组合到一起

我们为上述的每一个步骤都创建了一个变换矩阵：模型矩阵、观察矩阵和投影矩阵。一个顶点坐标将会根据以下过程被变换到裁剪坐标：

注意矩阵运算的顺序是相反的（记住我们需要从右往左阅读矩阵的乘法）。最后的顶点应该被赋值到顶点着色器中的`gl_Position`，OpenGL将会自动进行透视除法和裁剪。

然后呢？

顶点着色器的输出要求所有的顶点都在裁剪空间内，这正是我们刚才使用变换矩阵所做的。OpenGL然后对裁剪坐标执行透视除法从而将它们变换到标准化设备坐标。OpenGL会使用`glViewport`内部的参数来将标准化设备坐标映射到屏幕坐标，每个坐标都关联了一个屏幕上的点（在我们的例子中是一个800x600的屏幕）。这个过程称为视口变换。

这一章的主题可能会比较难理解，如果你仍然不确定每个空间的作用的话，你也不必太担心。接下来你会看到我们是怎样运用这些坐标空间的，而且之后也会有足够多的例子。

2.2.19 进入3D

既然我们知道了如何将3D坐标变换为2D坐标，我们可以开始使用真正的3D物体，而不是枯燥的2D平面了。

在开始进行3D绘图时，我们首先创建一个模型矩阵。这个模型矩阵包含了位移、缩放与旋转操作，它们会被应用到所有物体的顶点上，以变换它们到全局的世界空间。让我们变换一下我们的平面，将其绕着x轴旋转，使它看起来像放在地上一样。这个模型矩阵看起来是这样的：

```
glm::mat4 model;
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

通过将顶点坐标乘以这个模型矩阵，我们将该顶点坐标变换到世界坐标。我们的平面看起来就是在地板上，代表全局世界里的平面。

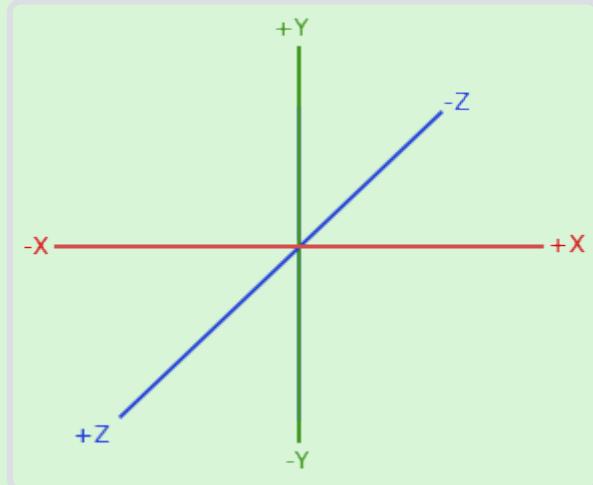
接下来我们需要创建一个观察矩阵。我们想要在场景里面稍微往后移动，以使得物体变成可见的（当在世界空间时，我们位于原点(0,0,0)）。要想在场景里面移动，先仔细想一想下面这个句子：

- 将摄像机向后移动，和将整个场景向前移动是一样的。

这正是观察矩阵所做的，我们以相反于摄像机移动的方向移动整个场景。因为我们想要往后移动，并且OpenGL是一个右手坐标系(Right-handed System)，所以我们需要沿着z轴的正方向移动。我们会通过将场景沿着z轴负方向平移来实现。它会给我们一种我们在往后移动的感觉。

右手坐标系(Right-handed System)

按照惯例，OpenGL是一个右手坐标系。简单来说，就是正x轴在你的右手边，正y轴朝上，而正z轴是朝向后方的。想象你的屏幕处于三个轴的中心，则正z轴穿过你的屏幕朝向你。坐标系画起来如下：



为了理解为什么被称为右手坐标系，按如下的步骤做：

- 沿着正y轴方向伸出你的右臂，手指着上方。
- 大拇指指向右方。
- 食指指向上方。
- 中指向下弯曲90度。

如果你的动作正确，那么你的大拇指指向正x轴方向，食指指向正y轴方向，中指指向正z轴方向。如果你用左臂来做这些动作，你会发现z轴的方向是相反的。这个叫做左手坐标系，它被DirectX广泛地使用。注意在标准化设备坐标系中OpenGL实际上使用的是左手坐标系（投影矩阵交换了左右手）。

在下一个教程中我们将会详细讨论如何在场景中移动。就目前来说，观察矩阵是这样的：

```
glm::mat4 view;
// 注意，我们将矩阵向我们要进行移动场景的反方向移动。
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

最后我们需要做的是定义一个投影矩阵。我们希望在场景中使用透视投影，所以像这样声明一个投影矩阵：

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight, 0.1f, 100.0f);
```

既然我们已经创建了变换矩阵，我们应该将它们传入着色器。首先，让我们在顶点着色器中声明一个uniform变换矩阵然后将它乘以顶点坐标：

```
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // 注意乘法要从右向左读
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    ...
}
```

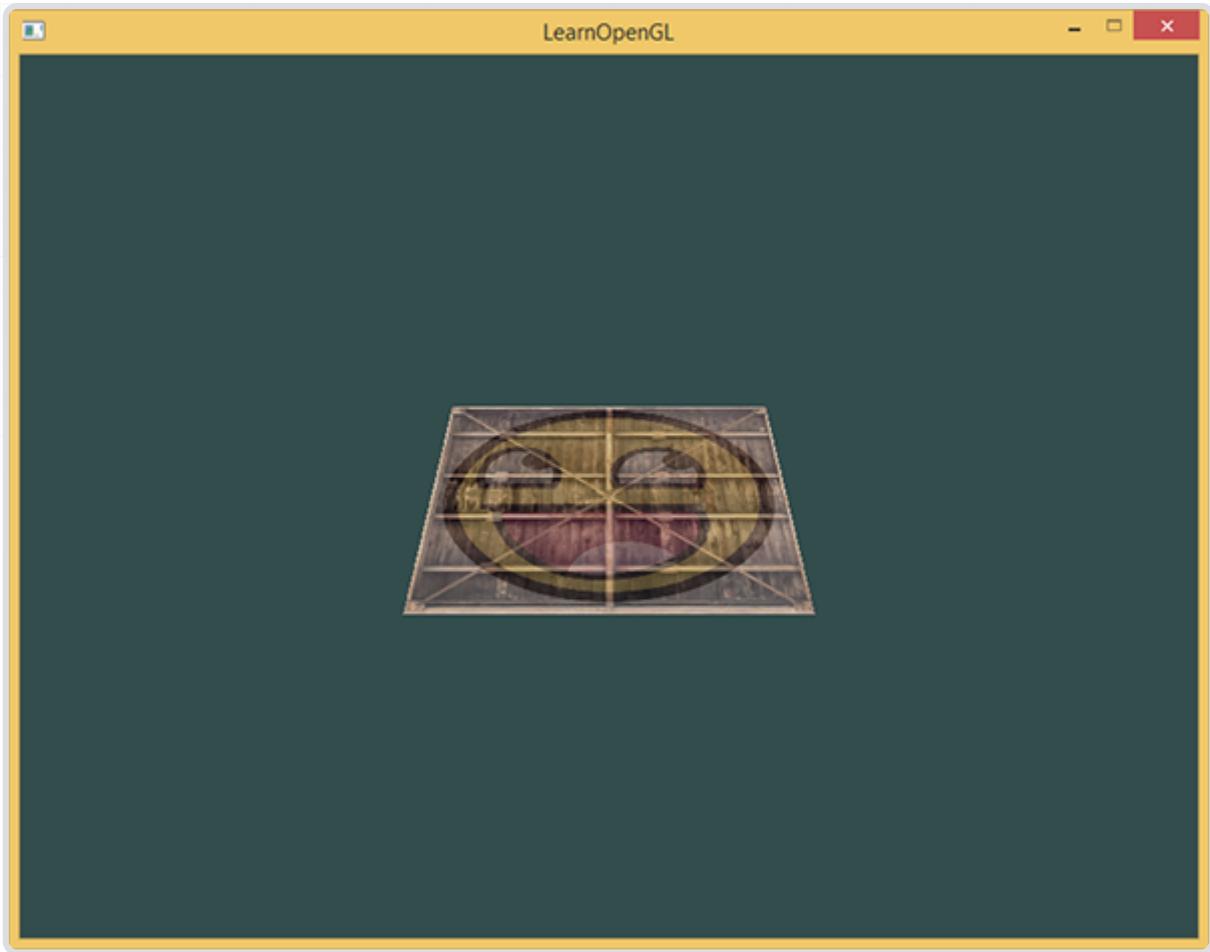
我们还应该将矩阵传入着色器（这通常在每次的渲染迭代中进行，因为变换矩阵会经常变动）：

```
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
... // 观察矩阵和投影矩阵与之类似
```

我们的顶点坐标已经使用模型、观察和投影矩阵进行变换了，最终的物体应该会：

- 稍微向后倾斜至地板方向。
- 离我们有一些距离。
- 有透视效果（顶点越远，变得越小）。

让我们检查一下结果是否满足这些要求：



它看起来就像是一个3D的平面，静止在一个虚构的地板上。如果你得到的不是相同的结果，请检查下完整的[源代码](#)。

更加 3D

到目前为止，我们一直都在使用一个2D平面，而且甚至是在3D空间里！所以，让我们大胆地拓展我们的2D平面为一个3D立方体。要想渲染一个立方体，我们一共需要36个顶点（6个面 x 每个面有2个三角形组成 x 每个三角形有3个顶点），这36个顶点的位置你可以从[这里](#)获取。

为了有趣一点，我们将让立方体随着时间旋转：

```
model = glm::rotate(model, (float)glfwGetTime() * glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));
```

然后我们使用`glDrawArrays`来绘制立方体，但这一次总共有36个顶点。

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

如果一切顺利的话你应该能得到下面这样的效果：

这的确有点像是一个立方体，但又有种说不出的奇怪。立方体的某些本应被遮挡住的面被绘制在了这个立方体其他面上。之所以这样是因为OpenGL是一个三角形一个三角形地来绘制你的立方体的，所以即便之前那里有东西它也会覆盖之前的像素。因为这个原因，有些三角形会被绘制在其它三角形上面，虽然它们本不应该是被覆盖的。

幸运的是，OpenGL存储深度信息在一个叫做**Z缓冲**(Z-buffer)的缓冲中，它允许OpenGL决定何时覆盖一个像素而何时不覆盖。通过使用Z缓冲，我们可以配置OpenGL来进行深度测试。

Z缓冲

OpenGL存储它的所有深度信息于一个Z缓冲(Z-buffer)中，也被称为**深度缓冲**(Depth Buffer)。GLFW会自动为你生成这样一个缓冲（就像它也有一个颜色缓冲来存储输出图像的颜色）。深度值存储在每个片段里面（作为片段的z值），当片段想要输出它的颜色时，OpenGL会将它的深度值和Z缓冲进行比较，如果当前的片段在其它片段之后，它将会被丢弃，否则将会覆盖。这个过程称为**深度测试**(Depth Testing)，它是由OpenGL自动完成的。

然而，如果我们想要确定OpenGL真的执行了深度测试，首先我们要告诉OpenGL我们想要启用深度测试；它默认是关闭的。我们可以通过`glEnable`函数来开启深度测试。`glEnable`和`glDisable`函数允许我们启用或禁用某个OpenGL功能。这个功能会一直保持启用/禁用状态，直到另一个调用来禁用/启用它。现在我们想启用深度测试，需要开启`GL_DEPTH_TEST`：

```
glEnable(GL_DEPTH_TEST);
```

因为我们使用了深度测试，我们也想要在每次渲染迭代之前清除深度缓冲（否则前一帧的深度信息仍然保存在缓冲中）。就像清除颜色缓冲一样，我们可以通过在`glClear`函数中指定`DEPTH_BUFFER_BIT`位来清除深度缓冲：

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

我们来重新运行下程序看看OpenGL是否执行了深度测试：

就是这样！一个开启了深度测试，各个面都是纹理，并且还在旋转的立方体！如果你的程序有问题可以到[这里](#)下载源码进行比对。

更多的立方体！

现在我们想在屏幕上显示10个立方体。每个立方体看起来都是一样的，区别在于它们在世界的位置及旋转角度不同。立方体的图形布局已经定义好了，所以当渲染更多物体的时候我们不需要改变我们的缓冲数组和属性数组，我们唯一需要做的只是改变每个对象的模型矩阵来将立方体变换到世界坐标系中。

首先，让我们为每个立方体定义一个位移向量来指定它在世界空间的位置。我们将在一个`glm::vec3`数组中定义10个立方体位置：

```
glm::vec3 cubePositions[] = {
    glm::vec3( 0.0f,  0.0f,  0.0f),
    glm::vec3( 2.0f,  5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3( 2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f,  3.0f, -7.5f),
    glm::vec3( 1.3f, -2.0f, -2.5f),
    glm::vec3( 1.5f,  2.0f, -2.5f),
    glm::vec3( 1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f)
};
```

现在，在游戏循环中，我们调用`glDrawArrays` 10次，但这次在我们渲染之前每次传入一个不同的模型矩阵到顶点着色器中。我们将会在游戏循环中创建一个小的循环用不同的模型矩阵渲染我们的物体10次。注意我们也对每个箱子加了一点旋转：

```
glBindVertexArray(VAO);
for(unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model;
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    ourShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

这段代码将会在每次新立方体绘制出来的时候更新模型矩阵，如此总共重复10次。然后我们应该就能看到一个拥有10个正在奇葩地旋转着的立方体的世界。



完美！看起来我们的箱子已经找到志同道合的小伙伴了。如果你在这里卡住了，你可以对照一下[源代码](#)。

练习

- 对GLM的`projection`函数中的`FoV`和`aspect-ratio`参数进行实验。看能否搞懂它们是如何影响透视平截头体的。
- 将观察矩阵在各个方向上进行位移，来看看场景是如何改变的。注意把观察矩阵当成摄像机对象。
- 使用模型矩阵只让3倍数的箱子旋转（以及第1个箱子），而让剩下的箱子保持静止。[参考解答](#)。

2.2.20 摄像机

原文 [Camera](#)

作者 JoeyDeVries

翻译 [Django](#), [Krasjet](#), [Geequlim](#), [BLumia](#)

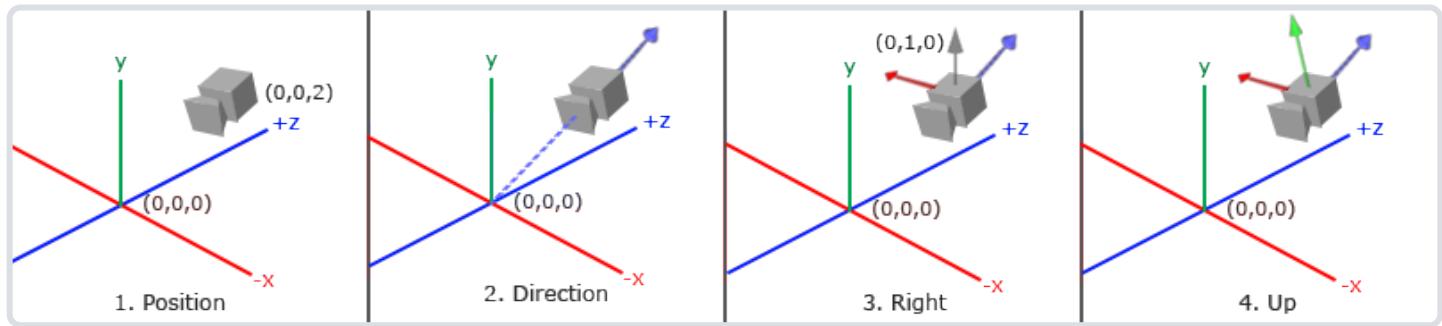
校对 暂未校对

前面的教程中我们讨论了观察矩阵以及如何使用观察矩阵移动场景（我们向后移动了一点）。OpenGL本身没有摄像机(Camera)的概念，但我们可以把场景中的所有物体往相反方向移动的方式来模拟出摄像机，产生一种我们在移动的感觉，而不是场景在移动。

本节我们将会讨论如何在OpenGL中配置一个摄像机，并且将会讨论FPS风格的摄像机，让你能够在3D场景中自由移动。我们也会讨论键盘和鼠标输入，最终完成一个自定义的摄像机类。

摄像机/观察空间

当我们讨论摄像机/观察空间(Camera/View Space)的时候，是在讨论以摄像机的视角作为场景原点时场景中所有的顶点坐标：观察矩阵把所有的世界坐标变换为相对于摄像机位置与方向的观察坐标。要定义一个摄像机，我们需要它在世界空间中的位置、观察的方向、一个指向它右侧的向量以及一个指向它上方的向量。细心的读者可能已经注意到我们实际上创建了一个三个单位轴相互垂直的、以摄像机的位置为原点的坐标系。



1. 摑像机位置

获取摄像机位置很简单。摄像机位置简单来说就是世界空间中一个指向摄像机位置的向量。我们把摄像机位置设置为上一节中的那个相同的位置：

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

不要忘记正z轴是从屏幕指向你的，如果我们希望摄像机向后移动，我们就沿着z轴的正方向移动。

2. 摄像机方向

下一个需要的向量是摄像机的方向，这里指的是摄像机指向哪个方向。现在我们让摄像机指向场景原点：(0, 0, 0)。还记得如果将两个矢量相减，我们就能得到这两个矢量的差吗？用场景原点向量减去摄像机位置向量的结果就是摄像机的指向向量。由于我们知道摄像机指向z轴负方向，但我们希望方向向量(Direction Vector)指向摄像机的z轴正方向。如果我们交换相减的顺序，我们就会获得一个指向摄像机正z轴方向的向量：

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```

方向向量(Direction Vector)并不是最好的名字，因为它实际上指向从它到目标向量的相反方向（译注：注意看前面的那个图，蓝色的方向向量大概指向z轴的正方向，与摄像机实际指向的方向是正好相反的）。

3. 右轴

我们需要的另一个向量是一个右向量(Right Vector)，它代表摄像机空间的x轴的正方向。为获取右向量我们需要先使用一个小技巧：先定义一个上向量(Up Vector)。接下来把上向量和第二步得到的方向向量进行叉乘。两个向量叉乘的结果会同时垂直于两向量，因此我们会得到指向x轴正方向的那个向量（如果我们交换两个向量叉乘的顺序就会得到相反的指向x轴负方向的向量）：

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

4. 上轴

现在我们已经有了x轴向量和z轴向量，获取一个指向摄像机的正y轴向量就相对简单了：我们把右向量和方向向量进行叉乘：

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

在叉乘和一些小技巧的帮助下，我们创建了所有构成观察/摄像机空间的向量。对于想学到更多数学原理的读者，提示一下，在线性代数中这个处理叫做[格拉姆—施密特正交化](#)(Gram-Schmidt Process)。使用这些摄像机向量我们就可以创建一个LookAt矩阵了，它在创建摄像机的时候非常有用。

Look At

使用矩阵的好处之一是如果你使用3个相互垂直（或非线性）的轴定义了一个坐标空间，你可以用这3个轴外加一个平移向量来创建一个矩阵，并且你可以用这个矩阵乘以任何向量来将其变换到那个坐标空间。这正是LookAt矩阵所做的，现在我们有了3个相互垂直的轴和一个定义摄像机空间的位置坐标，我们可以创建我们自己的LookAt矩阵了：

其中是右向量，是上向量，是方向向量是摄像机位置向量。注意，位置向量是相反的，因为我们最终希望把世界平移到与我们自身移动的相反方向。把这个LookAt矩阵作为观察矩阵可以很高效地把所有世界坐标变换到刚刚定义的观察空间。LookAt矩阵就像它的名字表达的那样：它会创建一个看着(Look at)给定目标的观察矩阵。

幸运的是，GLM已经提供了这些支持。我们要做的只是定义一个摄像机位置，一个目标位置和一个表示世界空间中的上向量（我们计算右向量使用的那个上向量）。接着GLM就会创建一个LookAt矩阵，我们可以把它当作我们的观察矩阵：

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                  glm::vec3(0.0f, 0.0f, 0.0f),
                  glm::vec3(0.0f, 1.0f, 0.0f));
```

`glm::LookAt`函数需要一个位置、目标和上向量。它会创建一个和在上一节使用的一样的观察矩阵。

在讨论用户输入之前，我们先来做些有意思的事，把我们的摄像机在场景中旋转。我们会将摄像机的注视点保持在(0, 0, 0)。

我们需要用到一点三角学的知识来在每一帧创建一个x和z坐标，它会代表圆上的一点，我们将会使用它作为摄像机的位置。通过重新计算x和y坐标，我们会遍历圆上的所有点，这样摄像机就会绕着场景旋转了。我们预先定义这个圆的半径`radius`，在每次渲染迭代中使用GLFW的`glfwGetTime`函数重新创建观察矩阵，来扩大这个圆。

```
float radius = 10.0f;
float camX = sin(glfwGetTime()) * radius;
float camZ = cos(glfwGetTime()) * radius;
glm::mat4 view;
view = glm::lookAt(glm::vec3(camX, 0.0, camZ), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
```

如果你运行代码，应该会得到下面的结果：

通过这一小段代码，摄像机现在会随着时间流逝围绕场景转动了。自己试试改变半径和位置/方向参数，看看LookAt矩阵是如何工作的。同时，如果你在哪卡住的话，这里有[源码](#)。

2.2.21 自由移动

让摄像机绕着场景转的确很有趣，但是让我们自己移动摄像机会更有趣！首先我们必须设置一个摄像机系统，所以在我们的程序前面定义一些摄像机变量很有用：

```
glm::vec3 cameraPos    = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp    = glm::vec3(0.0f, 1.0f, 0.0f);
```

`LookAt` 函数现在成了：

```
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

我们首先将摄像机位置设置为之前定义的`cameraPos`。方向是当前的位置加上我们刚刚定义的方向向量。这样能保证无论我们怎么移动，摄像机都会注视着目标方向。让我们摆弄一下这些向量，在按下某些按钮时更新`cameraPos`向量。

我们已经为GLFW的键盘输入定义过一个`processInput`函数了，我们来新添加几个需要检查的按键命令：

```
void processInput(GLFWwindow *window)
{
    ...
    float cameraSpeed = 0.05f; // adjust accordingly
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

当我们按下**WASD**键的任意一个，摄像机的位置都会相应更新。如果我们希望向前或向后移动，我们就把位置向量加上或减去方向向量。如果我们希望向左右移动，我们使用叉乘来创建一个右向量(Right Vector)，并沿着它相应移动就可以了。这样就创建了使用摄像机时熟悉的**横移**(Strafe)效果。

注意，我们对右向量进行了标准化。如果我们没对这个向量进行标准化，最后的叉乘结果会根据`cameraFront`变量返回大小不同的向量。如果我们不对向量进行标准化，我们就得根据摄像机的朝向不同加速或减速移动了，但如果进行了标准化移动就是匀速的。

现在你就应该能够移动摄像机了，虽然移动速度和系统有关，你可能会需要调整一下`cameraSpeed`。

移动速度

目前我们的移动速度是个常量。理论上没什么问题，但是实际情况下根据处理器的能力不同，有些人可能会比其他人每秒绘制更多帧，也就是以更高的频率调用`processInput`函数。结果就是，根据配置的不同，有些人可能移动很快，而有些人会移动很慢。当你发布你的程序的时候，你必须确保它在所有硬件上移动速度都一样。

图形程序和游戏通常会跟踪一个**时间差**(Deltatime)变量，它储存了渲染上一帧所用的时间。我们把所有速度都去乘以`deltaTime`值。结果就是，如果我们的`deltaTime`很大，就意味着上一帧的渲染花费了更多时间，所以这一帧的速度需要变得更高来平衡渲染所花去的时间。使用这种方法时，无论你的电脑快还是慢，摄像机的速度都会相应平衡，这样每个用户的体验就都一样了。

我们跟踪两个全局变量来计算出`deltaTime`值：

```
float deltaTime = 0.0f; // 当前帧与上一帧的时间差
float lastFrame = 0.0f; // 上一帧的时间
```

在每一帧中我们计算出新的`deltaTime`以备后用。

```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

现在我们有了`deltaTime`, 在计算速度的时候可以将其考虑进去了:

```
void processInput(GLFWwindow *window)
{
    float cameraSpeed = 2.5f * deltaTime;
    ...
}
```

与前面的部分结合在一起, 我们有了一个更流畅点的摄像机系统:

现在我们有了一个在任何系统上移动速度都一样的摄像机。同样, 如果你卡住了, 查看一下[源码](#)。我们可以看到任何移动都会影响返回的`deltaTime`值。

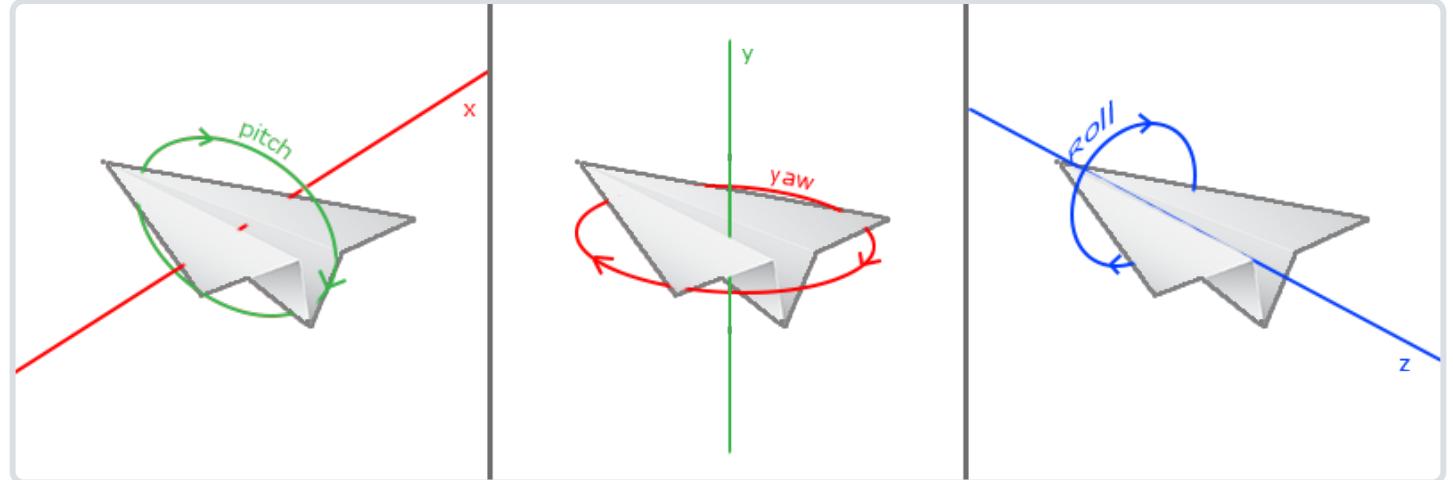
2.2.22 视角移动

只用键盘移动没什么意思。特别是我们还不能转向, 移动很受限制。是时候加入鼠标了!

为了能够改变视角, 我们需要根据鼠标的输入改变`cameraFront`向量。然而, 根据鼠标移动改变方向向量有点复杂, 需要一些三角学知识。如果你对三角学知之甚少, 别担心, 你可以跳过这一部分, 直接复制粘贴我们的代码; 当你想了解更多的时候再回来看。

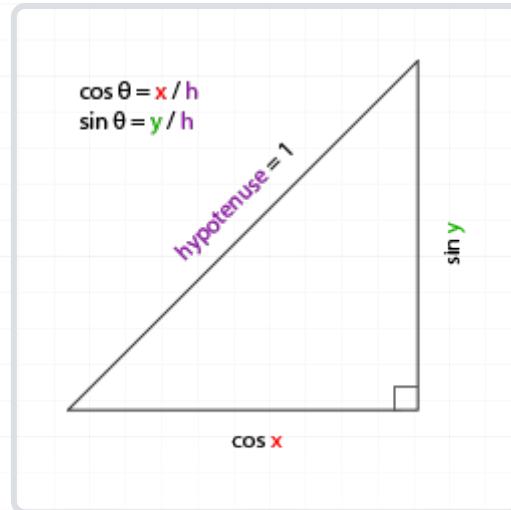
欧拉角

欧拉角(Euler Angle)是可以表示3D空间中任何旋转的3个值, 由莱昂哈德·欧拉(Leonhard Euler)在18世纪提出。一共有3种欧拉角: 俯仰角(Pitch)、偏航角(Yaw)和滚转角(Roll), 下面的图片展示了它们的含义:

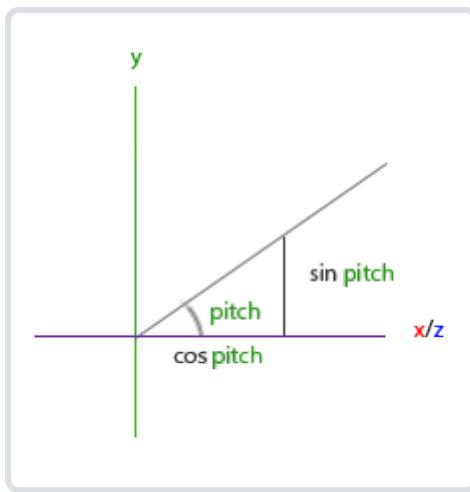


俯仰角是描述我们如何往上或往下看的角, 可以在第一张图中看到。第二张图展示了**偏航角**, 偏航角表示我们往左和往右看的程度。**滚转角**代表我们如何翻滚摄像机, 通常在太空飞船的摄像机中使用。每个欧拉角都有一个值来表示, 把三个角结合起来我们就能够计算3D空间中任何的旋转向量了。

对于我们的摄像机系统来说, 我们只关心俯仰角和偏航角, 所以我们不会讨论滚转角。给定一个俯仰角和偏航角, 我们可以把它们转换为一个代表新的方向向量的3D向量。俯仰角和偏航角转换为方向向量的处理需要一些三角学知识, 我们先从最基本的情况开始:



如果我们把斜边边长定义为1，我们就能知道邻边的长度是，它的对边是。这样我们获得了能够得到x和y方向长度的通用公式，它们取决于所给的角度。我们使用它来计算方向向量的分量：



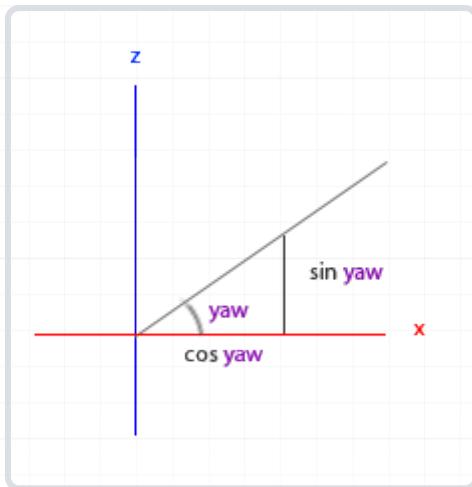
这个三角形看起来和前面的三角形很像，所以如果我们想象自己在xz平面上，看向y轴，我们可以基于第一个三角形计算来计算它的长度/y方向的强度(Strength)（我们往上或往下看多少）。从图中我们可以看到对于一个给定俯仰角的y值等于：

```
direction.y = sin(glm::radians(pitch)); // 注意我们先把角度转为弧度
```

这里我们只更新了y值，仔细观察x和z分量也被影响了。从三角形中我们可以看到它们的值等于：

```
direction.x = cos(glm::radians(pitch));
direction.z = cos(glm::radians(pitch));
```

看看我们是否能够为偏航角找到需要的分量：



就像俯仰角的三角形一样，我们可以看到x分量取决于 `cos(yaw)` 的值，z值同样取决于偏航角的正弦值。把这个加到前面的值中，会得到基于俯仰角和偏航角的方向向量：

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw)); // 译注: direction代表摄像机的前轴(Front)，这个前轴是和本文第一帧
direction.y = sin(glm::radians(pitch));
direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
```

这样我们就有了一个可以把俯仰角和偏航角转化为用来自由旋转视角的摄像机的3维方向向量了。你可能会奇怪：我们怎么得到俯仰角和偏航角？

鼠标输入

偏航角和俯仰角是通过鼠标（或手柄）移动获得的，水平的移动影响偏航角，竖直的移动影响俯仰角。它的原理就是，储存上一帧鼠标的位置，在当前帧中我们当前计算鼠标位置与上一帧的位置相差多少。如果水平/竖直差别越大那么俯仰角或偏航角就改变越大，也就是摄像机需要移动更多的距离。

首先我们要告诉GLFW，它应该隐藏光标，并捕捉(Capture)它。捕捉光标表示的是，如果焦点在你的程序上（译注：即表示你正在操作这个程序，Windows中拥有焦点的程序标题栏通常是有颜色的那个，而失去焦点的程序标题栏则是灰色的），光标应该停留在窗口中（除非程序失去焦点或者退出）。我们可以用一个简单地配置调用完成：

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

在调用这个函数之后，无论我们怎么去移动鼠标，光标都不会显示了，它也不会离开窗口。对于FPS摄像机系统来说非常完美。

为了计算俯仰角和偏航角，我们需要让GLFW监听鼠标移动事件。（和键盘输入相似）我们会用一个回调函数来完成，函数的原型如下：

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```

这里的`xpos`和`ypos`代表当前鼠标的位置。当我们用GLFW注册了回调函数之后，鼠标一移动`mouse_callback`函数就会被调用：

```
glfwSetCursorPosCallback(window, mouse_callback);
```

在处理FPS风格摄像机的鼠标输入的时候，我们必须在最终获取方向向量之前做下面这几步：

1. 计算鼠标距上一帧的偏移量。
2. 把偏移量添加到摄像机的俯仰角和偏航角中。
3. 对偏航角和俯仰角进行最大和最小值的限制。
4. 计算方向向量。

第一步是计算鼠标自上一帧的偏移量。我们必须先在程序中储存上一帧的鼠标位置，我们把它的初始值设置为屏幕的中心（屏幕的尺寸是800x600）：

```
float lastX = 400, lastY = 300;
```

然后在鼠标的回调函数中我们计算当前帧和上一帧鼠标位置的偏移量：

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // 注意这里是相反的，因为y坐标是从底部往顶部依次增大的
lastX = xpos;
lastY = ypos;

float sensitivity = 0.05f;
xoffset *= sensitivity;
yoffset *= sensitivity;
```

注意我们把偏移量乘以了 `sensitivity` (灵敏度) 值。如果我们忽略这个值，鼠标移动就会太大了；你可以自己实验一下，找到适合自己的灵敏度值。

接下来我们把偏移量加到全局变量 `pitch` 和 `yaw` 上：

```
yaw += xoffset;
pitch += yoffset;
```

第三步，我们需要给摄像机添加一些限制，这样摄像机就不会发生奇怪的移动了（这样也会避免一些奇怪的问题）。对于俯仰角，要让用户不能看向高于89度的地方（在90度时视角会发生逆转，所以我们把89度作为极限），同样也不允许小于-89度。这样能够保证用户只能看到天空或脚下，但是不能超越这个限制。我们可以在值超过限制的时候将其改为极限值来实现：

```
if(pitch > 89.0f)
    pitch = 89.0f;
if(pitch < -89.0f)
    pitch = -89.0f;
```

注意我们没有给偏航角设置限制，这是因为我们不希望限制用户的水平旋转。当然，给偏航角设置限制也很容易，如果你愿意可以自己实现。

第四也是最后一步，就是通过俯仰角和偏航角来计算以得到真正的方向向量：

```
glm::vec3 front;
front.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));
front.y = sin(glm::radians(pitch));
front.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
cameraFront = glm::normalize(front);
```

计算出来的方向向量就会包含根据鼠标移动计算出来的所有旋转了。由于 `cameraFront` 向量已经包含在GLM的 `lookAt` 函数中，我们这就没什么问题了。

如果你现在运行代码，你会发现在窗口第一次获取焦点的时候摄像机会突然跳一下。这个问题产生的原因是，在你的鼠标移动进窗口的那一刻，鼠标回调函数就会被调用，这时候的 `xpos` 和 `ypos` 会等于鼠标刚刚进入屏幕的那个位置。这通常是一个距离屏幕中心很远的地方，因而产生一个很大的偏移量，所以就会跳了。我们可以简单的使用一个 `bool` 变量检验我们是否是第一次获取鼠标输入，如果是，那么我们先把鼠标的初始位置更新为 `xpos` 和 `ypos` 值，这样就能解决这个问题；接下来的鼠标移动就会使用刚进入的鼠标位置坐标来计算偏移量了：

```
if(firstMouse) // 这个bool变量初始时是设定为true的
{
    lastX = xpos;
    lastY = ypos;
    firstMouse = false;
}
```

最后的代码应该是这样的：

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if(firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    float sensitivity = 0.05;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    if(pitch > 89.0f)
        pitch = 89.0f;
    if(pitch < -89.0f)
        pitch = -89.0f;

    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    cameraFront = glm::normalize(front);
}
```

现在我们就可以自由地在3D场景中移动了！

缩放

作为我们摄像机系统的一个附加内容，我们还会来实现一个缩放(Zoom)接口。在之前的教程中我们说视野(Field of View)或fov定义了我们可以看到场景中多大的范围。当视野变小时，场景投影出来的空间就会减小，产生放大(Zoom In)了的感觉。我们会使用鼠标的滚轮来放大。与鼠标移动、键盘输入一样，我们需要一个鼠标滚轮的回调函数：

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    if(fov >= 1.0f && fov <= 45.0f)
        fov -= yoffset;
    if(fov <= 1.0f)
        fov = 1.0f;
    if(fov >= 45.0f)
        fov = 45.0f;
}
```

当滚动鼠标滚轮的时候，yoffset值代表我们竖直滚动的大小。当scroll_callback函数被调用后，我们改变全局变量fov变量的内容。因为45.0f是默认的视野值，我们将会把缩放级别(Zoom Level)限制在1.0f到45.0f。

我们现在在每一帧都必须把透视投影矩阵上传到GPU，但现在使用fov变量作为它的视野：

```
projection = glm::perspective(glm::radians(fov), 800.0f / 600.0f, 0.1f, 100.0f);
```

最后不要忘记注册鼠标滚轮的回调函数：

```
glfwSetScrollCallback(window, scroll_callback);
```

现在，我们就实现了一个简单的摄像机系统了，它能够让我们在3D环境中自由移动。

你可以去自由地实验，如果遇到困难，可以对比[源代码](#)。

注意，使用欧拉角的摄像机系统并不完美。根据你的视角限制或者是配置，你仍然可能引入[万向节死锁](#)问题。最好的摄像机系统是使用四元数(Quaternions)的，但我们将会把这个留到后面讨论。（译注：[这里](#)可以查看四元数摄像机的实现）

2.2.23 摄像机类

接下来的教程中，我们将会一直使用一个摄像机来浏览场景，从各个角度观察结果。然而，由于一个摄像机会占用每篇教程很大的篇幅，我们将会从细节抽象出来，创建我们自己的摄像机对象，它会完成大多数的工作，而且还会提供一些附加的功能。与着色器教程不同，我们不会带你一步一步创建摄像机类，我们只会提供你一份（有完整注释的）代码，如果你想知道它的内部构造的话可以自己去阅读。

和着色器对象一样，我们把摄像机类写在一个单独的头文件中。你可以在[这里](#)找到它，你现在应该能够理解所有的代码了。我们建议您至少看一看这个类，看看如何创建一个自己的摄像机类。

我们介绍的摄像机系统是一个FPS风格的摄像机，它能够满足大多数情况需要，而且与欧拉角兼容，但是在创建不同的摄像机系统，比如飞行模拟摄像机，时就要当心。每个摄像机系统都有自己的优点和不足，所以确保对它们进行了详细研究。比如，这个FPS摄像机不允许俯仰角大于90度，而且我们使用了一个固定的上向量(0, 1, 0)，这在需要考虑滚转角的时候就不能用了。

使用新摄像机对象，更新后版本的源码可以在[这里](#)找到。

练习

- 看看你是否能够修改摄像机类，使得其能够变成一个真正的FPS摄像机（也就是说不能够随意飞行）；你只能够呆在xz平面上：
[参考解答](#)
- 试着创建你自己的LookAt函数，其中你需要手动创建一个我们在一开始讨论的观察矩阵。用你的函数实现来替换GLM的LookAt函数，看看它是否还能一样地工作：[参考解答](#)

2.2.24 复习

原文 [Review](#)

作者 JoeyDeVries

翻译 Krasjet

校对 Geequlim

恭喜您完成了本章的学习，至此为止你应该能够创建一个窗口，创建并且编译着色器，通过缓冲对象或者uniform发送顶点数据，绘制物体，使用纹理，理解向量和矩阵，并且可以综合上述知识创建一个3D场景并通过摄像机来移动。

最后这几章我们学了太多的东西了。你可以尝试在教程的基础上改动程序，或者实验一下，有一点自己的想法并解决问题。一旦你认为你真正熟悉了我们讨论的所有东西，你就可以进行[下一节](#)的学习。

词汇表

- **OpenGL**: 一个定义了函数布局和输出的图形API的正式规范。
- **GLAD**: 一个拓展加载库，用来为我们加载并设定所有OpenGL函数指针，从而让我们能够使用所有（现代）OpenGL函数。
- **视口(Viewport)**: 我们需要渲染的窗口。
- **图形管线(Graphics Pipeline)**: 一个顶点在呈现为像素之前经过的全部过程。
- **着色器(Shader)**: 一个运行在显卡上的小型程序。很多阶段的图形管道都可以使用自定义的着色器来代替原有的功能。
- **标准化设备坐标(Normalized Device Coordinates, NDC)**: 顶点在通过在剪裁坐标系中剪裁与透视线除法后最终呈现在现在的坐标系。所有位置在NDC下-1.0到1.0的顶点将不会被丢弃并且可见。
- **顶点缓冲对象(Vertex Buffer Object)**: 一个调用显存并存储所有顶点数据供显卡使用的缓冲对象。
- **顶点数组对象(Vertex Array Object)**: 存储缓冲区和顶点属性状态。
- **索引缓冲对象(Element Buffer Object)**: 一个存储索引供索引化绘制使用的缓冲对象。
- **Uniform**: 一个特殊类型的GLSL变量。它是全局的（在一个着色器程序中每一个着色器都能够访问uniform变量），并且只需要被设定一次。
- **纹理(Texture)**: 一种包裹着物体的特殊类型图像，给物体精细的视觉效果。
- **纹理缠绕(Texture Wrapping)**: 定义了一种当纹理顶点超出范围(0, 1)时指定OpenGL如何采样纹理的模式。
- **纹理过滤(Texture Filtering)**: 定义了一种当有多种纹素选择时指定OpenGL如何采样纹理的模式。这通常在纹理被放大情况下发生。
- **多级渐远纹理(Mipmaps)**: 被存储的材质的一些缩小版本，根据距观察者的距离会使用材质的合适大小。
- **stb_image.h**: 图像加载库。
- **纹理单元(Texture Units)**: 通过绑定纹理到不同纹理单元从而允许多个纹理在同一对象上渲染。
- **向量(Vector)**: 一个定义了在空间中方向和/或位置的数学实体。
- **矩阵(Matrix)**: 一个矩形阵列的数学表达式。
- **GLM**: 一个为OpenGL打造的数学库。
- **局部空间(Local Space)**: 一个物体的初始空间。所有的坐标都是相对于物体的原点的。
- **世界空间(World Space)**: 所有的坐标都相对于全局原点。
- **观察空间(View Space)**: 所有的坐标都是从摄像机的视角观察的。
- **裁剪空间(Clip Space)**: 所有的坐标都是从摄像机视角观察的，但是该空间应用了投影。这个空间应该是一个顶点坐标最终的空间，作为顶点着色器的输出。OpenGL负责处理剩下的事情（裁剪/透视线除法）。
- **屏幕空间(Screen Space)**: 所有的坐标都由屏幕视角来观察。坐标的范围是从0到屏幕的宽/高。
- **LookAt矩阵**: 一种特殊的观察矩阵，它创建了一个坐标系，其中所有坐标都根据从一个位置正在观察目标的用户旋转或者平移。
- **欧拉角(Euler Angles)**: 被定义为偏航角(Yaw)，俯仰角(Pitch)，和滚转角(Roll)从而允许我们通过这三个值构造任何3D方向。

2.3 光照

2.3.1 颜色

原文 [Colors](#)

作者 JoeyDeVries

翻译 [Geequlim](#), Krasjet

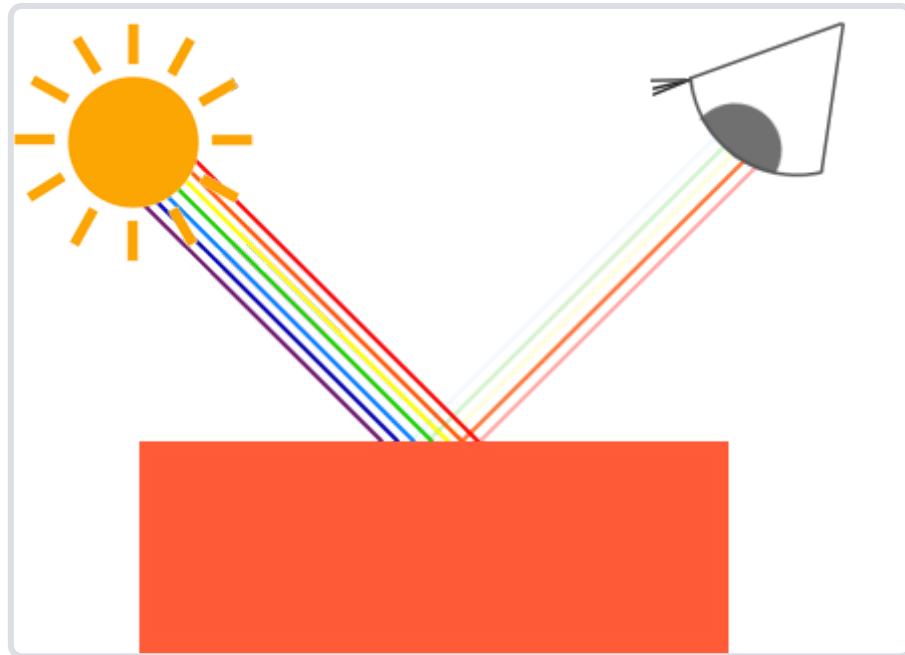
校对 暂未校对

在前面的教程中我们已经简要提到过该如何在OpenGL中使用颜色(Color)，但是我们至今所接触到的都是很浅层的知识。本节我们将会更深入地讨论什么是颜色，并且还会为接下来的光照(Lighting)教程创建一个场景。

现实世界中有无数种颜色，每一个物体都有它们自己的颜色。我们需要使用（有限的）数值来模拟真实世界中（无限）的颜色，所以并不是所有现实世界中的颜色都可以用数值来表示的。然而我们仍能通过数值来表现出非常多的颜色，甚至你可能都不会注意到与现实的颜色有任何的差异。颜色可以数字化的由红色(Red)、绿色(Green)和蓝色(Blue)三个分量组成，它们通常被缩写为RGB。仅仅用这三个值就可以组合出任意一种颜色。例如，要获取一个珊瑚红(Coral)色的话，我们可以定义这样的一个颜色向量：

```
glm::vec3 coral(1.0f, 0.5f, 0.31f);
```

我们在现实生活中看到某一物体的颜色并不是这个物体真正拥有的颜色，而是它所反射的(Reflected)颜色。换句话说，那些不能被物体所吸收(Absorb)的颜色（被拒绝的颜色）就是我们能够感知到的物体的颜色。例如，太阳光能被看见的白光其实是由许多不同的颜色组合而成的（如下图所示）。如果我们将白光照在一个蓝色的玩具上，这个蓝色的玩具会吸收白光中除了蓝色以外的所有子颜色，不被吸收的蓝色光被反射到我们的眼中，让这个玩具看起来是蓝色的。下图显示的是一个珊瑚红的玩具，它以不同强度反射了多个颜色。



你可以看到，白色的阳光实际上是所有可见颜色的集合，物体吸收了其中的大部分颜色。它仅反射了代表物体颜色的部分，被反射颜色的组合就是我们所感知到的颜色（此例中为珊瑚红）。

这些颜色反射的定律被直接地运用在图形领域。当我们在OpenGL中创建一个光源时，我们希望给光源一个颜色。在上一段中我们有一个白色的太阳，所以我们将光源设置为白色。当我们把光源的颜色与物体的颜色值相乘，所得到的就是这个物体所反射的颜色（也就是我们所感知到的颜色）。让我们再次审视我们的玩具（这一次它还是珊瑚红），看看如何在图形学中计算出它的反射颜色。我们将这两个颜色向量作分量相乘，结果就是最终的颜色向量了：

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (1.0f, 0.5f, 0.31f);
```

我们可以看到玩具的颜色吸收了白色光源中很大一部分的颜色，但它根据自身的颜色值对红、绿、蓝三个分量都做出了一定的反射。这也表现了现实中颜色的工作原理。由此，我们可以定义物体的颜色为物体从一个光源反射各个颜色分量的大小。现在，如果我们使用绿色的光源又会发生什么呢？

```
glm::vec3 lightColor(0.0f, 1.0f, 0.0f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (0.0f, 0.5f, 0.0f);
```

可以看到，并没有红色和蓝色的光让我们的玩具来吸收或反射。这个玩具吸收了光线中一半的绿色值，但仍然也反射了一半的绿色值。玩具现在看上去是深绿色(Dark-greenish)的。我们可以看到，如果我们用绿色光源来照射玩具，那么只有绿色分量能被反射和感知到，红色和蓝色都不能被我们所感知到。这样做的结果是，一个珊瑚红的玩具突然变成了深绿色物体。现在我们来看另一个例子，使用深橄榄绿色(Dark olive-green)的光源：

```
glm::vec3 lightColor(0.33f, 0.42f, 0.18f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (0.33f, 0.21f, 0.06f);
```

可以看到，我们可以使用不同的光源颜色来让物体显现出意想不到的颜色。有创意地利用颜色其实并不难。

这些颜色的理论已经足够了，下面我们来构造一个实验用的场景吧。

2.3.2 创建一个光照场景

在接下来的教程中，我们将会广泛地使用颜色来模拟现实世界中的光照效果，创造出一些有趣的视觉效果。由于我们现在将会使用光源了，我们希望将它们显示为可见的物体，并在场景中至少加入一个物体来测试模拟光照的效果。

首先我们需要一个物体来作为被投光(Cast the light)的对象，我们将使用前面教程中的那个著名的立方体箱子。我们还需要一个物体来代表光源在3D场景中的位置。简单起见，我们依然使用一个立方体来代表光源（我们已拥有立方体的[顶点数据](#)是吧？）。

填一个顶点缓冲对象(VBO)，设定一下顶点属性指针和其它一些乱七八糟的东西现在对你来说应该很容易了，所以我们就不再赘述那些步骤了。如果你仍然觉得这很困难，我建议你复习[之前的教程](#)，并且在继续学习之前先把练习过一遍。

我们首先需要一个顶点着色器来绘制箱子。与之前的顶点着色器相比，容器的顶点位置是保持不变的（虽然这一次我们不需要纹理坐标了），因此顶点着色器中没有新的代码。我们将会使用之前教程顶点着色器的精简版：

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

记得更新你的顶点数据和属性指针使其与新的顶点着色器保持一致（当然你可以继续留着纹理数据和属性指针。在这一节中我们将不会用到它们，但有一个全新的开始也不是什么坏主意）。

因为我们还要创建一个表示灯（光源）的立方体，所以我们还要为这个灯创建一个专门的VAO。当然我们也可以让这个灯和其它物体使用同一个VAO，简单地对它的`model`（模型）矩阵做一些变换就好了，然而接下来的教程中我们会频繁地对顶点数据和属性指针做出修改，我们并不想让这些修改影响到灯（我们只关心灯的顶点位置），因此我们有必要为灯创建一个新的VAO。

```
unsigned int lightVAO;
 glGenVertexArrays(1, &lightVAO);
 glBindVertexArray(lightVAO);
 // 只需要绑定VBO不用再次设置VBO的数据，因为箱子的VBO数据中已经包含了正确的立方体顶点数据
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 // 设置灯立方体的顶点属性（对我们的灯来说仅仅只有位置数据）
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);
```

这段代码对你来说应该非常直观。现在我们已经创建了表示灯和被照物体箱子，我们只需要再定义一个片段着色器就行了：

```
#version 330 core
out vec4 FragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;

void main()
{
    FragColor = vec4(lightColor * objectColor, 1.0);
}
```

这个片段着色器从uniform变量中接受物体的颜色和光源的颜色。正如本节一开始所讨论的那样，我们将光源的颜色和物体（反射的）颜色相乘。这个着色器理解起来应该很容易。我们把物体的颜色设置为之前提到的珊瑚红色，并把光源设置为白色。

```
// 在此之前不要忘记首先 use 对应的着色器程序（来设定uniform）
lightingShader.use();
lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
```

要注意的是，当我们修改顶点或者片段着色器后，灯的位置或颜色也会随之改变，这并不是我们想要的效果。我们不希望灯的颜色在接下来的教程中因光照计算的结果而受到影响，而是希望它能够与其它的计算分离。我们希望灯一直保持明亮，不受其它颜色变化的影响（这样它才更像是一个真实的光源）。

为了实现这个目标，我们需要为灯的绘制创建另外的一套着色器，从而能保证它能够在其它光照着色器发生改变的时候不受影响。顶点着色器与我们当前的顶点着色器是一样的，所以你可以直接把现在的顶点着色器用在灯上。灯的片段着色器给灯定义了一个不变的常量白色，保证了灯的颜色一直是亮的：

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0); // 将向量的四个分量全部设置为1.0
}
```

当我们想要绘制我们的物体的时候，我们需要使用刚刚定义的光照着色器来绘制箱子（或者可能是其它的物体）。当我们想要绘制灯的时候，我们会使用灯的着色器。在之后的教程里我们会逐步更新这个光照着色器，从而能够慢慢地实现更真实的效果。

使用这个灯立方体的主要目的是为了让我们知道光源在场景中的具体位置。我们通常在场景中定义一个光源的位置，但这只是一个位置，它并没有视觉意义。为了显示真正的灯，我们将表示光源的立方体绘制在与光源相同的位置。我们将使用我们为它新建的片段着色器来绘制它，让它一直处于白色的状态，不受场景中的光照影响。

我们声明一个全局 `vec3` 变量来表示光源在场景的世界空间坐标中的位置：

```
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
```

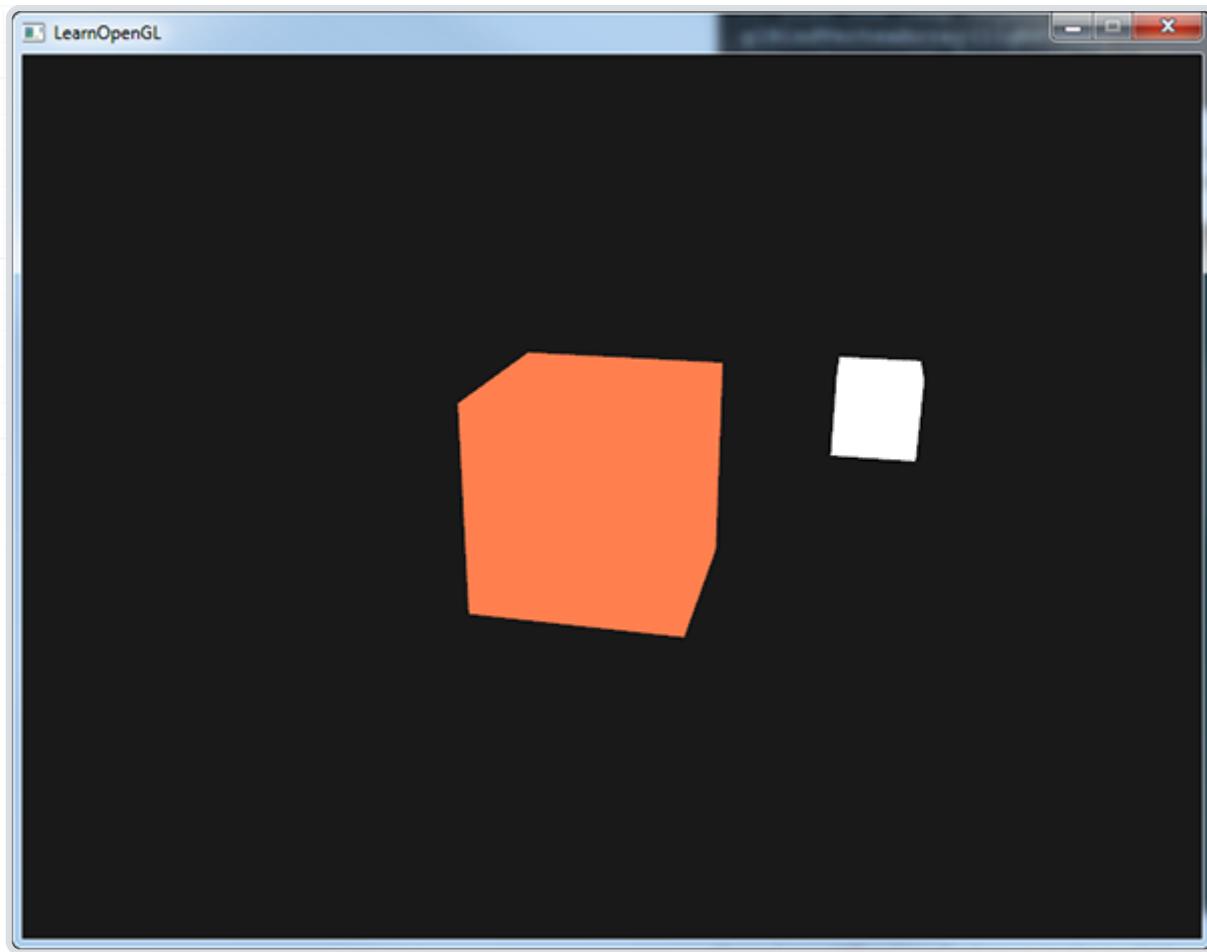
然后我们把灯位移到这里，然后将它缩小一点，让它不那么明显：

```
model = glm::mat4();
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));
```

绘制灯立方体的代码应该与下面的类似：

```
lampShader.use();
// 设置模型、视图和投影矩阵uniform
...
// 绘制灯立方体对象
glBindVertexArray(lightVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

请把上述的所有代码片段放在你程序中合适的位置，这样我们就能有一个干净的光照实验场地了。如果一切顺利，运行效果将会如下图所示：



没什么好看的是吗？但我保证在接下来的教程中它会变得更加有趣。

如果你觉得将上面的代码片段整合到一起非常困难，可以来[这里](#)看一下源代码，并仔细研究我的代码和注释。

我们现在已经对颜色有一定的了解了，并且已经创建了一个简单的场景供我们之后绘制动感的光照，我们可以进入[下一节](#)进行学习，真正的魔法即将开始！

2.3.3 基础光照

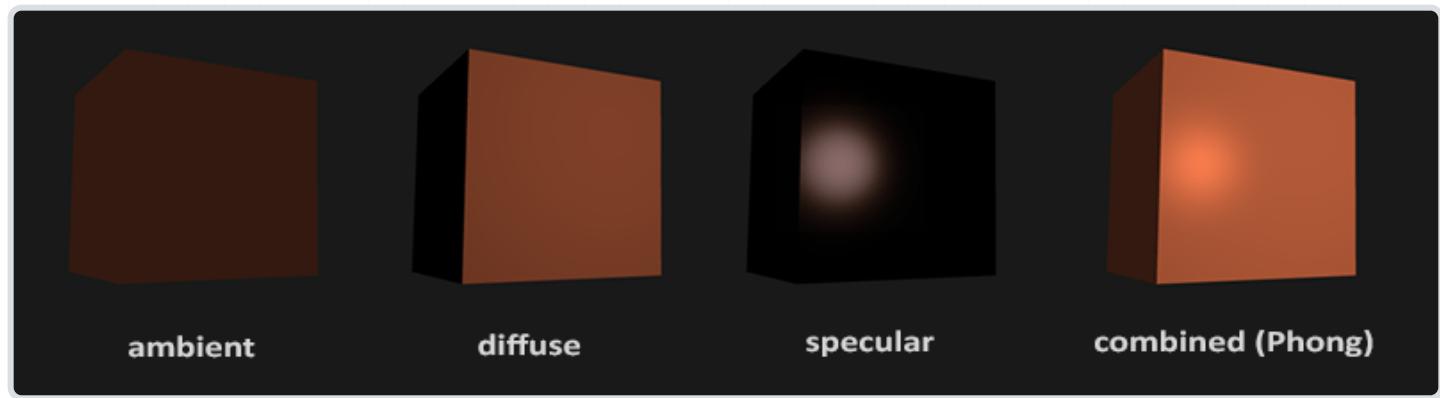
原文 [Basic Lighting](#)

作者 JoeyDeVries

翻译 [Django](#), Krasjet, Geequlim, [BLumia](#)

校对 暂未校对

现实世界的光照是极其复杂的，而且会受到诸多因素的影响，这是我们有限的计算能力所无法模拟的。因此OpenGL的光照使用的是简化的模型，对现实的情况进行近似，这样处理起来会更容易一些，而且看起来也差不多一样。这些光照模型都是基于我们对光的物理特性的理解。其中一个模型被称为[冯氏光照模型](#)(Phong Lighting Model)。冯氏光照模型的主要结构由3个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。下面这张图展示了这些光照分量看起来的样子：



- **环境光照**(Ambient Lighting)：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，我们会使用一个环境光照常量，它永远会给物体一些颜色。
- **漫反射光照**(Diffuse Lighting)：模拟光源对物体的方向性影响(Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮。
- **镜面光照**(Specular Lighting)：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

为了创建有趣的视觉场景，我们希望模拟至少这三种光照分量。我们将以最简单的一个开始：环境光照。

2.3.4 环境光照

光通常都不是来自于同一个光源，而是来自于我们周围分散的很多光源，即使它们可能并不是那么显而易见。光的一个属性是，它可以向很多方向发散并反弹，从而能够到达不是非常直接临近的点。所以，光能够在其它的表面上反射，对一个物体产生间接的影响。考虑到这种情况的算法叫做[全局照明](#)(Global Illumination)算法，但是这种算法既开销高昂又极其复杂。

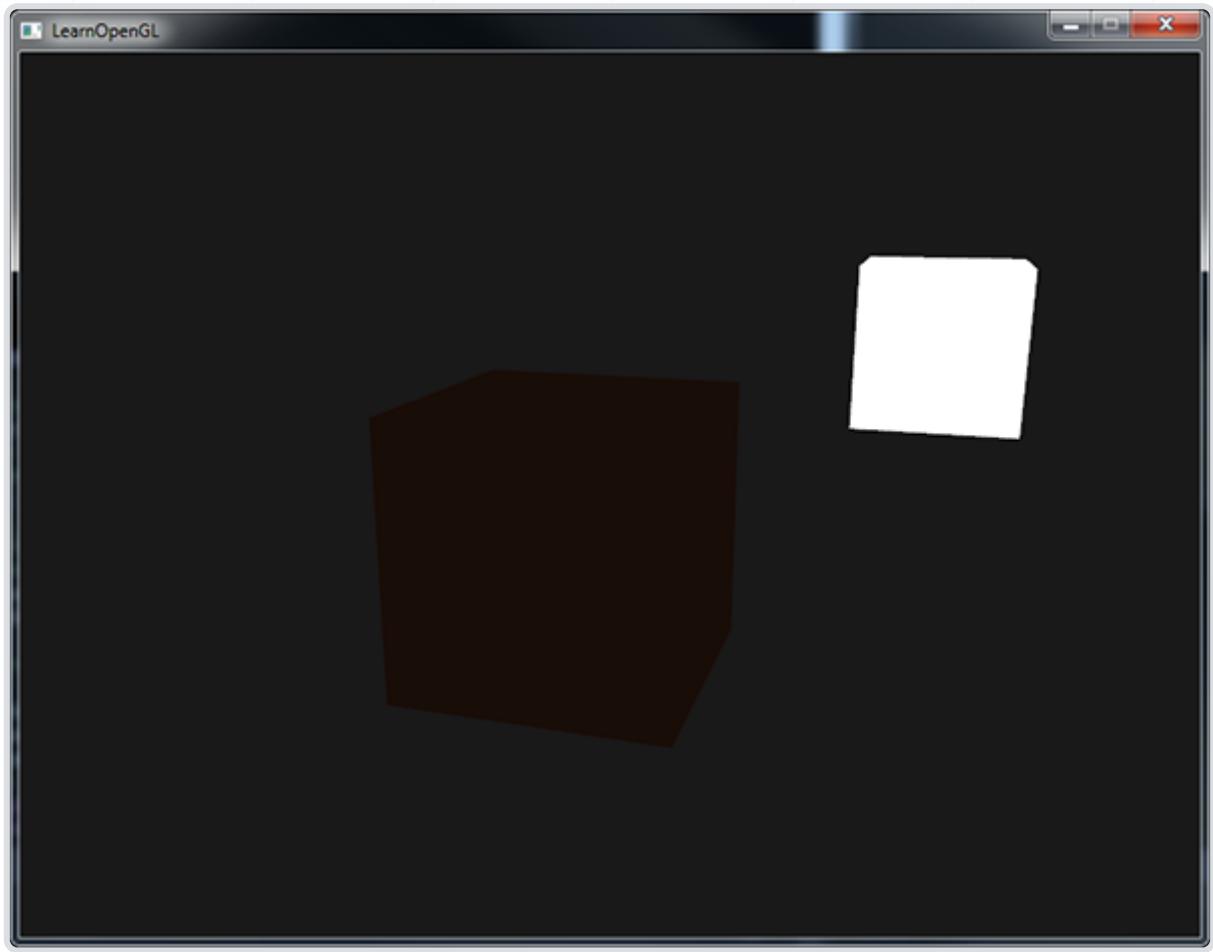
由于我们现在对那种又复杂又开销高昂的算法不是很感兴趣，所以我们将会先使用一个简化的全局照明模型，即**环境光照**。正如你在上一节所学到的，我们使用一个很小的常量（光照）颜色，添加到物体片段的最终颜色中，这样子的话即便场景中没有直接的光源也能看起来存在有一些发散的光。

把环境光照添加到场景里非常简单。我们用光的颜色乘以一个很小的常量环境因子，再乘以物体的颜色，然后将最终结果作为片段的颜色：

```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

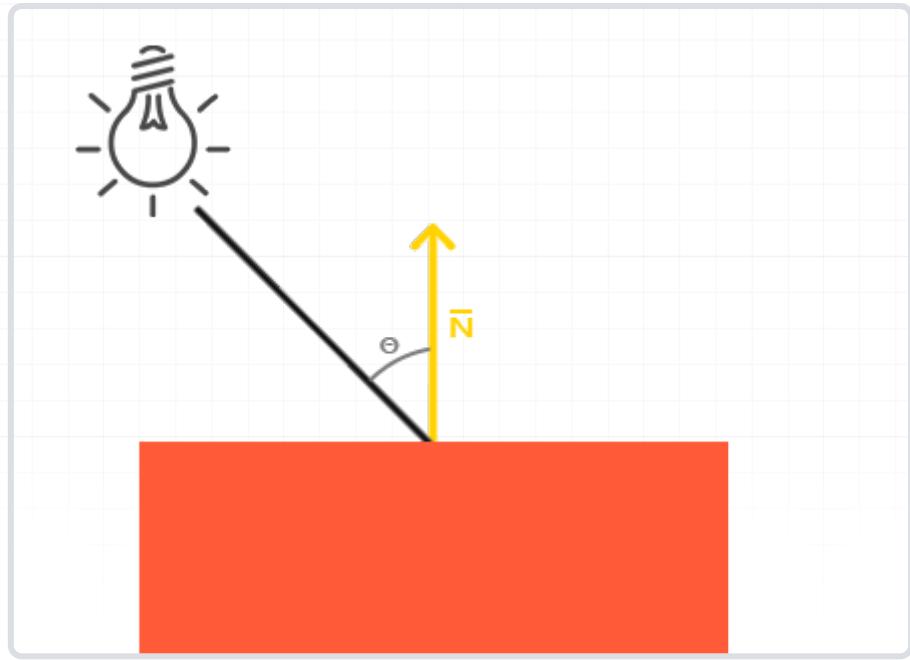
    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```

如果你现在运行你的程序，你会注意到冯氏光照的第一个阶段已经应用到你的物体上了。这个物体非常暗，但由于应用了环境光照（注意光源立方体没受影响是因为我们对它使用了另一个着色器），也不是完全黑的。它看起来应该像这样：



2.3.5 漫反射光照

环境光照本身不能提供最有趣的结果，但是漫反射光照就能开始对物体产生显著的视觉影响了。漫反射光照使物体上与光线方向越接近的片段能从光源处获得更多的亮度。为了能够更好的理解漫反射光照，请看下图：



图左上方有一个光源，它所发出的光线落在物体的一个片段上。我们需要测量这个光线是以什么角度接触到这个片段的。如果光线垂直于物体表面，这束光对物体的影响会最大化（译注：更亮）。为了测量光线和片段的角度，我们使用一个叫做法向量(Normal Vector)的东西，它是垂直于片段表面的一个向量（这里以黄色箭头表示），我们在后面再讲这个东西。这两个向量之间的角度很容易就能够通过点乘计算出来。

你可能记得在[变换](#)那一节教程里，我们知道两个单位向量的夹角越小，它们点乘的结果越倾向于1。当两个向量的夹角为90度的时候，点乘会变为0。这同样适用于，越大，光对片段颜色的影响就应该越小。

注意，为了（只）得到两个向量夹角的余弦值，我们使用的是单位向量（长度为1的向量），所以我们需要确保所有的向量都是标准化的，否则点乘返回的就不仅仅是余弦值了（见[变换](#)）。

点乘返回一个标量，我们可以用它计算光线对片段颜色的影响。不同片段朝向光源的方向的不同，这些片段被照亮的情况也不同。

所以，计算漫反射光照需要什么？

- 法向量：一个垂直于顶点表面的向量。
- 定向的光线：作为光源的位置与片段的位置之间向量差的方向向量。为了计算这个光线，我们需要光的位置向量和片段的位置向量。

法向量

法向量是一个垂直于顶点表面的（单位）向量。由于顶点本身并没有表面（它只是空间中一个独立的点），我们利用它周围的顶点来计算出这个顶点的表面。我们能够使用一个小技巧，使用叉乘对立方体所有的顶点计算法向量，但是由于3D立方体不是一个复杂的形状，所以我们能够简单地把法线数据手工添加到顶点数据中。更新后的顶点数据数组可以在[这里](#)找到。试着去想象一下，这些法向量真的是垂直于立方体各个平面的表面的（一个立方体由6个平面组成）。

由于我们向顶点数组添加了额外的数据，所以我们应该更新光照的顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
...
```

现在我们已经向每个顶点添加了一个法向量并更新了顶点着色器，我们还要更新顶点属性指针。注意，灯使用同样的顶点数组作为它的顶点数据，然而灯的着色器并没有使用新添加的法向量。我们不需要更新灯的着色器或者是属性的配置，但是我们必须至少修改一下顶点属性指针来适应新的顶点数组的大小：

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);
```

我们只想使用每个顶点的前三个float，并且忽略后三个float，所以我们只需要把步长参数改成float大小的6倍就行了。

虽然对灯的着色器使用不能完全利用的顶点数据看起来不是那么高效，但这些顶点数据已经从箱子对象载入后开始就储存在GPU的内存里了，所以我们并不需要储存新数据到GPU内存中。这实际上比给灯专门分配一个新的VBO更高效了。

所有光照的计算都是在片段着色器里进行，所以我们需要将法向量由顶点着色器传递到片段着色器。我们这么做：

```
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    Normal = aNormal;
}
```

接下来，在片段着色器中定义相应的输入变量：

```
in vec3 Normal;
```

计算漫反射光照

我们现在对每个顶点都有了法向量，但是我们仍然需要光源的位置向量和片段的位置向量。由于光源的位置是一个静态变量，我们可以简单地在片段着色器中把它声明为uniform：

```
uniform vec3 lightPos;
```

然后在渲染循环中（渲染循环的外面也可以，因为它不会改变）更新uniform。我们使用在前面声明的lightPos向量作为光源位置：

```
lightingShader.setVec3("lightPos", lightPos);
```

最后，我们还需要片段的位置。我们会在世界空间中进行所有的光照计算，因此我们需要一个在世界空间中的顶点位置。我们可以通过把顶点位置属性乘以模型矩阵（不是观察和投影矩阵）来把它变换到世界空间坐标。这个在顶点着色器中很容易完成，所以我们声明一个输出变量，并计算它的世界空间坐标：

```
out vec3 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = aNormal;
}
```

最后，在片段着色器中添加相应的输入变量。

```
in vec3 FragPos;
```

现在，所有需要的变量都设置好了，我们可以在片段着色器中添加光照计算了。

我们需要做的第一件事是计算光源和片段位置之间的方向向量。前面提到，光的方向向量是光源位置向量与片段位置向量之间的向量差。你可能记得在[变换](#)教程中，我们能够简单地通过让两个向量相减的方式计算向量差。我们同样希望确保所有相关向量最后都转换为单位向量，所以我们把法线和最终的方向向量都进行标准化：

```
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
```

当计算光照时我们通常不关心一个向量的模长或它的位置，我们只关心它们的方向。所以，几乎所有的计算都使用单位向量完成，因为这简化了大部分的计算（比如点乘）。所以当进行光照计算时，确保你总是对相关向量进行标准化，来保证它们是真正地单位向量。忘记对向量进行标准化是一个十分常见的错误。

下一步，我们对`norm`和`lightDir`向量进行点乘，计算光源对当前片段实际的漫发射影响。结果值再乘以光的颜色，得到漫反射分量。两个向量之间的角度越大，漫反射分量就会越小：

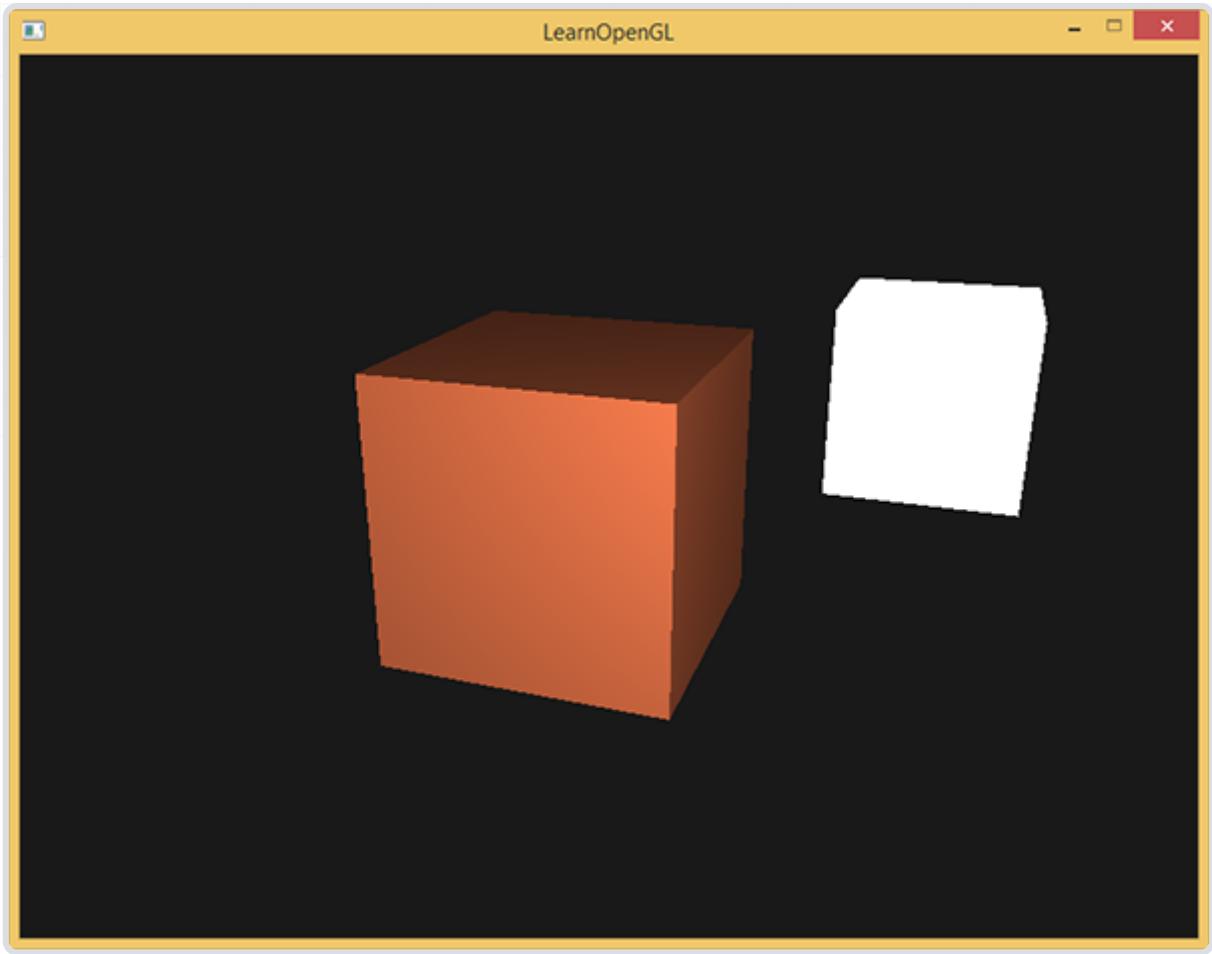
```
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;
```

如果两个向量之间的角度大于90度，点乘的结果就会变成负数，这样会导致漫反射分量变为负数。为此，我们使用`max`函数返回两个参数之间较大的参数，从而保证漫反射分量不会变成负数。负数颜色的光照是没有定义的，所以最好避免它，除非你是那种古怪的艺术家。

现在我们有了环境光分量和漫反射分量，我们把它们相加，然后把结果乘以物体的颜色，来获得片段最后的输出颜色。

```
vec3 result = (ambient + diffuse) * objectColor;
FragColor = vec4(result, 1.0);
```

如果你的应用（和着色器）编译成功了，你可能看到类似的输出：



你可以看到使用了漫反射光照，立方体看起来就真的像个立方体了。尝试在你的脑中想象一下法向量，并在立方体周围移动，注意观察法向量和光的方向向量之间的夹角越大，片段就会越暗。

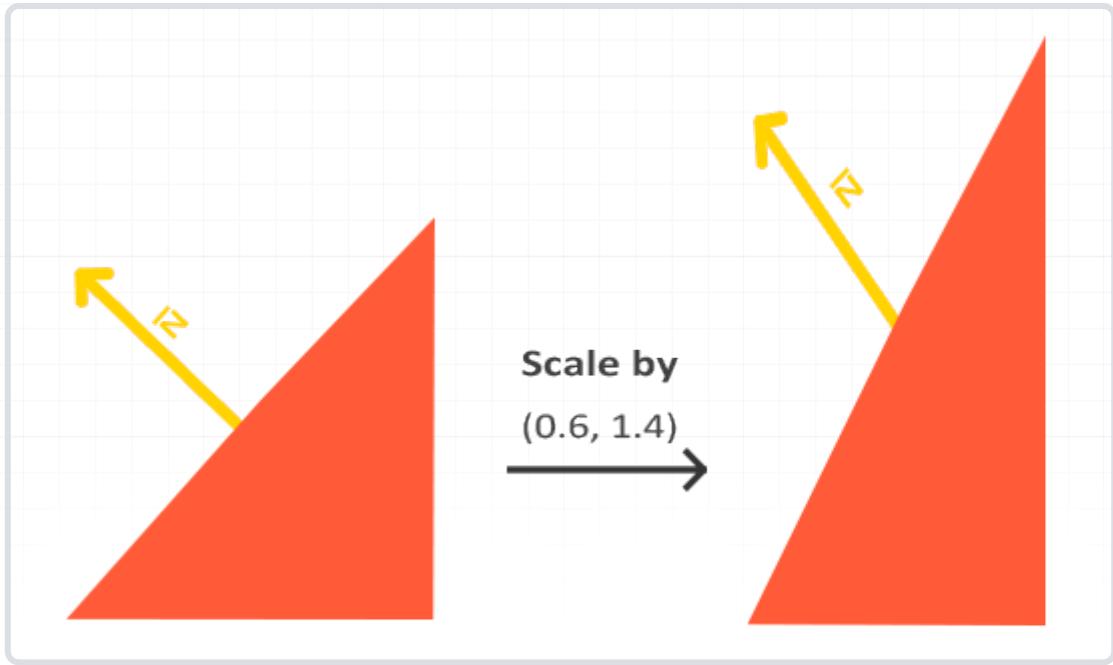
如果你在哪卡住了，可以在[这里](#)对比一下完整的源代码。

最后一件事

现在我们已经把法向量从顶点着色器传到了片段着色器。可是，目前片段着色器里的计算都是在世界空间坐标中进行的。所以，我们是不是应该把法向量也转换为世界空间坐标？基本正确，但是这不是简单地把它乘以一个模型矩阵就能搞定的。

首先，法向量只是一个方向向量，不能表达空间中的特定位置。同时，法向量没有齐次坐标（顶点位置中的w分量）。这意味着，位移不应该影响到法向量。因此，如果我们打算把法向量乘以一个模型矩阵，我们就要从矩阵中移除位移部分，只选用模型矩阵左上角 3×3 的矩阵（注意，我们也可以把法向量的w分量设置为0，再乘以 4×4 矩阵；这同样可以移除位移）。对于法向量，我们只希望对它实施缩放和旋转变换。

其次，如果模型矩阵执行了不等比缩放，顶点的改变会导致法向量不再垂直于表面了。因此，我们不能用这样的模型矩阵来变换法向量。下面的图展示了应用了不等比缩放的模型矩阵对法向量的影响：



每当我们应用一个不等比缩放时（注意：等比缩放不会破坏法线，因为法线的方向没被改变，仅仅改变了法线的长度，而这很容易通过标准化来修复），法向量就不再垂直于对应的表面了，这样光照就会被破坏。

修复这个行为的诀窍是使用一个为法向量专门定制的模型矩阵。这个矩阵称之为法线矩阵(Normal Matrix)，它使用了一些线性代数的操作来移除对法向量错误缩放的影响。如果你想知道这个矩阵是如何计算出来的，建议去阅读这个[文章](#)。

法线矩阵被定义为「模型矩阵左上角的逆矩阵的转置矩阵」。真是拗口，如果你不明白这是什么意思，别担心，我们还没有讨论逆矩阵(Inverse Matrix)和转置矩阵(Transpose Matrix)。注意，大部分的资源都会将法线矩阵定义为应用到模型-观察矩阵(Model-view Matrix)上的操作，但是由于我们只在世界空间中进行操作（不是在观察空间），我们只使用模型矩阵。

在顶点着色器中，我们可以使用`inverse`和`transpose`函数自己生成这个法线矩阵，这两个函数对所有类型矩阵都有效。注意我们还要把被处理过的矩阵强制转换为 3×3 矩阵，来保证它失去了位移属性以及能够乘以`vec3`的法向量。

```
Normal = mat3(transpose(inverse(model))) * aNormal;
```

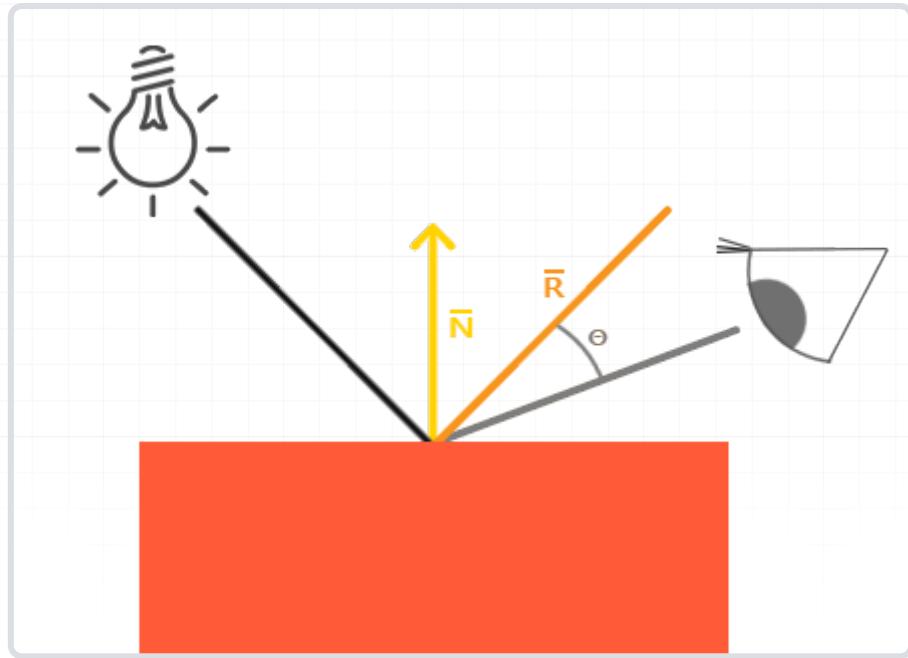
在漫反射光照部分，光照表现并没有问题，这是因为我们没有对物体本身执行任何缩放操作，所以并不是必须要使用一个法线矩阵，仅仅让模型矩阵乘以法线也可以。可是，如果你进行了不等比缩放，使用法线矩阵去乘以法向量就是必不可少的了。

即使是对着色器来说，逆矩阵也是一个开销比较大的运算，因此，只要可能就应该避免在着色器中进行逆矩阵运算，它们必须为你场景中的每个顶点都进行这样的处理。用作学习目这样做是可以的，但是对于一个对效率有要求的应用来说，在绘制之前你最好用CPU计算出法线矩阵，然后通过uniform把值传递给着色器（像模型矩阵一样）。

2.3.6 镜面光照

如果你还没被这些光照计算搞得精疲力尽，我们就再把镜面高光(Specular Highlight)加进来，这样冯氏光照才算完整。

和漫反射光照一样，镜面光照也是依据光的方向向量和物体的法向量来决定的，但是它也依赖于观察方向，例如玩家是从什么方向看着这个片段的。镜面光照是基于光的反射特性。如果我们想象物体表面像一面镜子一样，那么，无论我们从哪里去看那个表面所反射的光，镜面光照都会达到最大化。你可以从下面的图片看到效果：



我们通过反射法向量周围光的方向来计算反射向量。然后我们计算反射向量和视线方向的角度差，如果夹角越小，那么镜面光的影响就会越大。它的作用效果就是，当我们去看光被物体所反射的那个方向的时候，我们会看到一个高光。

观察向量是镜面光照附加的一个变量，我们可以使用观察者世界空间位置和片段的位置来计算它。之后，我们计算镜面光强度，用它乘以光源的颜色，再将它加上环境光和漫反射分量。

我们选择在世界空间进行光照计算，但是大多数人趋向于在观察空间进行光照计算。在观察空间计算的好处是，观察者的位置总是 $(0, 0, 0)$ ，所以这样你直接就获得了观察者位置。可是我现在学习的时候在世界空间中计算光照更符合直觉。如果你仍然希望在观察空间计算光照的话，你需要将所有相关的向量都用观察矩阵进行变换（记得也要改变法线矩阵）。

为了得到观察者的世界空间坐标，我们简单地使用摄像机对象的位置坐标代替（它当然就是观察者）。所以我们把另一个uniform添加到片段着色器，把相应的摄像机位置坐标传给片段着色器：

```
uniform vec3 viewPos;
```

```
lightingShader.setVec3("viewPos", camera.Position);
```

现在我们已经获得所有需要的变量，可以计算高光强度了。首先，我们定义一个镜面强度(Specular Intensity)变量，给镜面高光一个中等亮度颜色，让它不要产生过度的影响。

```
float specularStrength = 0.5;
```

如果我们把它设置为 $1.0f$ ，我们会得到一个非常亮的镜面光分量，这对于一个珊瑚色的立方体来说有点太多了。下一节教程中我们会讨论如何合理设置这些光照强度，以及它们是如何影响物体的。下一步，我们计算视线方向向量，和对应的沿着法线轴的反射向量：

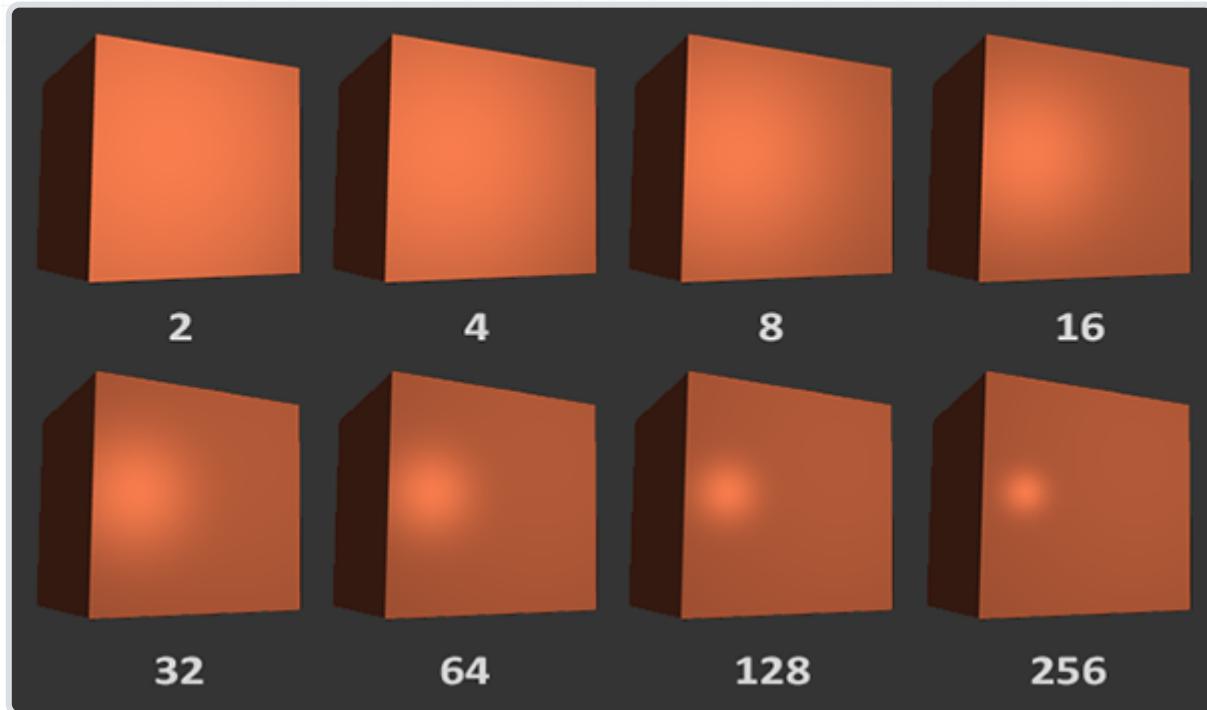
```
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
```

需要注意的是我们对 `lightDir` 向量进行了取反。`reflect` 函数要求第一个向量是从光源指向片段位置的向量，但是 `lightDir` 当前正好相反，是从片段指向光源（由先前我们计算 `lightDir` 向量时，减法的顺序决定）。为了保证我们得到正确的 `reflect` 向量，我们通过对 `lightDir` 向量取反来获得相反的方向。第二个参数要求是一个法向量，所以我们提供的是已标准化的 `norm` 向量。

剩下要做的是计算镜面分量。下面的代码完成了这件事：

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

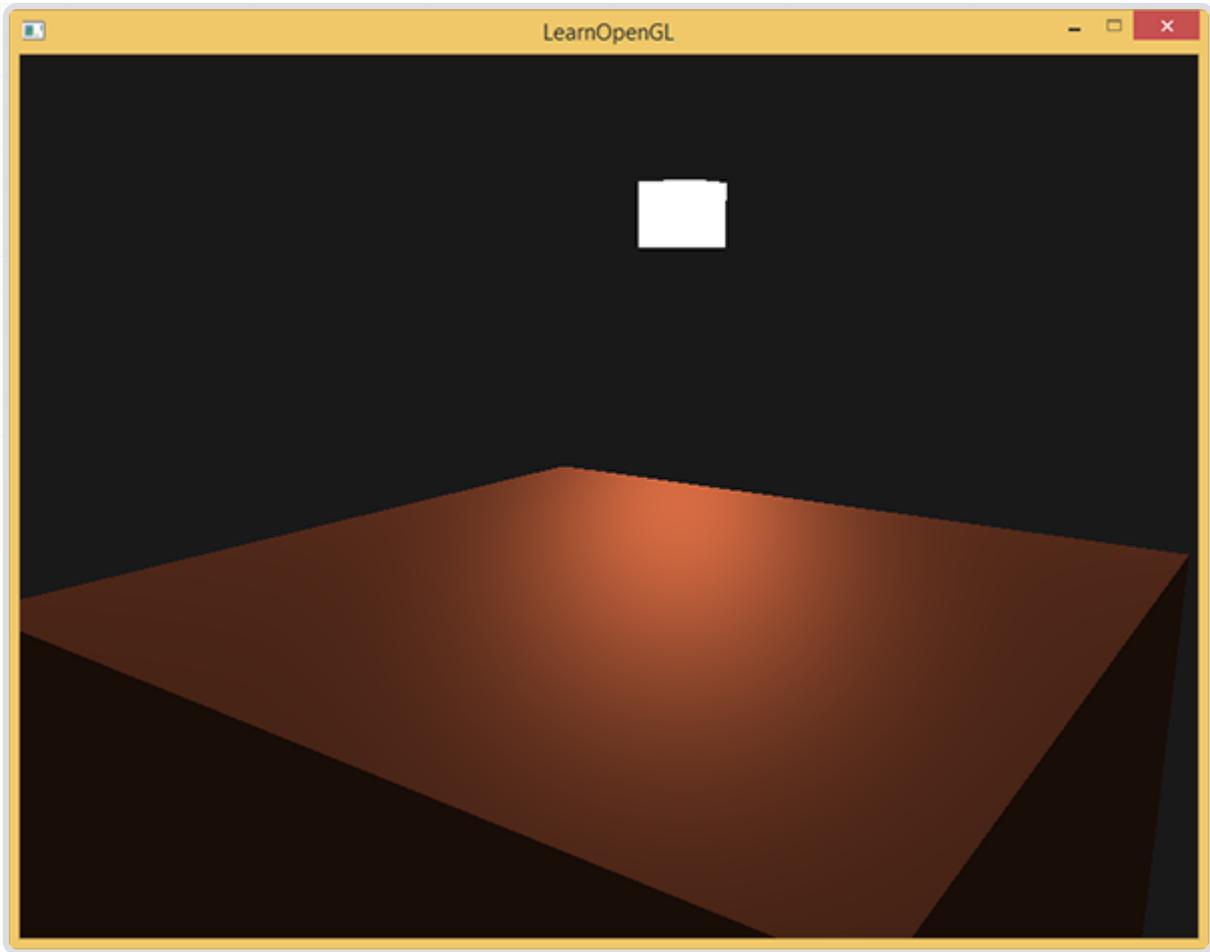
我们先计算视线方向与反射方向的点乘（并确保它不是负值），然后取它的32次幂。这个32是高光的反光度(Shininess)。一个物体的反光度越高，反射光的能力越强，散射得越少，高光点就会越小。在下面的图片里，你会看到不同反光度的视觉效果影响：



我们不希望镜面成分过于显眼，所以我们把指数保持为32。剩下的最后一件事情是把它加到环境光分量和漫反射分量里，再用结果乘以物体的颜色：

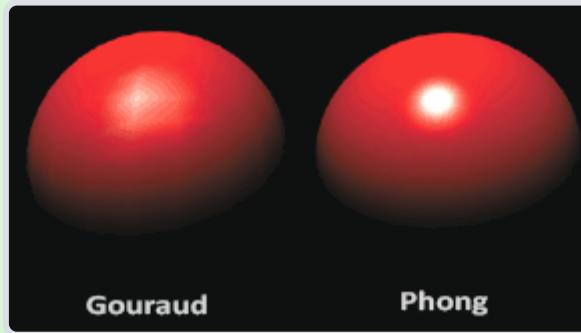
```
vec3 result = (ambient + diffuse + specular) * objectColor;
FragColor = vec4(result, 1.0);
```

我们现在为冯氏光照计算了全部的光照分量。根据你的视角，你可以看到类似下面的画面：



你可以在[这里](#)找到完整源码。

在光照着色器的早期，开发者曾经在顶点着色器中实现冯氏光照模型。在顶点着色器中做光照的优势是，相比片段来说，顶点要少得多，因此会更高效，所以（开销大的）光照计算频率会更低。然而，顶点着色器中的最终颜色值是仅仅只是那个顶点的颜色值，片段的颜色值是由插值光照颜色所得来的。结果就是这种光照看起来不会非常真实，除非使用了大量顶点。



在顶点着色器中实现的冯氏光照模型叫做**Gouraud着色**(Gouraud Shading)，而不是**冯氏着色**(Phong Shading)。记住，由于插值，这种光照看起来有点逊色。冯氏着色能产生更平滑的光照效果。

现在你应该能够看到着色器的强大之处了。只用很少的信息，着色器就能计算出光照如何影响到所有物体的片段颜色。[下一节](#)中，我们会更深入的研究光照模型，看看我们还能做些什么。

练习

- 目前，我们的光源是静止的，你可以尝试使用`sin`或`cos`函数让光源在场景中来回移动。观察光照随时间的改变能让你更容易理解冯氏光照模型。[参考解答](#)。
- 尝试使用不同的环境光、漫反射和镜面强度，观察它们是怎么影响光照效果的。同样，尝试实验一下镜面光照的反光度因子。尝试理解为什么某一个值能够有着某一种视觉输出。
- 在观察空间（而不是世界空间）中计算冯氏光照：[参考解答](#)。
- 尝试实现一个Gouraud着色（而不是冯氏着色）。如果你做对了话，立方体的光照应该会[看起来有些奇怪](#)，尝试推理为什么它会看起来这么奇怪：[参考解答](#)。

2.3.7 材质

原文 [Materials](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

在现实世界里，每个物体会对光产生不同的反应。比如说，钢看起来通常会比陶瓷花瓶更闪闪发光，木头箱子也不会像钢制箱子那样对光产生很强的反射。每个物体对镜面高光也有不同的反应。有些物体反射光的时候不会有太多的散射(Scatter)，因而产生一个较小的高光点，而有些物体则会散射很多，产生一个有着更大半径的高光点。如果我们想要在OpenGL中模拟多种类型的物体，我们必须为每个物体分别定义一个**材质**(Material)属性。

在上一节中，我们指定了一个物体和光的颜色，以及结合环境光和镜面强度分量，来定义物体的视觉输出。当描述一个物体的时候，我们可以用这三个分量来定义一个材质颜色(Material Color): 环境光照(Ambient Lighting)、漫反射光照(Diffuse Lighting)和镜面光照(Specular Lighting)。通过为每个分量指定一个颜色，我们就能够对物体的颜色输出有着精细的控制了。现在，我们再添加反光度(Shininess)这个分量到上述的三个颜色中，这就有我们需要的所有材质属性了：

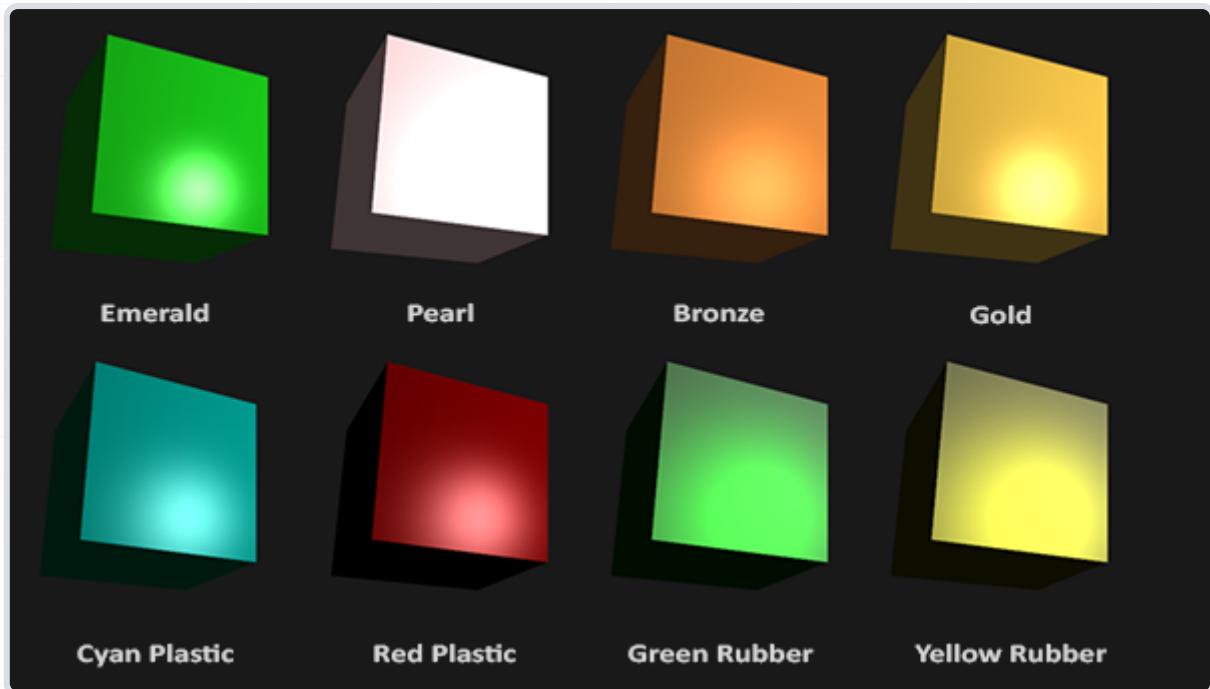
```
#version 330 core
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

uniform Material material;
```

在片段着色器中，我们创建一个结构体(Struct)来储存物体的材质属性。我们也可以把它们储存为独立的uniform值，但是作为一个结构体来储存会更有条理一些。我们首先定义结构体的布局(Layout)，然后使用刚创建的结构体为类型，简单地声明一个uniform变量。

你可以看到，我们为每个冯氏光照模型的分量都定义一个颜色向量。`ambient`材质向量定义了在环境光照下这个物体反射得是什么颜色，通常这是和物体颜色相同的颜色。`diffuse`材质向量定义了在漫反射光照下物体的颜色。（和环境光照一样）漫反射颜色也要设置为我们需要的物体颜色。`specular`材质向量设置的是镜面光照对物体的颜色影响（或者甚至可能反射一个物体特定的镜面高光颜色）。最后，`shininess`影响镜面高光的散射/半径。

这四个元素定义了一个物体的材质，通过它们我们能够模拟很多现实世界中的材质。devernay.free.fr上的一个表格展示了几种材质属性，它们模拟了现实世界中的真实材质。下面的图片展示了几种现实世界的材质对我们的立方体的影响：



可以看到，通过正确地指定一个物体的材质属性，我们对这个物体的感知也就不同了。效果非常明显，但是要想获得更真实的效果，我们最终需要更加复杂的形状，而不单单是一个立方体。在[后面的教程](#)中，我们会讨论更复杂的形状。

为一个物体赋予一款合适的材质是非常困难的，这需要大量实验和丰富的经验，所以由于不合适的材质而毁了物体的视觉质量是件经常发生的事。

让我们在着色器中实现这样的一个材质系统。

2.3.8 设置材质

我们在片段着色器中创建了一个材质结构体的uniform，所以下面我们希望修改一下光照的计算来顺应新的材质属性。由于所有材质变量都储存在结构体中，我们可以从uniform变量`material`中访问它们：

```
void main()
{
    // 环境光
    vec3 ambient = lightColor * material.ambient;

    // 漫反射
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = lightColor * (diff * material.diffuse);

    // 镜面光
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    vec3 specular = lightColor * (spec * material.specular);

    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0);
}
```

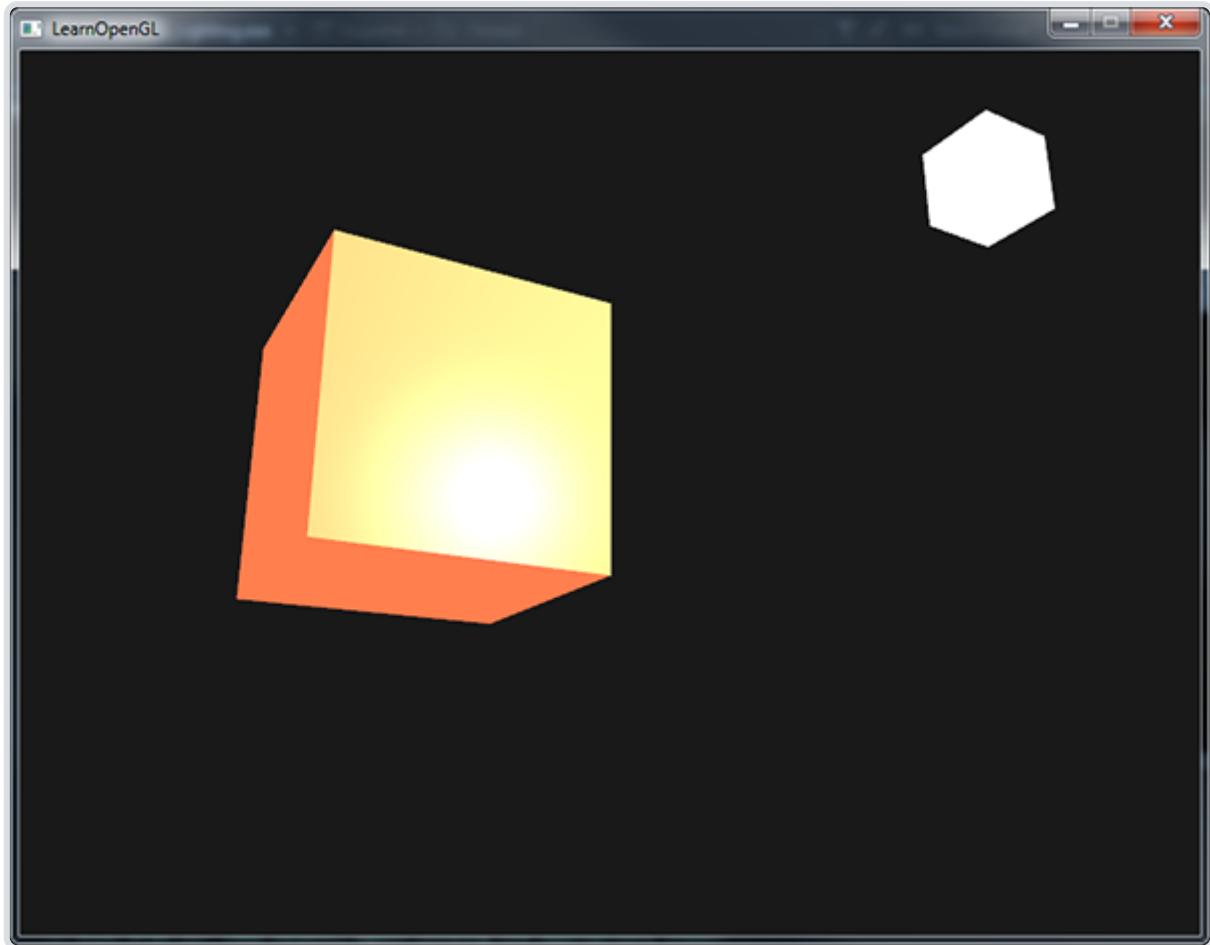
可以看到，我们现在在需要的地方访问了材质结构体中的所有属性，并且这次是根据材质的颜色来计算最终的输出颜色的。物体的每个材质属性都乘上了它们对应的光照分量。

我们现在可以在程序中设置适当的uniform，对物体设置材质了。GLSL中的结构体在设置uniform时并没有什么特别之处。结构体只是作为uniform变量的一个封装，所以如果想填充这个结构体的话，我们仍需要对每个单独的uniform进行设置，但这次要带上结构体名的前缀：

```
lightingShader.setVec3("material.ambient", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.diffuse", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.specular", 0.5f, 0.5f, 0.5f);
lightingShader.setFloat("material.shininess", 32.0f);
```

我们将环境光和漫反射分量设置成我们想要让物体所拥有的颜色，而将镜面分量设置为一个中等亮度的颜色，我们不希望镜面分量在这个物体上过于强烈。我们将反光度保持为32。现在我们能够程序中非常容易地修改物体的材质了。

运行程序，你应该会得到下面这样的结果：



但它看起来很奇怪不是吗？

光的属性

这个物体太亮了。物体过亮的原因是环境光、漫反射和镜面光这三个颜色对任何一个光源都会去全力反射。光源对环境光、漫反射和镜面光分量也具有着不同的强度。前面的教程，我们通过使用一个强度值改变环境光和镜面光强度的方式解决了这个问题。我们想做一个类似的系统，但是这次是要为每个光照分量都指定一个强度向量。如果我们假设`lightColor`是`vec3(1.0)`，代码会看起来像这样：

```
vec3 ambient = vec3(1.0) * material.ambient;
vec3 diffuse = vec3(1.0) * (diff * material.diffuse);
vec3 specular = vec3(1.0) * (spec * material.specular);
```

所以物体的每个材质属性对每一个光照分量都返回了最大的强度。对单个光源来说，这些`vec3(1.0)`值同样可以分别改变，而这通常就是我们想要的。现在，物体的环境光分量完全地影响了立方体的颜色，可是环境光分量实际上不应该对最终的颜色有这么大的影响，所以我们会将光源的环境光强度设置为一个小一点的值，从而限制环境光颜色：

```
vec3 ambient = vec3(0.1) * material.ambient;
```

我们可以用同样的方式修改光源的漫反射和镜面光强度。这和我们在[上一节](#)中所做的极为相似，你可以说我们已经创建了一些光照属性来影响每个单独的光照分量。我们希望为光照属性创建一个与材质结构体类似的结构体：

```
struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform Light light;
```

一个光源对它的`ambient`、`diffuse`和`specular`光照有着不同的强度。环境光照通常会设置为一个比较低的强度，因为我们不希望环境光颜色太过显眼。光源的漫反射分量通常设置为光所具有的颜色，通常是一个比较明亮的白色。镜面光分量通常会保持为`vec3(1.0)`，以最大强度发光。注意我们也将光源的位置添加到了结构体中。

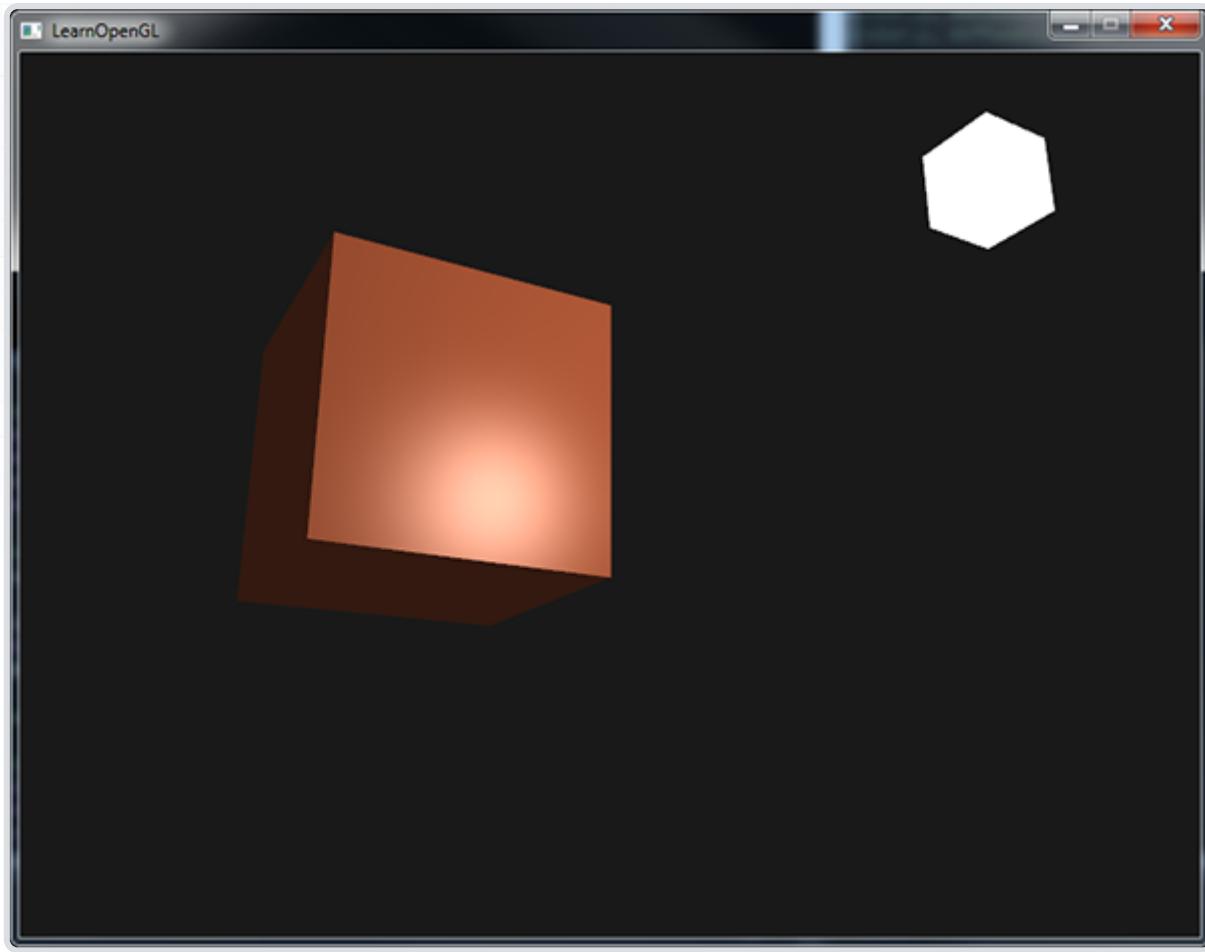
和材质uniform一样，我们需要更新片段着色器：

```
vec3 ambient = light.ambient * material.ambient;
vec3 diffuse = light.diffuse * (diff * material.diffuse);
vec3 specular = light.specular * (spec * material.specular);
```

我们接下来在程序中设置光照强度：

```
lightingShader.setVec3("light.ambient", 0.2f, 0.2f, 0.2f);
lightingShader.setVec3("light.diffuse", 0.5f, 0.5f, 0.5f); // 将光照调暗了一些以搭配场景
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);
```

现在我们调整了光照对物体材质的影响，我们应该能得到一个更类似于上一节的视觉效果。但这次我们有了对光照和物体材质的完全掌控：



改变物体的视觉效果现在变得相对容易了。让我们做点更有趣的事！

不同的光源颜色

到目前为止，我们都只对光源设置了从白到灰到黑范围内的颜色，这样只会改变物体各个分量的强度，而不是它的真正颜色。由于现在能够非常容易地访问光照的属性了，我们可以随着时间改变它们的颜色，从而获得一些非常有意思的效果。由于所有的东西都在片段着色器中配置好了，修改光源的颜色非常简单，我们能够立刻创造一些很有趣的效果：

你可以看到，不同的光照颜色能够极大地影响物体的最终颜色输出。由于光照颜色能够直接影响物体能够反射的颜色（回想[颜色](#)这一节），这对视觉输出有着显著的影响。

我们可以利用[sin](#)和[glfwGetTime](#)函数改变光源的环境光和漫反射颜色，从而很容易地让光源的颜色随着时间变化：

```
glm::vec3 lightColor;
lightColor.x = sin(glfwGetTime() * 2.0f);
lightColor.y = sin(glfwGetTime() * 0.7f);
lightColor.z = sin(glfwGetTime() * 1.3f);

glm::vec3 diffuseColor = lightColor * glm::vec3(0.5f); // 降低影响
glm::vec3 ambientColor = diffuseColor * glm::vec3(0.2f); // 很低的影响

lightingShader.setVec3("light.ambient", ambientColor);
lightingShader.setVec3("light.diffuse", diffuseColor);
```

尝试并实验一些光照和材质值，看看它们是怎样影响视觉输出的。你可以在[这里](#)找到程序的源码。

练习

- 你能像教程一开始那样，定义相应的材质来模拟现实世界的物体吗？注意[材质表格](#)中的环境光值可能与漫反射值不一样，它们没有考虑光照的强度。要想纠正这一问题，你需要将所有的光照强度都设置为 `vec3(1.0)`，这样才能得到正确的输出：[参考解答](#)，我做的是青色塑料(Cyan Plastic)的箱子。

2.3.9 光照贴图

原文 [Lighting maps](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

在[上一节](#)中，我们讨论了让每个物体都拥有自己独特的材质从而对光照做出不同的反应的方法。这样子能够很容易在一个光照的场景中给每个物体一个独特的外观，但是这仍不能对一个物体的视觉输出提供足够多的灵活性。

在上一节中，我们将整个物体的材质定义为一个整体，但现实世界中的物体通常并不只包含有一种材质，而是由多种材质所组成。想想一辆汽车：它的外壳非常有光泽，车窗会部分反射周围的环境，轮胎不会那么有光泽，所以它没有镜面高光，轮毂非常闪亮（如果你洗车了的话）。汽车同样会有漫反射和环境光颜色，它们在整个物体上也不会是一样的，汽车有着许多种不同的环境光/漫反射颜色。总之，这样的物体在不同的部件上都有不同的材质属性。

所以，上一节中的那个材质系统是肯定不够的，它只是一个最简单的模型，所以我们需要拓展之前的系统，引入漫反射和镜面光贴图(Map)。这允许我们对物体的漫反射分量（以及间接地对环境光分量，它们几乎总是一样的）和镜面光分量有着更精确的控制。

2.3.10 漫反射贴图

我们希望通过某种方式对物体的每个片段单独设置漫反射颜色。有能够让我们根据片段在物体上的位置来获取颜色值得系统吗？

这可能听起来很熟悉，而且事实上这个系统我们已经使用很长时间了。这听起来很像在[之前](#)教程中详细讨论过的纹理，而这基本就是这样：一个纹理。我们仅仅是对同样的原理使用了不同的名字：其实都是使用一张覆盖物体的图像，让我们能够逐片段索引其独立的颜色值。在光照场景中，它通常叫做一个[漫反射贴图](#)(Diffuse Map)（3D艺术家通常都这么叫它），它是一个表现了物体所有的漫反射颜色的纹理图像。

为了演示漫反射贴图，我们将会使用[下面的图片](#)，它是一个有钢边框的木箱：



在着色器中使用漫反射贴图的方法和纹理教程中是完全一样的。但这次我们会将纹理储存为Material结构体中的一个 sampler2D。我们将之前定义的 vec3 漫反射颜色向量替换为漫反射贴图。

注意 sampler2D 是所谓的**不透明类型(Opaque Type)**，也就是说我们不能将它实例化，只能通过uniform来定义它。如果我们使用除uniform以外的方法（比如函数的参数）实例化这个结构体，GLSL会抛出一些奇怪的错误。这同样也适用于任何封装了不透明类型的结构体。

我们也移除了环境光材质颜色向量，因为环境光颜色在几乎所有情况下都等于漫反射颜色，所以我们不需要将它们分开储存：

```
struct Material {
    sampler2D diffuse;
    vec3      specular;
    float     shininess;
};

...
in vec2 TexCoords;
```

如果你非常固执，仍想将环境光颜色设置为一个（漫反射值之外）不同的值，你也可以保留这个环境光的 `vec3`，但整个物体仍只能拥有一个环境光颜色。如果想要对不同片段有不同的环境光值，你需要对环境光值单独使用另外一个纹理。

注意我们将在片段着色器中再次需要纹理坐标，所以我们声明一个额外的输入变量。接下来我们只需要从纹理中采样片段的漫反射颜色值即可：

```
vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
```

不要忘记将环境光得材质颜色设置为漫反射材质颜色同样的值。

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
```

这就是使用漫反射贴图的全部步骤了。你可以看到，这并不是什么新的东西，但这能够极大地提高视觉品质。为了让它正常工作，我们还需要使用纹理坐标更新顶点数据，将它们作为顶点属性传递到片段着色器，加载材质并绑定材质到合适的纹理单元。

更新后的顶点数据可以[在这里](#)找到。顶点数据现在包含了顶点位置、法向量和立方体顶点处的纹理坐标。让我们更新顶点着色器来以顶点属性的形式接受纹理坐标，并将它们传递到片段着色器中：

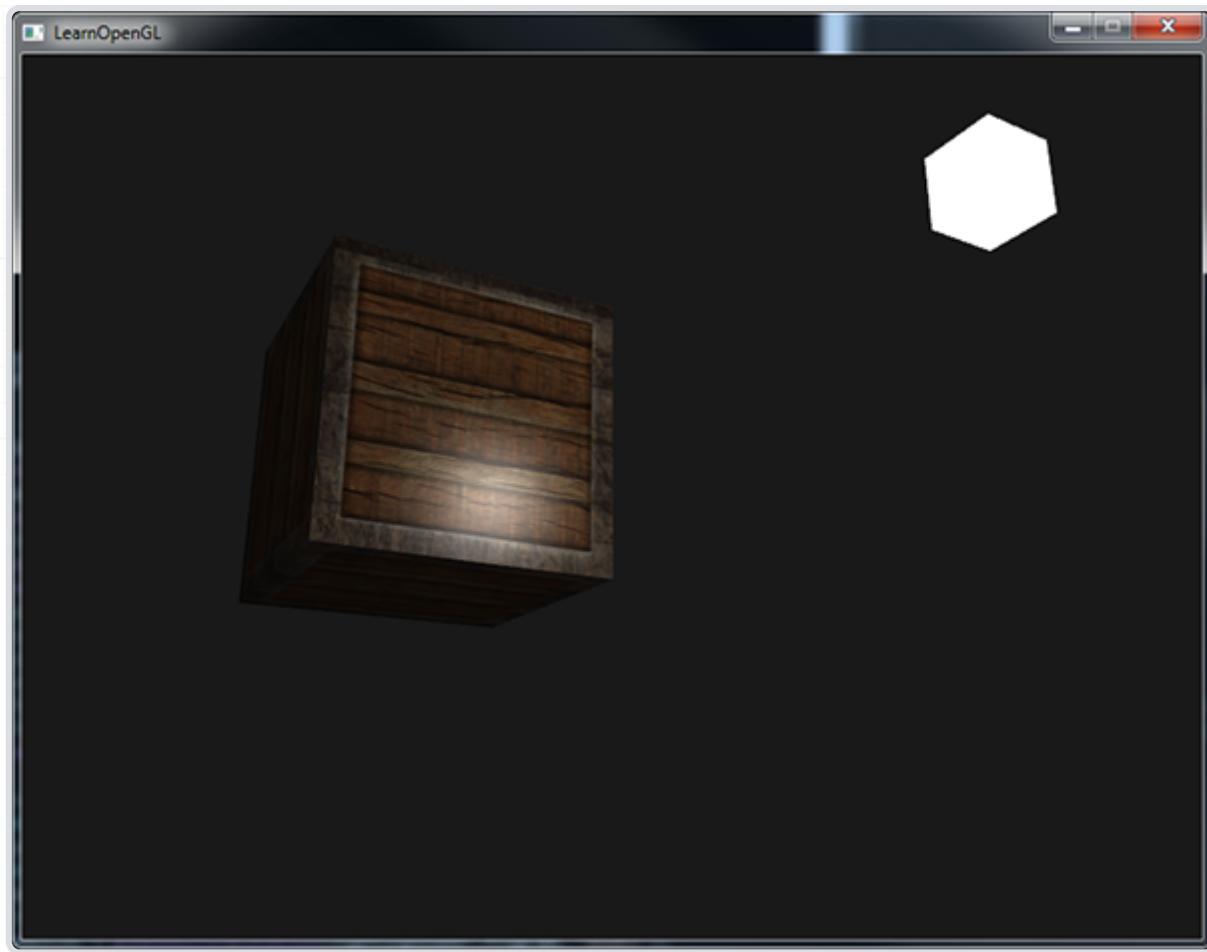
```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
...
out vec2 TexCoords;

void main()
{
    ...
    TexCoords = aTexCoords;
}
```

记得去更新两个VAO的顶点属性指针来匹配新的顶点数据，并加载箱子图像为一个纹理。在绘制箱子之前，我们希望将要用的纹理单元赋值到`material.diffuse`这个uniform采样器，并绑定箱子的纹理到这个纹理单元：

```
lightingShader.setInt("material.diffuse", 0);
...
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, diffuseMap);
```

使用了漫反射贴图之后，细节再一次得到惊人的提升，这次箱子有了光照开始闪闪发光（字面意思也是）了。你的箱子看起来可能像这样：

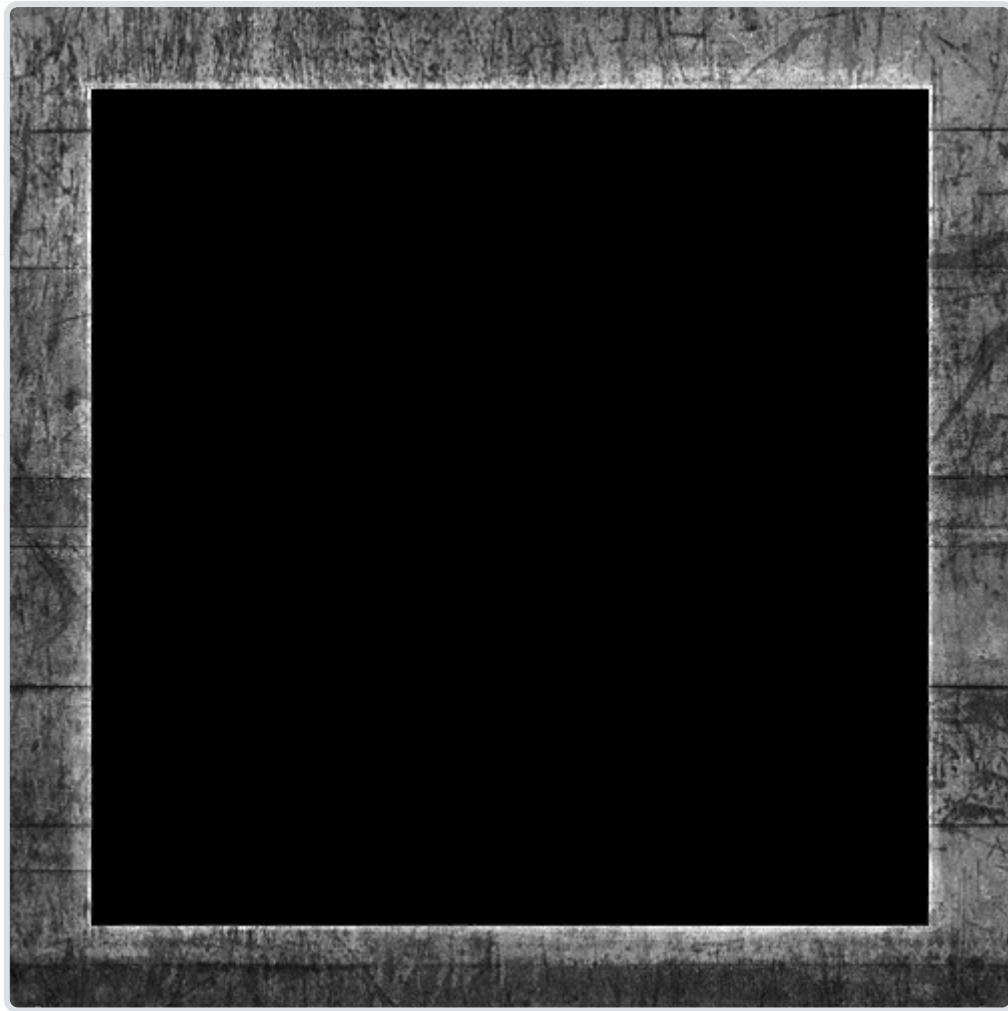


你可以在[这里](#)找到程序的全部代码。

2.3.11 镜面光贴图

你可能会注意到，镜面高光看起来有些奇怪，因为我们的物体大部分都是木头，我们知道木头不应该有这么强的镜面高光的。我们可以将物体的镜面光材质设置为 `vec3(0.0)` 来解决这个问题，但这也意味着箱子钢制的边框将不再能够显示镜面高光了，我们知道钢铁应该是有一些镜面高光的。所以，我们想要让物体的某些部分以不同的强度显示镜面高光。这个问题看起来和漫反射贴图非常相似。是巧合吗？我想不是。

我们同样可以使用一个专门用于镜面高光的纹理贴图。这也就意味着我们需要生成一个黑白的（如果你想得话也可以是彩色的）纹理，来定义物体每部分的镜面光强度。下面是一个[镜面光贴图](#)(Specular Map)的例子：



镜面高光的强度可以通过图像每个像素的亮度来获取。镜面光贴图上的每个像素都可以由一个颜色向量来表示，比如说黑色代表颜色向量 `vec3(0.0)`，灰色代表颜色向量 `vec3(0.5)`。在片段着色器中，我们接下来会取样对应的颜色值并将它乘以光源的镜面强度。一个像素越「白」，乘积就会越大，物体的镜面光分量就会越亮。

由于箱子大部分都由木头所组成，而且木头材质应该没有镜面高光，所以漫反射纹理的整个木头部分全部都转换成了黑色。箱子钢制边框的镜面光强度是有细微变化的，钢铁本身会比较容易受到镜面高光的影响，而裂缝则不会。

从实际角度来说，木头其实也有镜面高光，尽管它的反光度(Shininess)很小（更多的光被散射），影响也比较小，但是为了教学目的，我们可以假设木头不会对镜面光有任何反应。

使用Photoshop或Gimp之类的工具，将漫反射纹理转换为镜面光纹理还是比较容易的，只需要剪切掉一些部分，将图像转换为黑白的，并增加亮度/对比度就好了。

采样镜面光贴图

镜面光贴图和其他的纹理非常类似，所以代码也和漫反射贴图的代码很类似。记得要保证正确地加载图像并生成一个纹理对象。由于我们正在同一个片段着色器中使用另一个纹理采样器，我们必须对镜面光贴图使用一个不同的纹理单元（见[纹理](#)），所以我们在渲染之前先把它绑定到合适的纹理单元上：

```
lightingShader.setInt("material.specular", 1);
...
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, specularMap);
```

接下来更新片段着色器的材质属性，让其接受一个 `sampler2D` 而不是 `vec3` 作为镜面光分量：

```
struct Material {
    sampler2D diffuse;
    sampler2D specular;
    float      shininess;
};
```

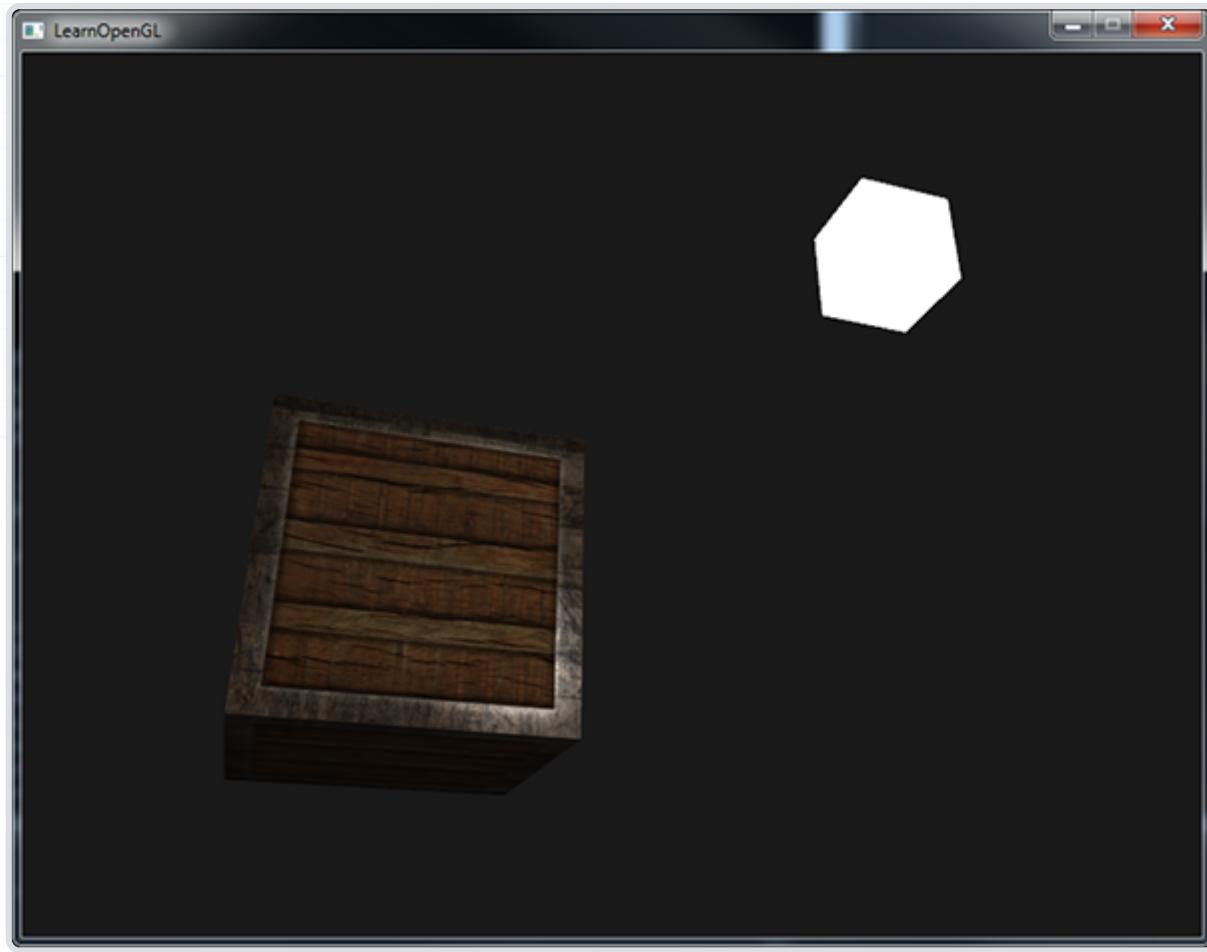
最后我们希望采样镜面光贴图，来获取片段所对应的镜面光强度：

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
FragColor = vec4(ambient + diffuse + specular, 1.0);
```

通过使用镜面光贴图我们可以对物体设置大量的细节，比如物体的哪些部分需要有闪闪发光的属性，我们甚至可以设置它们对应的强度。镜面光贴图能够在漫反射贴图之上给予我们更高一层的控制。

如果你想另辟蹊径，你也可以在镜面光贴图中使用真正的颜色，不仅设置每个片段的镜面光强度，还设置了镜面高光的颜色。从现实角度来说，镜面高光的颜色大部分（甚至全部）都是由光源本身所决定的，所以这样并不能生成非常真实的视觉效果（这也是为什么图像通常是黑白的，我们只关心强度）。

如果你现在运行程序的话，你可以清楚地看到箱子的材质现在和真实的钢制边框箱子非常类似了：



你可以在[这里](#)找到程序的全部源码。

通过使用漫反射和镜面光贴图，我们可以给相对简单的物体添加大量的细节。我们甚至可以使用法线/凹凸贴图(Normal/Bump Map)或者反射贴图(Reflection Map)给物体添加更多的细节，但这些将会留到之后的教程中。把你的箱子给你的朋友或者家人看看，并且坚信我们的箱子有一天会比现在更加漂亮！

练习

- 调整光源的环境光、漫反射和镜面光向量，看看它们如何影响箱子的视觉输出。
- 尝试在片段着色器中反转镜面光贴图的颜色值，让木头显示镜面高光而钢制边缘不反光（由于钢制边缘中有一些裂缝，边缘仍会显示一些镜面高光，虽然强度会小很多）：[参考解答](#)
- 使用漫反射贴图创建一个彩色而不是黑白的镜面光贴图，看看结果看起来并不是那么真实了。如果你不会生成的话，可以使用这张[彩色的镜面光贴图：最终效果](#)
- 添加一个叫做放射光贴图(Emission Map)的东西，它是一个储存了每个片段的发光值(Emission Value)的贴图。发光值是一个包含（假设）光源的物体发光(Emit)时可能显现的颜色，这样的话物体就能够忽略光照条件进行发光(Glow)。游戏中某个物体在发光的时候，你通常看到的就是放射光贴图（比如[机器人的手](#)，或是[箱子上的灯带](#)）。将[这个纹理](#)（作者为 creativesam）作为放射光贴图添加到箱子上，产生这些字母都在发光的效果：[参考解答](#), [最终效果](#)

2.3.12 投光物

原文 [Light casters](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

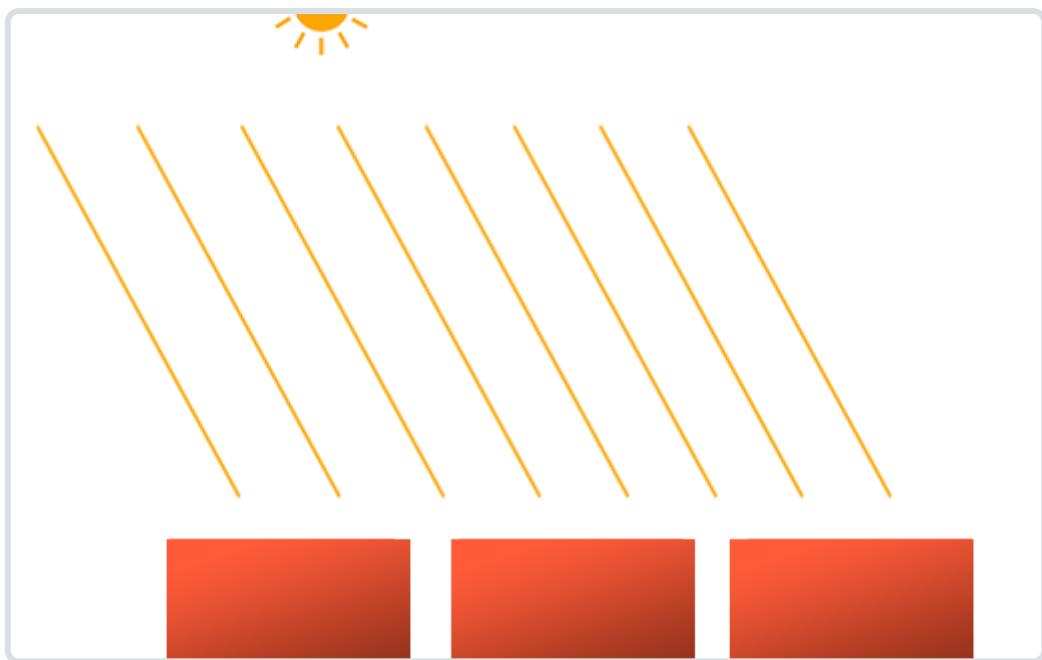
我们目前使用的光照都来自于空间中的一个点。它能给我们不错的效果，但现实世界中，我们有很多种类的光照，每种的表现都不同。将光投射(Cast)到物体的光源叫做**投光物**(Light Caster)。在这一节中，我们将会讨论几种不同类型的投光物。学会模拟不同种类的光源是又一个能够进一步丰富场景的工具。

我们首先将会讨论定向光(Directional Light)，接下来是点光源(Point Light)，它们是我们之前学习的光源的拓展，最后我们将会讨论聚光(Spotlight)。在[下一节](#)中我们将讨论如何将这些不同种类的光照类型整合到一个场景之中。

2.3.13 平行光

当一个光源处于很远的地方时，来自光源的每条光线就会近似于互相平行。不论物体和/或者观察者的位置，看起来好像所有的光都来自于同一个方向。当我们使用一个假设光源处于无限远处的模型时，它就被称为**定向光**，因为它的所有光线都有着相同的方向，它与光源的位置是没有关系的。

定向光非常好的一个例子就是太阳。太阳距离我们并不是无限远，但它已经远到在光照计算中可以把它视为无限远了。所以来自太阳的所有光线将被模拟为平行光线，我们可以在下图看到：



因为所有的光线都是平行的，所以物体与光源的相对位置是不重要的，因为对场景中每一个物体光的方向都是一致的。由于光的位置向量保持一致，场景中每个物体的光照计算将会是类似的。

我们可以定义一个光线方向向量而不是位置向量来模拟一个定向光。着色器的计算基本保持不变，但这次我们将直接使用光的 `direction` 向量而不是通过 `direction` 来计算 `lightDir` 向量。

```
struct Light {
    // vec3 position; // 使用定向光就不再需要了
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

...
void main()
{
    vec3 lightDir = normalize(-light.direction);
    ...
}
```

注意我们首先对 `light.direction` 向量取反。我们目前使用的光照计算需求一个从片段至光源的光线方向，但人们更习惯定义定向光为一个从光源出发的全局方向。所以我们需要对全局光照方向向量取反来改变它的方向，它现在是一个指向光源的方向向量了。而且，记得对向量进行标准化，假设输入向量为一个单位向量是很不明智的。

最终的 `lightDir` 向量将和以前一样用在漫反射和镜面光计算中。

为了清楚地展示定向光对多个物体具有相同的影响，我们将会再次使用 [坐标系统](#) 章节最后的那个箱子派对的场景。如果你错过了派对，我们先定义了十个不同的 [箱子位置](#)，并对每个箱子都生成了一个不同的模型矩阵，每个模型矩阵都包含了对应的局部-世界坐标变换：

```
for(unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model;
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    lightingShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

同时，不要忘记定义光源的方向（注意我们将方向定义为从光源出发的方向，你可以很容易看到光的方向朝下）。

```
lightingShader.setVec3("light.direction", -0.2f, -1.0f, -0.3f);
```

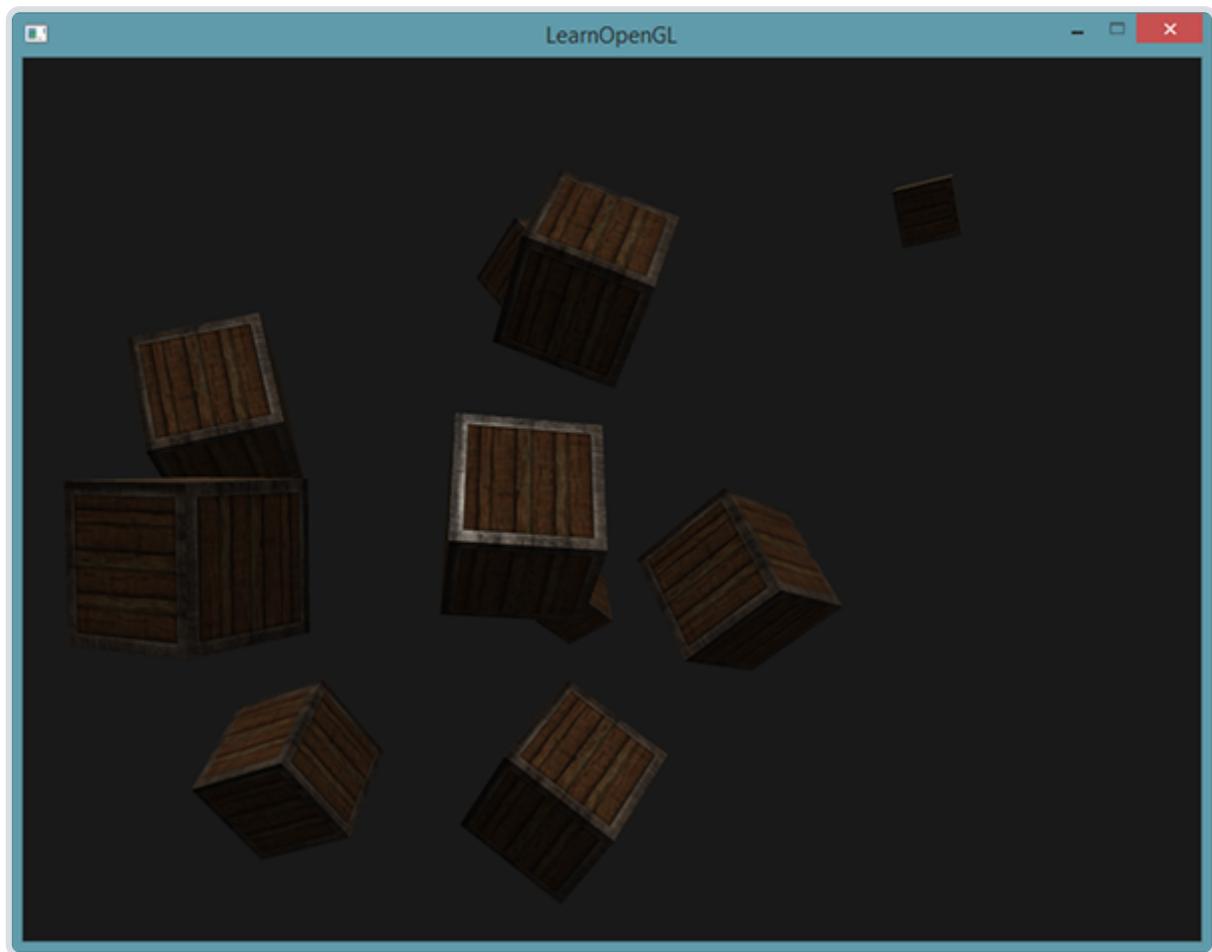
我们一直将光的位置和位置向量定义为 `vec3`，但一些人会喜欢将所有的向量都定义为 `vec4`。当我们把位置向量定义为一个 `vec4` 时，很重要的一点是要将w分量设置为1.0，这样变换和投影才能正确应用。然而，当我们定义一个方向向量为 `vec4` 的时候，我们不想让位移有任何的效果（因为它仅仅代表的是方向），所以我们将w分量设置为0.0。

方向向量就会像这样来表示：`vec4(0.2f, 1.0f, 0.3f, 0.0f)`。这也能够作为一个快速检测光照类型的工具：你可以检测w分量是否等于1.0，来检测它是否是光的位置向量；w分量等于0.0，则它是光的方向向量，这样就能根据这个来调整光照计算了：

```
if(lightVector.w == 0.0) // 注意浮点数据类型的误差
    // 执行定向光照计算
else if(lightVector.w == 1.0)
    // 根据光源的位置做光照计算（与上一节一样）
```

你知道吗：这正是旧OpenGL（固定函数式）决定光源是定向光还是位置光源(Positional Light Source)的方法，并根据它来调整光照。

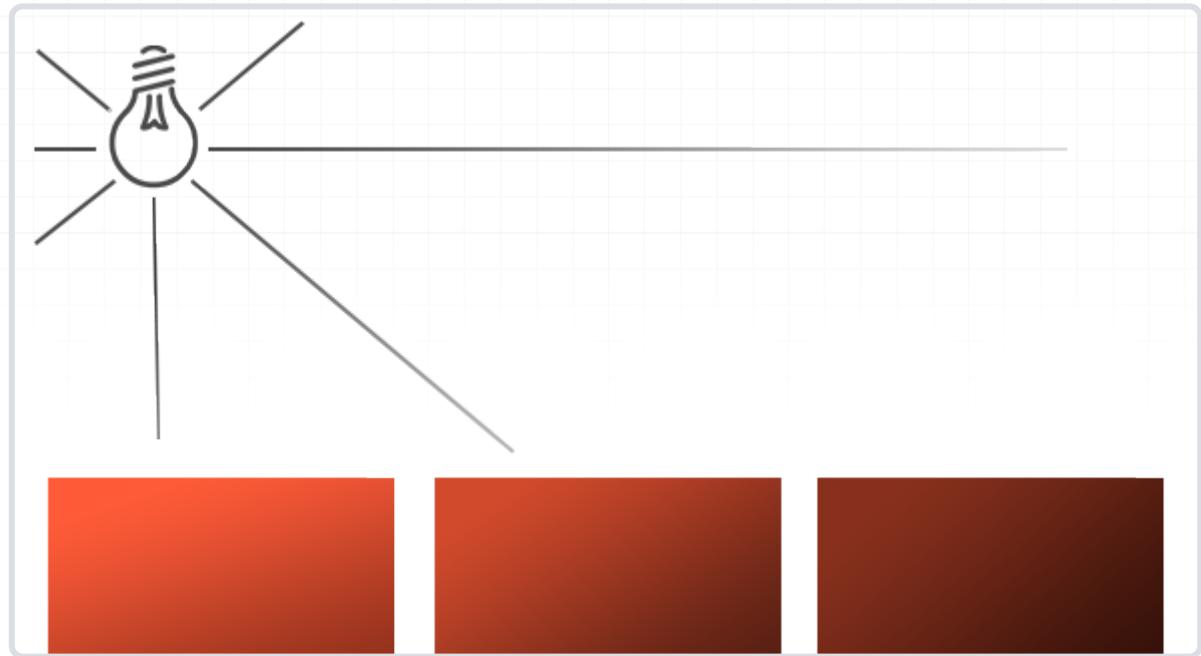
如果你现在编译程序，在场景中自由移动，你就可以看到好像有一个太阳一样的光源对所有的物体投光。你能注意到漫反射和镜面光分量的反应都好像在天空中有一个光源的感觉吗？它会看起来像这样：



你可以在[这里](#)找到程序的所有代码。

2.3.14 点光源

定向光对于照亮整个场景的全局光源是非常棒的，但除了定向光之外我们也需要一些分散在场景中的点光源(Point Light)。点光源是处于世界中某一个位置的光源，它会朝着所有方向发光，但光线会随着距离逐渐衰减。想象作为投光物的灯泡和火把，它们都是点光源。



在之前的教程中，我们一直都在使用一个（简化的）点光源。我们在给定位置有一个光源，它会从它的光源位置开始朝着所有方向散射光线。然而，我们定义的光源模拟的是永远不会衰减的光线，这看起来像是光源亮度非常的强。在大部分的3D模拟中，我们都希望模拟的光源仅照亮光源附近的区域而不是整个场景。

如果你将10个箱子加入到上一节光照场景中，你会注意到在最后面的箱子和在灯面前的箱子都以相同的强度被照亮，并没有定义一个公式来将光随距离衰减。我们希望在后排的箱子与前排的箱子相比仅仅是被轻微地照亮。

衰减

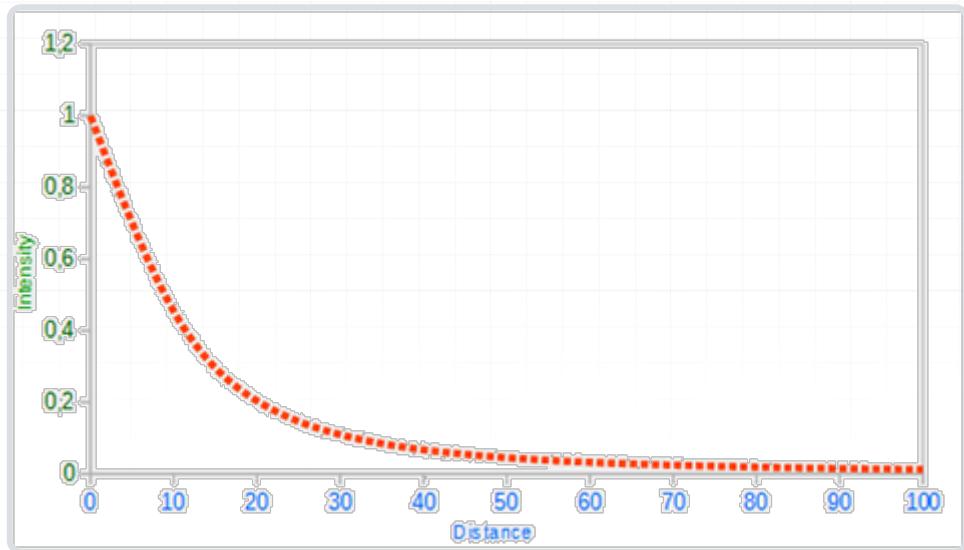
随着光线传播距离的增长逐渐削减光的强度通常叫做衰减(Attenuation)。随距离减少光强度的一种方式是使用一个线性方程。这样的方程能够随着距离的增长线性地减少光的强度，从而让远处的物体更暗。然而，这样的线性方程通常会看起来比较假。在现实世界中，灯在近处通常会非常亮，但随着距离的增加光源的亮度一开始会下降非常快，但在远处时剩余的光强度就会下降的非常缓慢了。所以，我们需要一个不同的公式来减少光的强度。

幸运的是一些聪明的人已经帮我们解决了这个问题。下面这个公式根据片段距光源的距离计算了衰减值，之后我们会将它乘以光的强度向量：

在这里代表了片段距光源的距离。接下来为了计算衰减值，我们定义3个（可配置的）项：常数项、一次项和二次项。

- 常数项通常保持为1.0，它的主要作用是保证分母永远不会比1小，否则的话在某些距离上它反而会增加强度，这肯定不是我们想要的效果。
- 一次项会与距离值相乘，以线性的方式减少强度。
- 二次项会与距离的平方相乘，让光源以二次递减的方式减少强度。二次项在距离比较小的时候影响会比一次项小很多，但当距离值比较大的时候它就会比一次项更大了。

由于二次项的存在，光线会在大部分时候以线性的方式衰退，直到距离变得足够大，让二次项超过一次项，光的强度会以更快的速度下降。这样的结果就是，光在近距离时亮度很高，但随着距离变远亮度迅速降低，最后会以更慢的速度减少亮度。下面这张图显示了在100的距离内衰减的效果：



你可以看到光在近距离的时候有着最高的强度，但随着距离增长，它的强度明显减弱，并缓慢地在距离大约100的时候强度接近0。这正是我们想要的。

选择正确的值

但是，该对这三个项设置什么值呢？正确地设定它们的值取决于很多因素：环境、希望光覆盖的距离、光的类型等。在大多数情况下，这都是经验的问题，以及适量的调整。下面这个表格显示了模拟一个（大概）真实的，覆盖特定半径（距离）的光源时，这些项可能取的一些值。第一列指定的是在给定的三项时光所能覆盖的距离。这些值是大多数光源很好的起始点，它们由[Ogre3D的Wiki](#)所提供：

距离	常数项	一次项	二次项
7	1.0	0.7	1.8
13	1.0	0.35	0.44
20	1.0	0.22	0.20
32	1.0	0.14	0.07
50	1.0	0.09	0.032
65	1.0	0.07	0.017
100	1.0	0.045	0.0075
160	1.0	0.027	0.0028
200	1.0	0.022	0.0019
325	1.0	0.014	0.0007
600	1.0	0.007	0.0002
3250	1.0	0.0014	0.000007

你可以看到，常数项在所有的情况下都是1.0。一次项为了覆盖更远的距离通常都很小，二次项甚至更小。尝试对这些值进行实验，看看它们在你的实现中有什么效果。在我们的环境中，32到100的距离对大多数的光源都足够了。

实现衰减

为了实现衰减，在片段着色器中我们还需要三个额外的值：也就是公式中的常数项、一次项和二次项。它们最好储存在之前定义的 `Light` 结构体中。注意我们使用上一节中计算 `lightDir` 的方法，而不是上面定向光部分的。

```
struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    float constant;
    float linear;
    float quadratic;
};
```

然后我们将在OpenGL中设置这些项：我们希望光源能够覆盖50的距离，所以我们会使用表格中对应的常数项、一次项和二次项：

```
lightingShader.setFloat("light.constant", 1.0f);
lightingShader.setFloat("light.linear", 0.09f);
lightingShader.setFloat("light.quadratic", 0.032f);
```

在片段着色器中实现衰减还是比较直接的：我们根据公式计算衰减值，之后再分别乘以环境光、漫反射和镜面光分量。

我们仍需要公式中距光源的距离，还记得我们是怎么计算一个向量的长度的吗？我们可以通过获取片段和光源之间的向量差，并获取结果向量的长度作为距离项。我们可以使用GLSL内建的 `length` 函数来完成这一点：

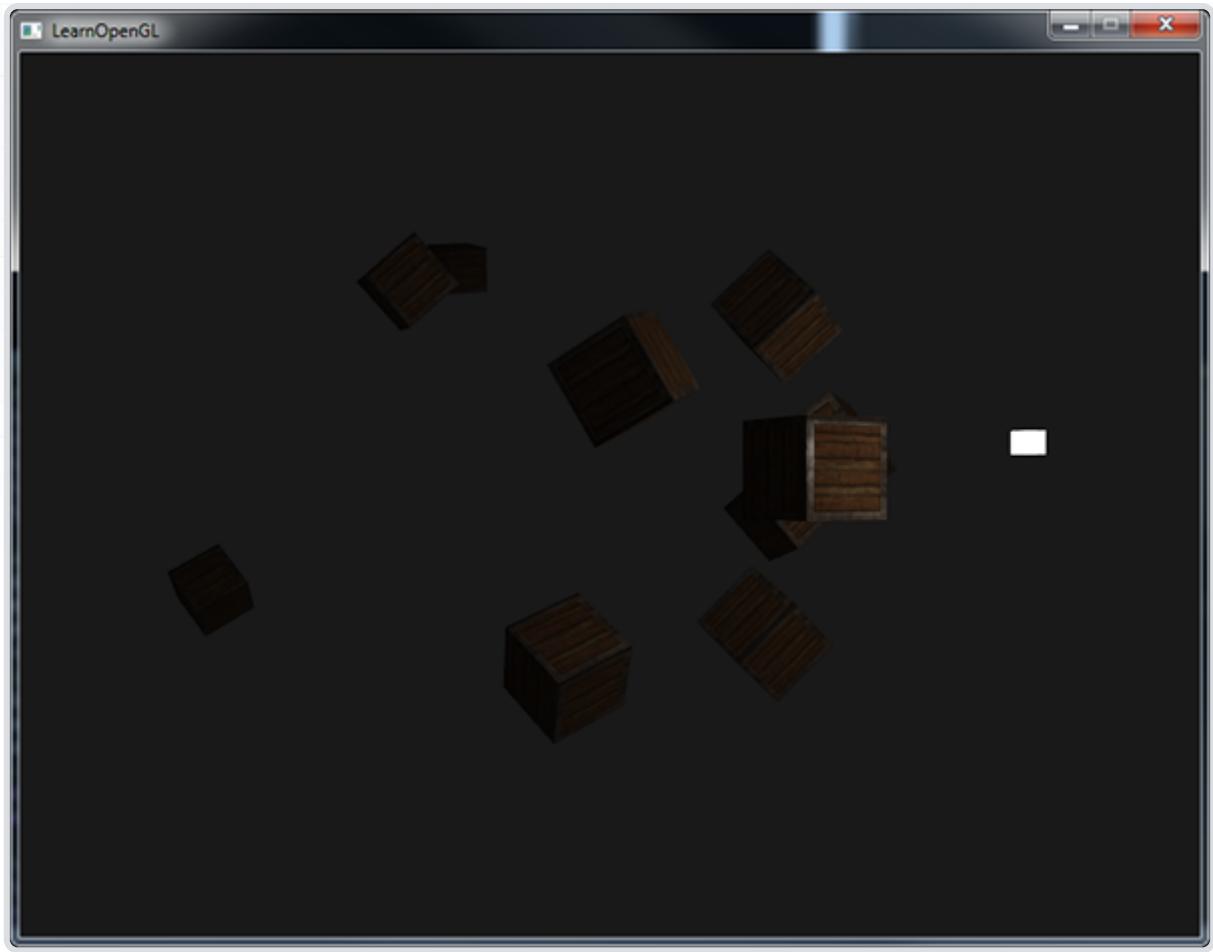
```
float distance = length(light.position - FragPos);
float attenuation = 1.0 / (light.constant + light.linear * distance +
    light.quadratic * (distance * distance));
```

接下来，我们将包含这个衰减值到光照计算中，将它分别乘以环境光、漫反射和镜面光颜色。

我们可以将环境光分量保持不变，让环境光照不会随着距离减少，但是如果我们将多个光源，所有的环境光分量将会开始叠加，所以在这种情况下我们也希望衰减环境光照。简单实验一下，看看什么才能在你的环境中效果最好。

```
ambient *= attenuation;
diffuse *= attenuation;
specular *= attenuation;
```

如果你运行程序的话，你会获得这样的结果：



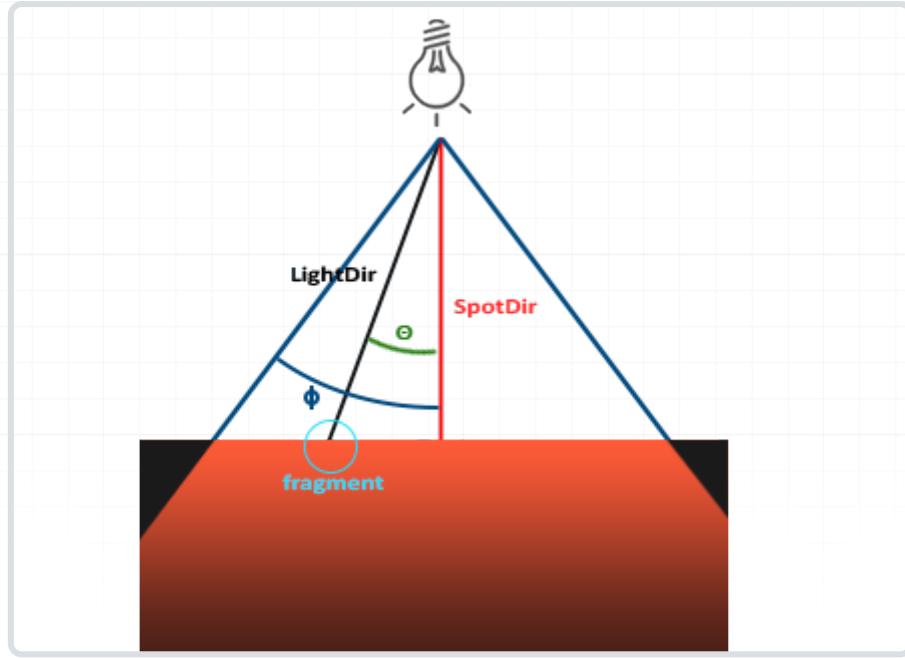
你可以看到，只有前排的箱子被照亮的，距离最近的箱子是最亮的。后排的箱子一点都没有照亮，因为它们离光源实在是太远了。你可以在[这里](#)找到程序的代码。

点光源就是一个能够配置位置和衰减的光源。它是我们光照工具箱中的又一个光照类型。

2.3.15 聚光

我们要讨论的最后一类光是聚光(Spotlight)。聚光是位于环境中某个位置的光源，它只朝一个特定方向而不是所有方向照射光线。这样的结果就是只有在聚光方向的特定半径内的物体才会被照亮，其它的物体都会保持黑暗。聚光很好的例子就是路灯或手电筒。

OpenGL中聚光是用一个世界空间位置、一个方向和一个切光角(Cutoff Angle)来表示的，切光角指定了聚光的半径（译注：是圆锥的半径不是距光源距离那个半径）。对于每个片段，我们计算片段是否位于聚光的切光方向之间（也就是在锥形内），如果是的话，我们就会相应地照亮片段。下面这张图会让你明白聚光是如何工作的：



- `LightDir`：从片段指向光源的向量。
- `SpotDir`：聚光所指向的方向。
- `Phi`：指定了聚光半径的切光角。落在这个角度之外的物体都不会被这个聚光所照亮。
- `Theta`：`LightDir`向量和`SpotDir`向量之间的夹角。在聚光内部的话值应该比值小。

所以我们要做的就是计算`LightDir`向量和`SpotDir`向量之间的点积（还记得它会返回两个单位向量夹角的余弦值吗？），并将它与切光角值对比。你现在应该了解聚光究竟是什么了，下面我们将以手电筒的形式创建一个聚光。

手电筒

手电筒(Flashlight)是一个位于观察者位置的聚光，通常它都会瞄准玩家视角的正前方。基本上说，手电筒就是普通的聚光，但它的位置和方向会随着玩家的位置和朝向不断更新。

所以，在片段着色器中我们需要的值有聚光的位置向量（来计算光的方向向量）、聚光的方向向量和一个切光角。我们可以将它们储存在`Light`结构体中：

```
struct Light {
    vec3 position;
    vec3 direction;
    float cutOff;
    ...
};
```

接下来我们将合适的值传到着色器中：

```
lightingShader.setVec3("light.position", camera.Position);
lightingShader.setVec3("light.direction", camera.Front);
lightingShader.setFloat("light.cutOff", glm::cos(glm::radians(12.5f)));
```

你可以看到，我们并没有给切光角设置一个角度值，反而是用角度值计算了一个余弦值，将余弦结果传递到片段着色器中。这样做的原因是在片段着色器中，我们会计算 `LightDir` 和 `SpotDir` 向量的点积，这个点积返回的将是一个余弦值而不是角度值，所以我们不能直接使用角度值和余弦值进行比较。为了获取角度值我们需要计算点积结果的反余弦，这是一个开销很大的计算。所以为了节约一点点性能开销，我们将会计算切光角对应的余弦值，并将它的结果传入片段着色器中。由于这两个角度现在都由余弦角来表示了，我们可以直接对它们进行比较而不用进行任何开销高昂的计算。

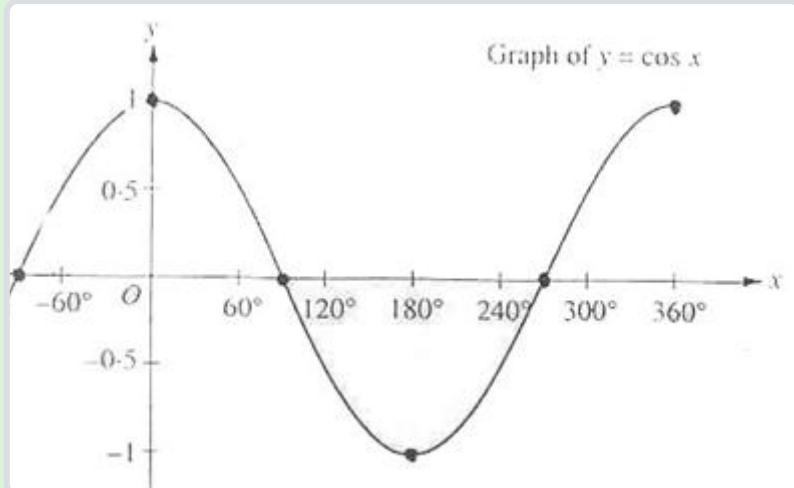
接下来就是计算值，并将它和切光角对比，来决定是否在聚光的内部：

```
float theta = dot(lightDir, normalize(-light.direction));

if(theta > light.cutOff)
{
    // 执行光照计算
}
else // 否则，使用环境光，让场景在聚光之外时不至于完全黑暗
    color = vec4(light.ambient * vec3(texture(material.diffuse, TexCoords)), 1.0);
```

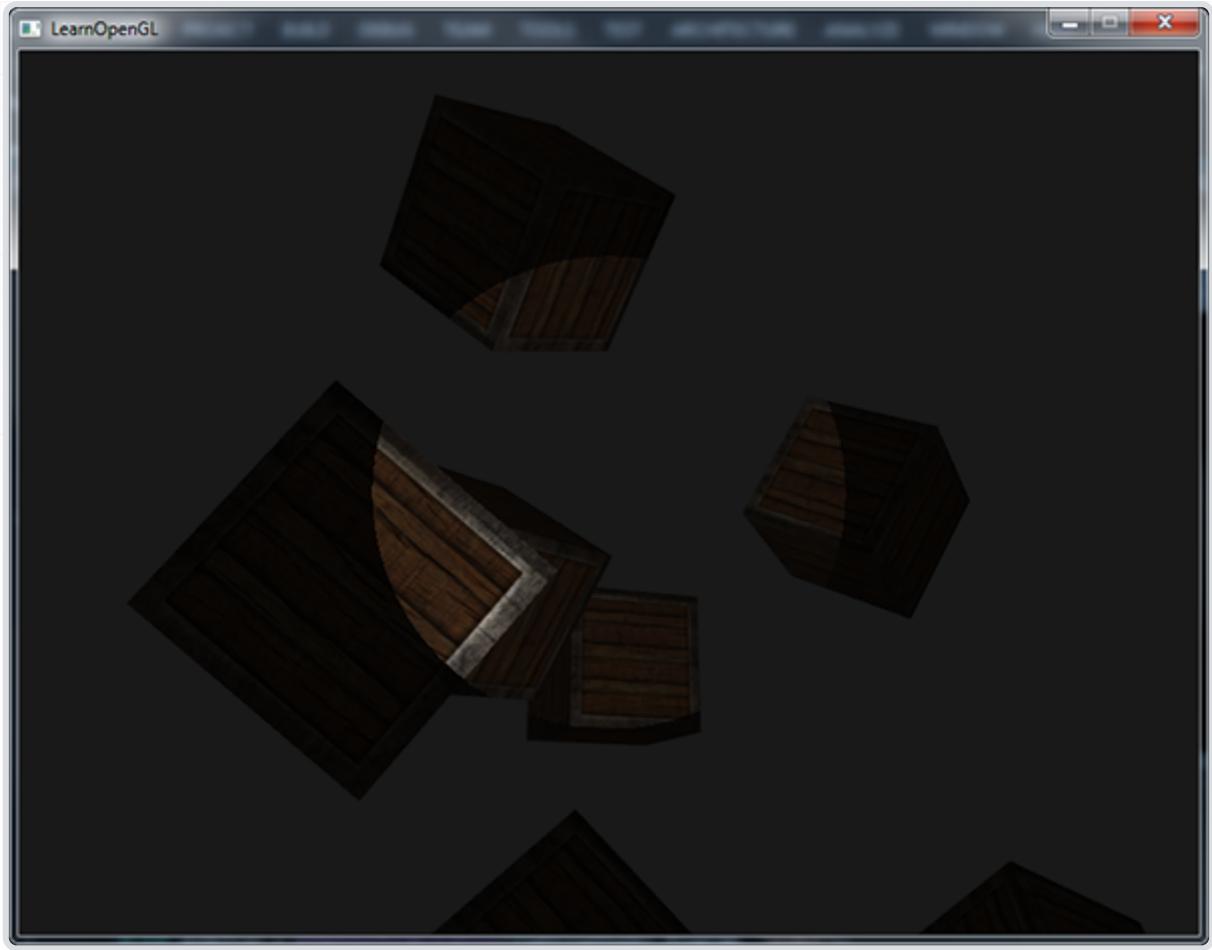
我们首先计算了 `lightDir` 和取反的 `direction` 向量（取反的是因为我们想让向量指向光源而不是从光源出发）之间的点积。记住要对所有的相关向量标准化。

你可能奇怪为什么在 if 条件中使用的是 `>` 符号而不是 `<` 符号。`theta` 不应该比光的切光角更小才是在聚光内部吗？这并没有错，但不要忘记角度值现在都由余弦值来表示的。一个 0 度的角度表示的是 1.0 的余弦值，而一个 90 度的角度表示的是 0.0 的余弦值，你可以在下图中看到：



你现在可以看到，余弦值越接近 1.0，它的角度就越小。这也解释了为什么 `theta` 要比切光值更大了。切光值目前设置为 12.5 的余弦，约等于 0.9978，所以在 0.9979 到 1.0 内的 `theta` 值才能保证片段在聚光内，从而被照亮。

运行程序，你将会看到一个聚光，它仅会照亮聚光圆锥内的片段。看起来像是这样的：



你可以在[这里](#)获得全部源码。

但这仍看起来有些假，主要是因为聚光有一圈硬边。当一个片段遇到聚光圆锥的边缘时，它会完全变暗，没有一点平滑的过渡。一个真实的聚光将会在边缘处逐渐减少亮度。

平滑/软化边缘

为了创建一种看起来边缘平滑的聚光，我们需要模拟聚光有一个内圆锥(Inner Cone)和一个外圆锥(Outer Cone)。我们可以将内圆锥设置为上一部分中的那个圆锥，但我们也需要一个外圆锥，来让光从内圆锥逐渐减暗，直到外圆锥的边界。

为了创建一个外圆锥，我们只需要再定义一个余弦值来代表聚光方向向量和外圆锥向量（等于它的半径）的夹角。然后，如果一个片段处于内外圆锥之间，将会给它计算出一个0.0到1.0之间的强度值。如果片段在内圆锥之内，它的强度就是1.0，如果在外圆锥之外强度值就是0.0。

我们可以用下面这个公式来计算这个值：

这里(Epsilon)是内 (θ_i) 和外圆锥 (θ_o) 之间的余弦值差 ($\Delta\theta$)。最终的值就是在当前片段聚光的强度。

很难去表现这个公式是怎么工作的，所以我们用一些实例值来看看：

(角度)	(内光切)	(角度)	(外光切)	(角度)		
0.87 30	0.91	25	0.82	35	$0.91 - 0.82 = 0.09$	$0.87 - 0.82 / 0.09 = 0.56$
0.9 26	0.91	25	0.82	35	$0.91 - 0.82 = 0.09$	$0.9 - 0.82 / 0.09 = 0.89$
0.97 14	0.91	25	0.82	35	$0.91 - 0.82 = 0.09$	$0.97 - 0.82 / 0.09 = 1.67$
0.83 34	0.91	25	0.82	35	$0.91 - 0.82 = 0.09$	$0.83 - 0.82 / 0.09 = 0.11$
0.64 50	0.91	25	0.82	35	$0.91 - 0.82 = 0.09$	$0.64 - 0.82 / 0.09 = -2.0$
0.966 15	0.9978	12.5	0.953	17.5	$0.966 - 0.953 = 0.0448$	$0.966 - 0.953 / 0.0448 = 0.29$

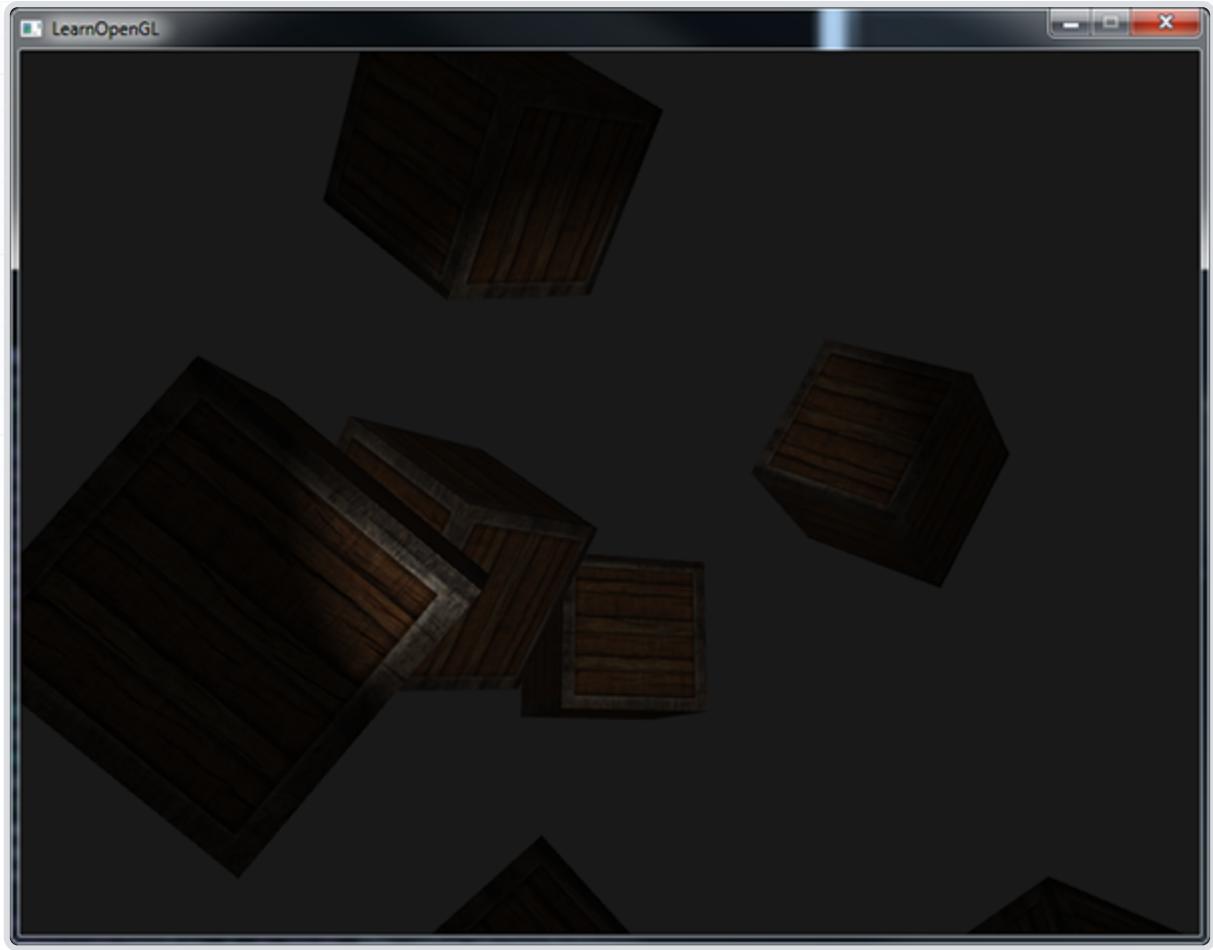
你可以看到，我们基本是在内外余弦值之间根据插值。如果你仍不明白发生了什么，不必担心，只需要记住这个公式就好了，在你更聪明的时候再回来看看。

我们现在有了一个在聚光外是负的，在内圆锥内大于1.0的，在边缘处于两者之间的强度值了。如果我们正确地约束(Clamp)这个值，在片段着色器中就不再需要 `if-else` 了，我们能够使用计算出来的强度值直接乘以光照分量：

```
float theta      = dot(lightDir, normalize(-light.direction));
float epsilon    = light.cutOff - light.outerCutOff;
float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
...
// 将不对环境光做出影响，让它总是能有一点光
diffuse *= intensity;
specular *= intensity;
...
```

注意我们使用了 `clamp` 函数，它把第一个参数约束(Clamp)在了0.0到1.0之间。这保证强度值不会在[0, 1]区间之外。

确定你将 `outerCutOff` 值添加到了 `Light` 结构体之中，并在程序中设置它的uniform值。下面的图片中，我们使用的内切光角是12.5，外切光角是17.5：



啊，这样看起来就好多了。稍微对内外切光角实验一下，尝试创建一个更能符合你需求的聚光。你可以在[这里](#)找到程序的源码。

这样的手电筒/聚光类型的灯光非常适合恐怖游戏，结合定向光和点光源，环境就会开始被照亮了。在[下一节](#)的教程中，我们将会结合我们至今讨论的所有光照和技巧。

练习

- 尝试实验一下上面的所有光照类型和它们的片段着色器。试着对一些向量进行取反，并使用`<`来代替`>`。试着解释不同视觉效果产生的原因。

2.3.16 多光源

原文 [Multiple lights](#)

作者 JoeyDeVries

翻译 [Geequlim](#), Krasjet

校对 暂未校对

我们在前面的教程中已经学习了许多关于OpenGL中光照的知识，其中包括冯氏着色(Phong Shading)、材质(Material)、光照贴图(Lighting Map)以及不同种类的投光物(Light Caster)。在这一节中，我们将结合之前学过的所有知识，创建一个包含六个光源的场景。我们将模拟一个类似太阳的定向光(Directional Light)光源，四个分散在场景中的点光源(Point Light)，以及一个手电筒(Flashlight)。

为了在场景中使用多个光源，我们希望将光照计算封装到GLSL函数中。这样做的原因是，每一种光源都需要一种不同的计算方法，而一旦我们想对多个光源进行光照计算时，代码很快就会变得非常复杂。如果我们只在main函数中进行所有的这些计算，代码很快就会变得难以理解。

GLSL中的函数和C函数很相似，它有一个函数名、一个返回值类型，如果函数不是在main函数之前声明的，我们还必须在代码文件顶部声明一个原型。我们对每个光照类型都创建一个不同的函数：定向光、点光源和聚光。

当我们在场景中使用多个光源时，通常使用以下方法：我们需要有一个单独的颜色向量代表片段的输出颜色。对于每一个光源，它对片段的贡献颜色将会加到片段的输出颜色向量上。所以场景中的每个光源都会计算它们各自对片段的影响，并结合为一个最终的输出颜色。大体的结构会像是这样：

```
out vec4 FragColor;

void main()
{
    // 定义一个输出颜色值
    vec3 output;
    // 将定向光的贡献加到输出中
    output += someFunctionToCalculateDirectionalLight();
    // 对所有的点光源也做相同的事情
    for(int i = 0; i < nr_of_point_lights; i++)
        output += someFunctionToCalculatePointLight();
    // 也加上其它的光源（比如聚光）
    output += someFunctionToCalculateSpotLight();

    FragColor = vec4(output, 1.0);
}
```

实际的代码对每一种实现都可能不同，但大体的结构都是差不多的。我们定义了几个函数，用来计算每个光源的影响，并将最终的结果颜色加到输出颜色向量上。例如，如果两个光源都很靠近一个片段，那么它们所结合的贡献将会形成一个比单个光源照亮时更加明亮的片段。

定向光

我们需要在片段着色器中定义一个函数来计算定向光对相应片段的贡献：它接受一些参数并计算一个定向光照射颜色。

首先，我们需要定义一个定向光源最少所需要的变量。我们可以将这些变量储存在一个叫做`DirLight`的结构体中，并将它定义为一个uniform。需要的变量在[上一节](#)中都介绍过：

```
struct DirLight {
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform DirLight dirLight;
```

接下来我们可以将`dirLight`传入一个有着一下原型的函数。

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
```

和C/C++一样，如果我们想调用一个函数（这里是在`main`函数中调用），这个函数需要在调用者的行数之前被定义过。在这个例子中我们更喜欢在`main`函数以下定义函数，所以上面要求就不满足了。所以，我们需要在`main`函数之上定义函数的原型，这和C语言中是一样的。

你可以看到，这个函数需要一个`DirLight`结构体和其它两个向量来进行计算。如果你认真完成了上一节的话，这个函数的内容应该理解起来很容易：

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);
    // 漫反射着色
    float diff = max(dot(normal, lightDir), 0.0);
    // 镜面光着色
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // 合并结果
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    return (ambient + diffuse + specular);
}
```

我们基本上只是从上一节中复制了代码，并使用函数参数的两个向量来计算定向光的贡献向量。最终环境光、漫反射和镜面光的贡献将会合并为单个颜色向量返回。

点光源

和定向光一样，我们也希望定义一个用于计算点光源对相应片段贡献，以及衰减，的函数。同样，我们定义一个包含了点光源所需所有变量的结构体：

```
struct PointLight {
    vec3 position;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

#define NR_POINT_LIGHTS 4
uniform PointLight pointLights[NR_POINT_LIGHTS];
```

你可以看到，我们在GLSL中使用了预处理指令来定义了我们场景中点光源的数量。接着我们使用了这个`NR_POINT_LIGHTS`常量来创建了一个`PointLight`结构体的数组。GLSL中的数组和C数组一样，可以使用一对方括号来创建。现在我们有四个待填充数据的`PointLight`结构体。

我们也可以定义一个大的结构体（而不是为每种类型的光源定义不同的结构体），包含所有不同种光照类型所需的变量，并将这个结构体用到所有的函数中，只需要忽略用不到的变量就行了。然而，我个人觉得当前的方法会更直观一点，不仅能够节省一些代码，而且由于不是所有光照类型都需要所有的变量，这样也能节省一些内存。

点光源函数的原型如下：

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
```

这个函数从参数中获取所需的所有数据，并返回一个代表该点光源对片段的颜色贡献的 `vec3`。我们再一次聪明地从之前的教程中复制粘贴代码，完成了下面这样的函数：

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // 漫反射着色
    float diff = max(dot(normal, lightDir), 0.0);
    // 镜面光着色
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // 衰减
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance +
        light.quadratic * (distance * distance));
    // 合并结果
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

将这些功能抽象到这样一个函数中的优点是，我们能够不用重复的代码而很容易地计算多个点光源的光照了。在 `main` 函数中，我们只需要创建一个循环，遍历整个点光源数组，对每个点光源调用 `CalcPointLight` 就可以了。

合并结果

现在我们已经定义了一个计算定向光的函数和一个计算点光源的函数了，我们可以将它们合并放到 `main` 函数中。

```
void main()
{
    // 属性
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);

    // 第一阶段：定向光照
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    // 第二阶段：点光源
    for(int i = 0; i < NR_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
    // 第三阶段：聚光
    //result += CalcSpotLight(spotLight, norm, FragPos, viewDir);

    FragColor = vec4(result, 1.0);
}
```

每个光源类型都将它们的贡献加到了最终的输出颜色上，直到所有的光源都处理完了。最终的颜色包含了场景中所有光源的颜色影响所合并的结果。如果你想的话，你也可以实现一个聚光，并将它的效果加到输出颜色中。我们会将 `CalcSpotLight` 函数留给读者作为练习。

设置定向光结构体的 `uniform` 应该非常熟悉了，但是你可能会在想我们该如何设置点光源的 `uniform` 值，因为点光源的 `uniform` 现在是一个 `PointLight` 的数组了。这并不是我们以前讨论过的话题。

很幸运的是，这并不是很复杂，设置一个结构体数组的uniform和设置一个结构体的uniform是很相似的，但是这一次在访问uniform位置的时候，我们需要定义对应的数组下标值：

```
lightingShader.setFloat("pointLights[0].constant", 1.0f);
```

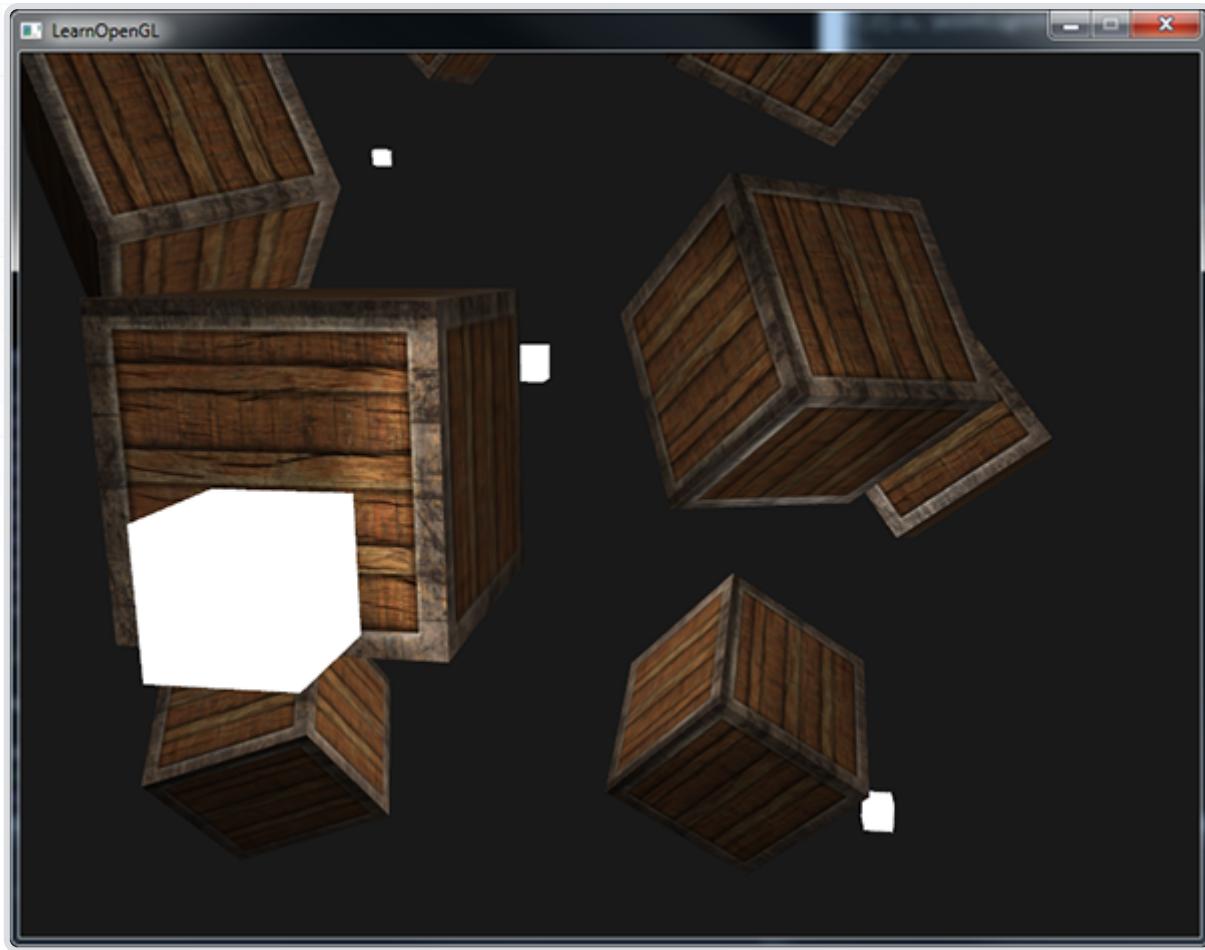
在这里我们索引了`pointLights`数组中的第一个`PointLight`，并获取了`constant`变量的位置。但这也意味着不幸的是我们必须对这四个点光源手动设置uniform值，这让点光源本身就产生了28个uniform调用，非常冗长。你也可以尝试将这些抽象出去一点，定义一个点光源类，让它来为你设置uniform值，但最后你仍然要用这种方式设置所有光源的uniform值。

别忘了，我们还需要为每个点光源定义一个位置向量，所以我们让它们在场景中分散一点。我们会定义另一个`glm::vec3`数组来包含点光源的位置：

```
glm::vec3 pointLightPositions[] = {
    glm::vec3( 0.7f,  0.2f,  2.0f),
    glm::vec3( 2.3f, -3.3f, -4.0f),
    glm::vec3(-4.0f,  2.0f, -12.0f),
    glm::vec3( 0.0f,  0.0f, -3.0f)
};
```

接下来我们从`pointLights`数组中索引对应的`PointLight`，将它的`position`值设置为刚刚定义的位置值数组中的其中一个。同时我们还要保证现在绘制的是四个灯立方体而不是仅仅一个。只要对每个灯物体创建一个不同的模型矩阵就可以了，和我们之前对箱子的处理类似。

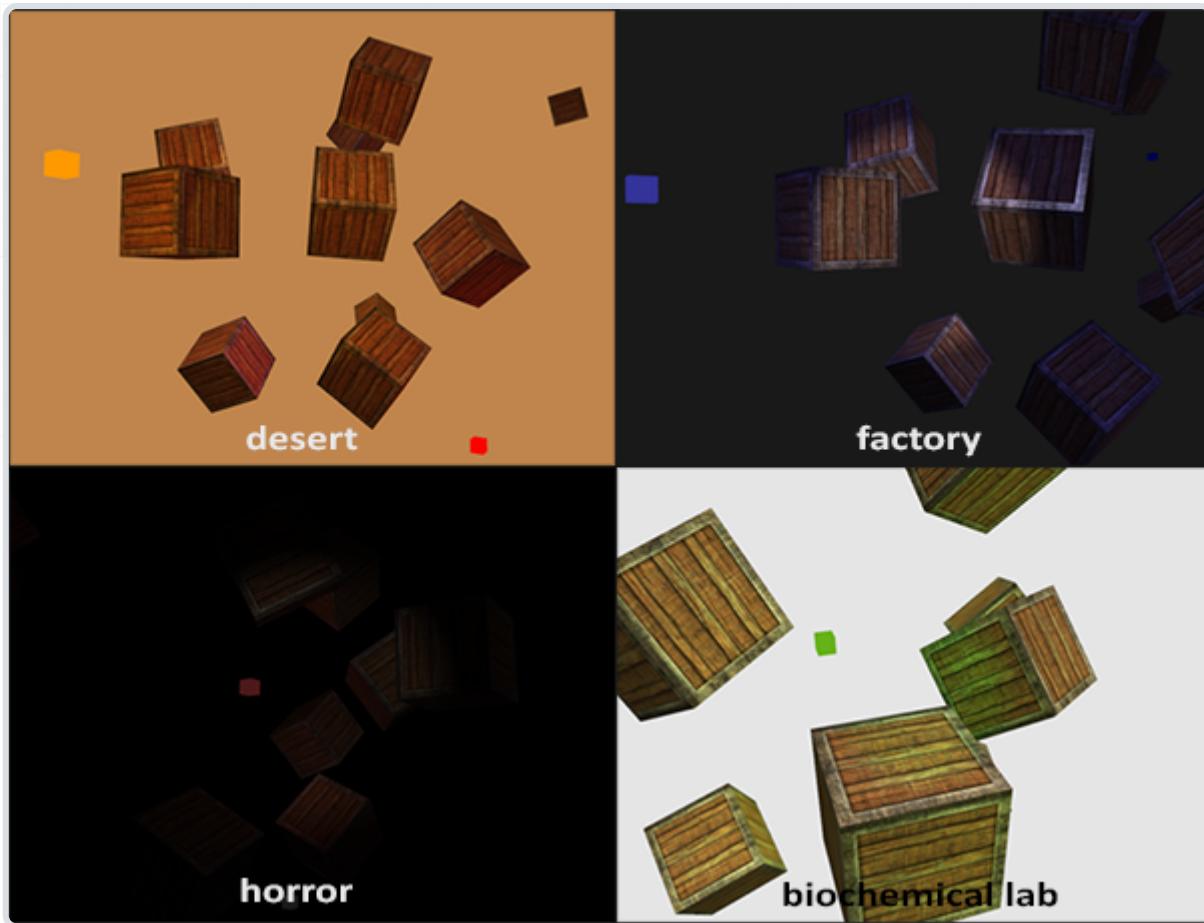
如果你还使用了手电筒的话，所有光源组合的效果将看起来和下图差不多：



你可以看到，很显然天空中有一个全局照明（像一个太阳），我们有四个光源分散在场景中，以及玩家视角的手电筒。看起来是不是非常不错？

你可以在[这里](#)找到最终程序的源代码。

上面图片中的所有光源都是使用上一节中所使用的默认属性，但如果你愿意实验这些数值的话，你能够得到很多有意思的结果。艺术家和关卡设计师通常都在编辑器中不断的调整这些光照参数，保证光照与环境相匹配。在我们刚刚创建的简单光照环境中，你可以简单的调整一下光源的属性，创建很多有意思的视觉效果：



我们也改变了清屏的颜色来更好地反应光照。你可以看到，只需要简单地调整一些光照参数，你就能创建完全不同的氛围。

相信你现在已经对OpenGL的光照有很好的理解了。有了目前所学的这些知识，我们已经可以创建出丰富有趣的环境和氛围了。尝试实验一下不同的值，创建出你自己的氛围吧。

练习

- 你能通过调节光照属性变量，（大概地）重现最后一张图片上不同的氛围吗？[参考解答](#)

2.3.17 复习

原文 [Review](#)

作者 JoeyDeVries

翻译 Krasjet

校对 [Geequlim](#), Krasjet

恭喜您已经学习到了这个地方！辛苦啦！不知道你有没有注意到，总的来说我们在学习光照教程的时候关于OpenGL本身并没有什么新东西，除了像访问uniform数组这样细枝末节的知识。目前为止的所有教程都是关于使用一些技巧或者公式来操作着色器，达到真实的光照效果。这再一次向你展示了着色器的威力。着色器是非常灵活的，你也亲眼见证了我们仅仅使用一些3D向量和可配置的变量就能够创造出惊人的图像这一点。

在前面的几个教程中，你学习了颜色、冯氏光照模型（包括环境光照、漫反射光照和镜面光照）、物体的材质、可配置的光照属性、漫反射和镜面光贴图、不同种类的光，并且学习了怎样将所有所学知识融会贯通，合并到一个程序当中。记得去实验一下不同的光照、材质颜色、光照属性，并且试着利用你无穷的创造力创建自己的环境。

在[下一节](#)当中，我们在我们的场景当中加入更高级的形状，这些形状将会在我们之前讨论过的光照模型中非常好看。

词汇表

- 颜色向量(Color Vector): 一个通过红绿蓝(RGB)分量的组合描绘大部分真实颜色的向量。一个物体的颜色实际上是该物体所不能吸收的反射颜色分量。
- 冯氏光照模型(Phong Lighting Model): 一个通过计算环境光，漫反射，和镜面光分量的值来估计真实光照的模型。
- 环境光照(Ambient Lighting): 通过给每个没有被光照的物体很小的亮度，使其不是完全黑暗的，从而对全局光照进行估计。
- 漫反射着色(Diffuse Shading): 一个顶点/片段与光线方向越接近，光照会越强。使用了法向量来计算角度。
- 法向量(Normal Vector): 一个垂直于平面的单位向量。
- 法线矩阵(Normal Matrix): 一个3x3矩阵，或者说是没有平移的模型（或者模型-观察）矩阵。它也被以某种方式修改（逆转置），从而在应用非统一缩放时，保持法向量朝向正确的方向。否则法向量会在使用非统一缩放时被扭曲。
- 镜面光照(Specular Lighting): 当观察者视线靠近光源在表面的反射线时会显示的镜面高光。镜面光照是由观察者的方向，光源的方向和设定高光分散量的反光度值三个量共同决定的。
- 冯氏着色(Phong Shading): 冯氏光照模型应用在片段着色器。
- Gouraud着色(Gouraud shading): 冯氏光照模型应用在顶点着色器上。在使用很少数量的顶点时会产生明显的瑕疵。会得到效率提升但是损失了视觉质量。
- GLSL结构体(GLSL struct): 一个类似于C的结构体，用作着色器变量的容器。大部分时间用来管理输入/输出/uniform。
- 材质(Material): 一个物体反射的环境光，漫反射，镜面光颜色。这些东西设定了物体所拥有的颜色。
- 光照属性(Light(properties)): 一个光的环境光，漫反射，镜面光的强度。可以使用任何颜色值，对每一个冯氏分量(Phong Component)定义光源发出的颜色/强度。
- 漫反射贴图(Diffuse Map): 一个设定了每个片段中漫反射颜色的纹理图片。
- 镜面光贴图(Specular Map): 一个设定了每一个片段的镜面光强度/颜色的纹理贴图。仅在物体的特定区域显示镜面高光。
- 定向光(Directional Light): 只有一个方向的光源。它被建模为不管距离有多长所有光束都是平行而且其方向向量在整个场景中保持不变。
- 点光源(Point Light): 一个在场景中有位置的，光线逐渐衰减的光源。
- 衰减(Attenuation): 光随着距离减少强度的过程，通常使用在点光源和聚光下。
- 聚光(Spotlight): 一个被定义为在某一个方向上的锥形的光源。
- 手电筒(Flashlight): 一个摆放在观察者视角的聚光。
- GLSL uniform数组(GLSL Uniform Array): 一个uniform值数组。它的工作原理和C语言数组大致一样，只是不能动态分配内存。

2.4 模型加载

2.4.1 Assimp

原文 [Assimp](#)

作者 JoeyDeVries

翻译 Cocoonshu, Krasjet, [Geequlim](#)

校对 暂未校对

到目前为止的所有场景中，我们一直都在滥用我们的箱子朋友，但时间久了甚至是我们最好的朋友也会感到无聊。在日常的图形程序中，通常都会使用非常复杂且好玩的模型，它们比静态的箱子要好看多了。然而，和箱子对象不同，我们不太能够对像是房子、汽车或者人形角色这样的复杂形状手工定义所有的顶点、法线和纹理坐标。我们想要的是将这些模型(Model)导入(Import)到程序当中。模型通常都由3D艺术家在[Blender](#)、[3DS Max](#)或者[Maya](#)这样的工具中精心制作。

这些所谓的3D建模工具(3D Modeling Tool)可以让艺术家创建复杂的形状，并使用一种叫做UV映射(uv-mapping)的手段来应用贴图。这些工具将会在导出到模型文件的时候自动生成所有的顶点坐标、顶点法线以及纹理坐标。这样子艺术家们即使不了解图形技术细节的情况下，也能拥有一套强大的工具来构建高品质的模型了。所有的技术细节都隐藏在了导出的模型文件中。但是，作为图形开发者，我们就必须要了解这些技术细节了。

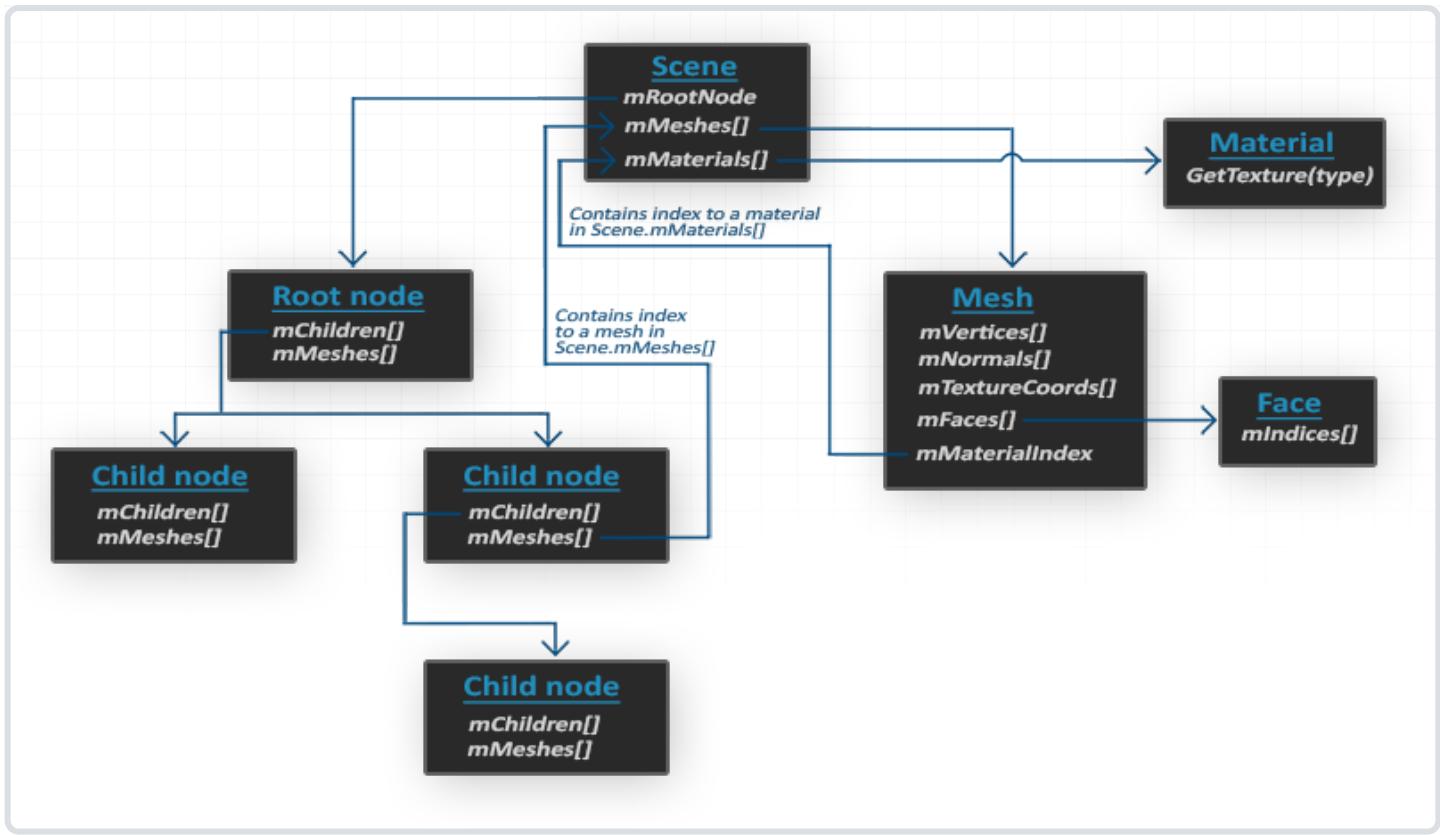
所以，我们的工作就是解析这些导出的模型文件以及提取所有有用的信息，将它们储存为OpenGL能够理解的格式。一个很常见的问题是，模型的文件格式有很多种，每一种都会以它们自己的方式来导出模型数据。像是[Wavefront的.obj](#)这样的模型格式，只包含了模型数据以及材质信息，像是模型颜色和漫反射/镜面光贴图。而以XML为基础的[Collada文件格式](#)则非常的丰富，包含模型、光照、多种材质、动画数据、摄像机、完整的场景信息等等。Wavefront的.obj格式通常被认为是一个易于解析的模型格式。建议至少去Wavefront的wiki页面上看看文件格式的信息是如何封装的。这应该能让你认识到模型文件的基本结构。

总而言之，不同种类的文件格式有很多，它们之间通常并没有一个通用的结构。所以如果我们想从这些文件格式中导入模型的话，我们必须要去自己对每一种需要导入的文件格式写一个导入器。很幸运的是，正好有一个库专门处理这个问题。

模型加载库

一个非常流行的模型导入库是[Assimp](#)，它是Open Asset Import Library（开放的资产导入库）的缩写。Assimp能够导入很多种不同的模型文件格式（并也能够导出部分的格式），它会将所有的模型数据加载至Assimp的通用数据结构中。当Assimp加载完模型之后，我们就能够从Assimp的数据结构中提取我们所需的所有数据了。由于Assimp的数据结构保持不变，不论导入的是什么种类的文件格式，它都能够将我们从这些不同的文件格式中抽象出来，用同一种方式访问我们需要的数据。

当使用Assimp导入一个模型的时候，它通常会将整个模型加载进一个场景(Scene)对象，它会包含导入的模型/场景中的所有数据。Assimp会将场景载入为一系列的节点(Node)，每个节点包含了场景对象中所储存数据的索引，每个节点都可以有任意数量的子节点。Assimp数据结构的（简化）模型如下：



- 和材质和网格(Mesh)一样，所有的场景/模型数据都包含在Scene对象中。Scene对象也包含了场景根节点的引用。
- 场景的Root node（根节点）可能包含子节点（和其他的节点一样），它会有一系列指向场景对象中mMeshes数组中储存的网格数据的索引。Scene下的mMeshes数组储存了真正的Mesh对象，节点中的mMeshes数组保存的只是场景中网格数组的索引。
- 一个Mesh对象本身包含了渲染所需要的所有相关数据，像是顶点位置、法向量、纹理坐标、面(Face)和物体的材质。
- 一个网格包含了多个面。Face代表的是物体的渲染图元(Primitive)（三角形、方形、点）。一个面包含了组成图元的顶点的索引。由于顶点和索引是分开的，使用一个索引缓冲来渲染是非常简单的（见[你好，三角形](#)）。
- 最后，一个网格也包含了一个Material对象，它包含了一些函数能让我们获取物体的材质属性，比如说颜色和纹理贴图（比如漫反射和镜面光贴图）。

所以，我们需要做的第一件事是将一个物体加载到Scene对象中，遍历节点，获取对应的Mesh对象（我们需要递归搜索每个节点的子节点），并处理每个Mesh对象来获取顶点数据、索引以及它的材质属性。最终的结果是一系列的网格数据，我们会将它们包含在一个Model对象中。

网格

当使用建模工具对物体建模的时候，艺术家通常不会用单个形状创建出整个模型。通常每个模型都由几个子模型/形状组合而成。组合模型的每个单独的形状就叫做一个网格(Mesh)。比如说有一个人形的角色：艺术家通常会将头部、四肢、衣服、武器建模为分开的组件，并将这些网格组合而成的结果表现为最终的模型。一个网格是我们在OpenGL中绘制物体所需的最小单位（顶点数据、索引和材质属性）。一个模型（通常）会包括多个网格。

在[下一节](#)中，我们将创建我们自己的Model和Mesh类来加载并使用刚刚介绍的结构储存导入后的模型。如果我们想要绘制一个模型，我们不需要将整个模型渲染为一个整体，只需要渲染组成模型的每个独立的网格就可以了。然而，在我们开始导入模型之前，我们首先需要将Assimp包含到我们的工程当中。

构建Assimp

你可以在Assimp的[下载页面](#)中选择相应的版本。在写作时使用的Assimp最高版本为3.1.1。我们建议你自己编译Assimp库，因为它们的预编译库在大部分系统上都是不能运行的。如果你忘记如何使用CMake自己编译一个库的话，可以复习[创建窗口](#)小节。

构建Assimp时可能会出现一些问题，所以我会将它们的解决方案列在这里，便于大家排除错误：

- CMake在读取配置列表时，不断报出DirectX库丢失的错误。报错如下：

```
Could not locate DirectX
CMake Error at cmake-modules/FindPkgMacros.cmake:110 (message):
Required library DirectX not found! Install the library (including dev packages)
and try again. If the library is already installed, set the missing variables
manually in cmake.
```

这个问题的解决方案是安装DirectX SDK，如果你之前没安装过的话。你可以从[这里](#)下载SDK。

- 安装DirectX SDK时，可能遇到一个错误码为 `s1023` 的错误。这种情况下，请在安装SDK之前根据[这个](#)先卸载C++ Redistributable package(s)。
- 一旦配置完成，你就可以生成解决方案文件了，打开解决方案文件并编译Assimp库（可以编译为Debug版本也可以编译为 Release版本，只要能工作就行）。
- 使用默认配置构建的Assimp是一个动态库(Dynamic Library)，所以我们需要包含所生成的`assimp.dll`文件以及程序的二进制文件。你可以简单地将DLL复制到我们程序可执行文件的同一目录中。
- Assimp编译之后，生成的库和DLL文件位于`code/Debug`或者`code/Release`文件夹中。
- 接着把编译好的LIB文件和DLL文件拷贝到工程的相应目录下，并在解决方案中链接它们。并且记得把Assimp的头文件也复制到你的`include`目录中（头文件可以在从Assimp中下载的`include`目录里找到）。

如果你仍遇到了未报告的错误，欢迎在评论区中寻求帮助。

如果你想让Assimp使用多线程来获得更高的性能，你可以使用Boost库来编译Assimp。你可以在它们的[安装页面](#)找到完整的安装介绍。

现在，你应该已经编译完Assimp库并将它链接到你的程序中了。下一步：[导入漂亮的3D物体！](#)

2.4.2 网格

原文 [Mesh](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

通过使用Assimp，我们可以加载不同的模型到程序中，但是载入后它们都被储存为Assimp的数据结构。我们最终仍要将这些数据转换为OpenGL能够理解的格式，这样才能渲染这个物体。我们从上一节中学到，网格(Mesh)代表的是单个的可绘制实体，我们现在先来定义一个我们自己的网格类。

首先我们来回顾一下我们目前学到的知识，想想一个网格最少需要什么数据。一个网格应该至少需要一系列的顶点，每个顶点包含一个位置向量、一个法向量和一个纹理坐标向量。一个网格还应该包含用于索引绘制的索引以及纹理形式的材质数据（漫反射/镜面光贴图）。

既然我们有了一个网格类的最低需求，我们可以在OpenGL中定义一个顶点了：

```
struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
};
```

我们将所有需要的向量储存到一个叫做`Vertex`的结构体中，我们可以用它来索引每个顶点属性。除了`Vertex`结构体之外，我们还需要将纹理数据整理到一个`Texture`结构体中。

```
struct Texture {
    unsigned int id;
    string type;
};
```

我们储存了纹理的id以及它的类型，比如是漫反射贴图或者是镜面光贴图。

知道了顶点和纹理的实现，我们可以开始定义网格类的结构了：

```
class Mesh {
public:
    /* 网格数据 */
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;
    /* 函数 */
    Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures);
    void Draw(Shader shader);
private:
    /* 渲染数据 */
    unsigned int VAO, VBO, EBO;
    /* 函数 */
    void setupMesh();
};
```

你可以看到这个类并不复杂。在构造器中，我们将所有必须的数据赋予了网格，我们在`setupMesh`函数中初始化缓冲，并最终使用`Draw`函数来绘制网格。注意我们将一个着色器传入了`Draw`函数中，将着色器传入网格类中可以让我们在绘制之前设置一些uniform（像是链接采样器到纹理单元）。

构造器的内容非常易于理解。我们只需要使用构造器的参数设置类的公有变量就可以了。我们在构造器中还调用了`setupMesh`函数：

```
Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures)
{
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;

    setupMesh();
}
```

这里没什么可说的。我们接下来讨论`setupMesh`函数。

初始化

由于有了构造器，我们现在有一大列的网格数据用于渲染。在此之前我们还必须配置正确的缓冲，并通过顶点属性指针定义顶点着色器的布局。现在你应该对这些概念都很熟悉了，但我们这次会稍微有一点变动，使用结构体中的顶点数据：

```
void setupMesh()
{
    glGenVertexArrays(1, &VA0);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
                 &indices[0], GL_STATIC_DRAW);

    // 顶点位置
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    // 顶点法线
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)(offsetof(Vertex, Normal)));
    // 顶点纹理坐标
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)(offsetof(Vertex, TexCoords)));

    glBindVertexArray(0);
}
```

代码应该和你所想得没什么不同，但有了`Vertex`结构体的帮助，我们使用了一些小技巧。

C++结构体有一个很棒的特性，它们的内存布局是连续的(Sequential)。也就是说，如果我们将结构体作为一个数据数组使用，那么它将会以顺序排列结构体的变量，这将会直接转换为我们在数组缓冲中所需要的float（实际上是字节）数组。比如说，如果我们有一个填充后的Vertex结构体，那么它的内存布局将会等于：

```
Vertex vertex;
vertex.Position = glm::vec3(0.2f, 0.4f, 0.6f);
vertex.Normal = glm::vec3(0.0f, 1.0f, 0.0f);
vertex.TexCoords = glm::vec2(1.0f, 0.0f);
// = [0.2f, 0.4f, 0.6f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f];
```

由于有了这个有用的特性，我们能够直接传入一大列的Vertex结构体的指针作为缓冲的数据，它们将会完美地转换为glBufferData所能用的参数：

```
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);
```

自然 sizeof 运算也可以用在结构体上来计算它的字节大小。这个应该是32字节的（8个float * 每个4字节）。

结构体的另外一个很好的用途是它的预处理指令 `offsetof(s, m)`，它的第一个参数是一个结构体，第二个参数是这个结构体中变量的名字。这个宏会返回那个变量距结构体头部的字节偏移量(Byte Offset)。这正好可以用在定义glVertexAttribPointer函数中的偏移参数：

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
```

偏移量现在是使用offsetof来定义了，在这里它会将法向量的字节偏移量设置为结构体中法向量的偏移量，也就是3个float，即12字节。注意，我们同样将步长参数设置为了Vertex结构体的大小。

使用这样的一个结构体不仅能够提供可读性更高的代码，也允许我们很容易地拓展这个结构。如果我们希望添加另一个顶点属性，我们只需要将它添加到结构体中就可以了。由于它的灵活性，渲染的代码不会被破坏。

渲染

我们需要为Mesh类定义最后一个函数，它的Draw函数。在真正渲染这个网格之前，我们需要在调用glDrawElements函数之前先绑定相应的纹理。然而，这实际上有些困难，我们一开始并不知道这个网格（如果有的话）有多少纹理、纹理是什么类型的。所以我们该如何在着色器中设置纹理单元和采样器呢？

为了解决这个问题，我们需要设定一个命名标准：每个漫反射纹理被命名为 `texture_diffuseN`，每个镜面光纹理应该被命名为 `texture_specularN`，其中 `N` 的范围是1到纹理采样器最大允许的数字。比如说我们对某一个网格有3个漫反射纹理，2个镜面光纹理，它们的纹理采样器应该之后会被调用：

```
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_diffuse2;
uniform sampler2D texture_diffuse3;
uniform sampler2D texture_specular1;
uniform sampler2D texture_specular2;
```

根据这个标准，我们可以在着色器中定义任意需要数量的纹理采样器，如果一个网格真的包含了（这么多）纹理，我们也能知道它们的名字是什么。根据这个标准，我们也能在一个网格中处理任意数量的纹理，开发者也可以自由选择需要使用的数量，他只需要定义正确的采样器就可以了（虽然定义少的话会有点浪费绑定和uniform调用）。

像这样的问题有很多种不同的解决方案。如果你不喜欢这个解决方案，你可以自己想一个你自己的解决办法。

最终的渲染代码是这样的：

```
void Draw(Shader shader)
{
    unsigned int diffuseNr = 1;
    unsigned int specularNr = 1;
    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
        glActiveTexture(GL_TEXTURE0 + i); // 在绑定之前激活相应的纹理单元
        // 获取纹理序号 (diffuse_textureN 中的 N)
        string number;
        string name = textures[i].type;
        if(name == "texture_diffuse")
            number = std::to_string(diffuseNr++);
        else if(name == "texture_specular")
            number = std::to_string(specularNr++);
        shader.setFloat(("material." + name + number).c_str(), i);
    }
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}

// 绘制网格
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
}
```

我们首先计算了每个纹理类型的N-分量，并将其拼接到纹理类型字符串上，来获取对应的uniform名称。接下来我们查找对应的采样器，将它的位置值设置为当前激活的纹理单元，并绑定纹理。这也是我们在Draw函数中需要着色器的原因。我们也将"material."添加到了最终的uniform名称中，因为我们希望将纹理储存在一个材质结构体中（这在每个实现中可能都不同）。

注意我们在将漫反射计数器和镜面光计数器插入 `stringstream` 时，对它们进行了递增。在C++中，这个递增操作：

`variable++` 将会返回变量本身，之后再递增，而 `++variable` 则是先递增，再返回值。在我们的例子中是首先将原本的计数器值插入 `stringstream`，之后再递增它，供下一次循环使用。

你可以在[这里](#)找到Mesh类的完整源代码

我们刚定义的Mesh类是我们之前讨论的很多话题的抽象结果。在[下一节](#)中，我们将创建一个模型，作为多个网格对象的容器，并真正地实现Assimp的加载接口。

2.4.3 模型

原文 [Model](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

现在是时候接触Assimp并创建实际的加载和转换代码了。这个教程的目标是创建另一个类来完整地表示一个模型，或者说是包含多个网格，甚至是多个物体的模型。一个包含木制阳台、塔楼、甚至游泳池的房子可能仍会被加载为一个模型。我们会使用Assimp来加载模型，并将它转换(Translate)至多个在[上一节](#)中创建的Mesh对象。

事不宜迟，我会先把Model类的结构给你：

```
class Model
{
public:
    /* 函数 */
    Model(char *path)
    {
        loadModel(path);
    }
    void Draw(Shader shader);
private:
    /* 模型数据 */
    vector<Mesh> meshes;
    string directory;
    /* 函数 */
    void loadModel(string path);
    void processNode(aiNode *node, const aiScene *scene);
    Mesh processMesh(aiMesh *mesh, const aiScene *scene);
    vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type,
                                         string typeName);
};
```

Model类包含了一个Mesh对象的vector（译注：这里指的是C++中的vector模板类，之后遇到均不译），构造器需要我们给它一个文件路径。在构造器中，它会直接通过loadModel来加载文件。私有函数将会处理Assimp导入过程中的一部分，我们很快就会介绍它们。我们还将储存文件路径的目录，在之后加载纹理的时候还会用到它。

Draw函数没有什么特别之处，基本上就是遍历了所有网格，并调用它们各自的Draw函数。

```
void Draw(Shader shader)
{
    for(unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
}
```

导入3D模型到OpenGL

要想导入一个模型，并将它转换到我们自己的数据结构中的话，首先我们需要包含Assimp对应的头文件，这样编译器就不会抱怨我们了。

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

首先需要调用的函数是`loadModel`，它会从构造器中直接调用。在`loadModel`中，我们使用Assimp来加载模型至Assimp的一个叫做`scene`的数据结构中。你可能还记得在模型加载章节的[第一节](#)教程中，这是Assimp数据接口的根对象。一旦我们有了这个场景对象，我们就能访问到加载后的模型中所有所需的数据了。

Assimp很棒的一点在于，它抽象掉了加载不同文件格式的所有技术细节，只需要一行代码就能完成所有的工作：

```
Assimp::Importer importer;
const aiScene *scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);
```

我们首先声明了Assimp命名空间内的一个`Importer`，之后调用了它的`ReadFile`函数。这个函数需要一个文件路径，它的第二个参数是一些[后期处理\(Post-processing\)](#)的选项。除了加载文件之外，Assimp允许我们设定一些选项来强制它对导入的数据做一些额外的计算或操作。通过设定`aiProcess_Triangulate`，我们告诉Assimp，如果模型不是（全部）由三角形组成，它需要将模型所有的图元形状变换为三角形。`aiProcess_FlipUVs`将在处理的时候翻转y轴的纹理坐标（你可能还记得我们在[纹理](#)教程中说过，在OpenGL中大部分的图像的y轴都是反的，所以这个后期处理选项将会修复这个）。其它一些比较有用的选项有：

- `aiProcess_GenNormals`: 如果模型不包含法向量的话，就为每个顶点创建法线。
- `aiProcess_SplitLargeMeshes`: 将比较大的网格分割成更小的子网格，如果你的渲染有最大顶点数限制，只能渲染较小的网格，那么它会非常有用。
- `aiProcess_OptimizeMeshes`: 和上个选项相反，它会将多个小网格拼接为一个大的网格，减少绘制调用从而进行优化。

Assimp提供了很多有用的后期处理指令，你可以在[这里](#)找到全部的指令。实际上使用Assimp加载模型是非常容易的（你也可以看到）。困难的是之后使用返回的场景对象将加载的数据转换到一个`Mesh`对象的数组。

完整的`loadModel`函数将会是这样的：

```
void loadModel(string path)
{
    Assimp::Importer import;
    const aiScene *scene = import.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);

    if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        cout << "ERROR::ASSIMP::" << import.GetErrorString() << endl;
        return;
    }
    directory = path.substr(0, path.find_last_of('/'));

    processNode(scene->mRootNode, scene);
}
```

在我们加载了模型之后，我们会检查场景和其根节点不为null，并且检查了它的一个标记(Flag)，来查看返回的数据是不是不完整的。如果遇到了任何错误，我们都会通过导入器的`GetErrorString`函数来报告错误并返回。我们也获取了文件路径的目录路径。

如果什么错误都没有发生，我们希望处理场景中的所有节点，所以我们将第一个节点（根节点）传入了递归的`processNode`函数。因为每个节点（可能）包含有多个子节点，我们希望首先处理参数中的节点，再继续处理该节点所有的子节点，以此类推。这正符合一个递归结构，所以我们将定义一个递归函数。递归函数在做一些处理之后，使用不同的参数递归调用这个函数自身，直到某个条件被满足停止递归。在我们的例子中退出条件(Exit Condition)是所有的节点都被处理完毕。

你可能还记得Assimp的结构中，每个节点包含了一系列的网格索引，每个索引指向场景对象中的那个特定网格。我们接下来就想去获取这些网格索引，获取每个网格，处理每个网格，接着对每个节点的子节点重复这一过程。`processNode`函数的内容如下：

```
void processNode(aiNode *node, const aiScene *scene)
{
    // 处理节点所有的网格（如果有的话）
    for(unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh *mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }
    // 接下来对它的子节点重复这一过程
    for(unsigned int i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }
}
```

我们首先检查每个节点的网格索引，并索引场景的`mMeshes`数组来获取对应的网格。返回的网格将会传递到`processMesh`函数中，它会返回一个`Mesh`对象，我们可以将它存储在`meshes`列表/vector。

所有网格都被处理之后，我们会遍历节点的所有子节点，并对它们调用相同的`processMesh`函数。当一个节点不再有任何子节点之后，这个函数将会停止执行。

认真的读者可能会发现，我们可以基本上忘掉处理任何的节点，只需要遍历场景对象的所有网格，就不需要为了索引做这一堆复杂的东西了。我们仍这么做的原因是，使用节点的最初想法是将网格之间定义一个父子关系。通过这样递归地遍历这层关系，我们就能将某个网格定义为另一个网格的父网格了。

这个系统的一个使用案例是，当你想位移一个汽车的网格时，你可以保证它的所有子网格（比如引擎网格、方向盘网格、轮胎网格）都会随着一起位移。这样的系统能够用父子关系很容易地创建出来。

然而，现在我们并没有使用这样一种系统，但如果你想对你的网格数据有更多的控制，通常都是建议使用这种方法的。这种类节点的关系毕竟是由创建了这个模型的艺术家所定义。

下一步就是将Assimp的数据解析到上一节中创建的`Mesh`类中。

从Assimp到网格

将一个 `aiMesh` 对象转化为我们自己的网格对象不是那么困难。我们要做的只是访问网格的相关属性并将它们储存到我们自己的对象中。`processMesh` 函数的大体结构如下：

```
Mesh processMesh(aiMesh *mesh, const aiScene *scene)
{
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;

    for(unsigned int i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        // 处理顶点位置、法线和纹理坐标
        ...
        vertices.push_back(vertex);
    }
    // 处理索引
    ...
    // 处理材质
    if(mesh->mMaterialIndex >= 0)
    {
        ...
    }

    return Mesh(vertices, indices, textures);
}
```

处理网格的过程主要有三部分：获取所有的顶点数据，获取它们的网格索引，并获取相关的材质数据。处理后的数据将会储存在三个 `vector` 当中，我们会利用它们构建一个 `Mesh` 对象，并返回它到函数的调用者那里。

获取顶点数据非常简单，我们定义了一个 `Vertex` 结构体，我们将在每个迭代之后将它加到 `vertices` 数组中。我们会遍历网格中的所有顶点（使用 `mesh->mNumVertices` 来获取）。在每个迭代中，我们希望使用所有的相关数据填充这个结构体。顶点的位置是这样处理的：

```
glm::vec3 vector;
vector.x = mesh->mVertices[i].x;
vector.y = mesh->mVertices[i].y;
vector.z = mesh->mVertices[i].z;
vertex.Position = vector;
```

注意我们为了传输 Assimp 的数据，我们定义了一个 `vec3` 的临时变量。使用这样一个临时变量的原因是 Assimp 对向量、矩阵、字符串等都有自己的一套数据类型，它们并不能完美地转换到 GLM 的数据类型中。

Assimp 将它的顶点位置数组叫做 `mVertices`，这其实并不是那么直观。

处理法线的步骤也是差不多的：

```
vector.x = mesh->mNormals[i].x;
vector.y = mesh->mNormals[i].y;
vector.z = mesh->mNormals[i].z;
vertex.Normal = vector;
```

纹理坐标的处理也大体相似，但Assimp允许一个模型在一个顶点上有最多8个不同的纹理坐标，我们不会用到那么多，我们只关心第一组纹理坐标。我们同样也想检查网格是否真的包含了纹理坐标（可能并不会一直如此）

```
if(mesh->mTextureCoords[0]) // 网格是否有纹理坐标?
{
    glm::vec2 vec;
    vec.x = mesh->mTextureCoords[0][i].x;
    vec.y = mesh->mTextureCoords[0][i].y;
    vertex.TexCoords = vec;
}
else
    vertex.TexCoords = glm::vec2(0.0f, 0.0f);
```

`vertex`结构体现在已经填充好了需要的顶点属性，我们在迭代的最后将它压入`vertices`这个vector的尾部。这个过程会对每个网格的顶点都重复一遍。

索引

Assimp的接口定义了每个网格都有一个面(Face)数组，每个面代表了一个图元，在我们的例子中（由于使用了`aiProcess_Triangulate`选项）它总是三角形。一个面包含了多个索引，它们定义了在每个图元中，我们应该绘制哪个顶点，并以什么顺序绘制，所以如果我们遍历了所有的面，并储存了面的索引到`indices`这个vector中就可以了。

```
for(unsigned int i = 0; i < mesh->mNumFaces; i++)
{
    aiFace face = mesh->mFaces[i];
    for(unsigned int j = 0; j < face.mNumIndices; j++)
        indices.push_back(face.mIndices[j]);
}
```

所有的外部循环都结束了，我们现在有了一系列的顶点和索引数据，它们可以用来通过`glDrawElements`函数来绘制网格。然而，为了结束这个话题，并且对网格提供一些细节，我们还需要处理网格的材质。

材质

和节点一样，一个网格只包含了一个指向材质对象的索引。如果想要获取网格真正的材质，我们还需要索引场景的`mMaterials`数组。网格材质索引位于它的`mMaterialIndex`属性中，我们同样可以用它来检测一个网格是否包含有材质：

```
if(mesh->mMaterialIndex >= 0)
{
    aiMaterial *material = scene->mMaterials[mesh->mMaterialIndex];
    vector<Texture> diffuseMaps = loadMaterialTextures(material,
                                                          aiTextureType_DIFFUSE, "texture_diffuse");
    textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
    vector<Texture> specularMaps = loadMaterialTextures(material,
                                                          aiTextureType_SPECULAR, "texture_specular");
    textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
}
```

我们首先从场景的 `mMaterials` 数组中获取 `aiMaterial` 对象。接下来我们希望加载网格的漫反射和/或镜面光贴图。一个材质对象的内部对每种纹理类型都存储了一个纹理位置数组。不同的纹理类型都以 `aiTextureType_` 为前缀。我们使用一个叫做 `loadMaterialTextures` 的工具函数来从材质中获取纹理。这个函数将会返回一个 `Texture` 结构体的 `vector`，我们将在模型的 `textures` `vector` 的尾部之后存储它。

`loadMaterialTextures` 函数遍历了给定纹理类型的所有纹理位置，获取了纹理的文件位置，并加载并生成了纹理，将信息储存在了一个 `Vertex` 结构体中。它看起来会像这样：

```
vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName)
{
    vector<Texture> textures;
    for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        Texture texture;
        texture.id = TextureFromFile(str.C_Str(), directory);
        texture.type = typeName;
        texture.path = str;
        textures.push_back(texture);
    }
    return textures;
}
```

我们首先通过 `GetTextureCount` 函数检查储存在材质中纹理的数量，这个函数需要一个纹理类型。我们会使用 `GetTexture` 获取每个纹理的文件位置，它会将结果储存在一个 `aiString` 中。我们接下来使用另外一个叫做 `TextureFromFile` 的工具函数，它将会（用 `stb_image.h`）加载一个纹理并返回该纹理的 ID。如果你不确定这样的代码是如何写出来的话，可以查看最后的完整代码。

注意，我们假设了模型文件中纹理文件的路径是相对于模型文件的本地(Local)路径，比如说与模型文件处于同一目录下。我们可以将纹理位置字符串拼接到之前（在 `loadModel` 中）获取的目录字符串上，来获取完整的纹理路径（这也是为什么 `GetTexture` 函数也需要一个目录字符串）。

在网络上找到的某些模型会对纹理位置使用绝对(Absolute)路径，这就不能在每台机器上都工作了。在这种情况下，你可能会需要手动修改这个文件，来让它对纹理使用本地路径（如果可能的话）。

这就是使用 Assimp 导入模型的全部了。

2.4.4 重大优化

这还没有完全结束，因为我们还想做出一个重大的（但不是完全必须的）优化。大多数场景都会在多个网格中重用部分纹理。还是想想一个房子，它的墙壁有着花岗岩的纹理。这个纹理也可以被应用到地板、天花板、楼梯、桌子，甚至是附近的一口井上。加载纹理并不是一个开销不大的操作，在我们当前的实现中，即便同样的纹理已经被加载过很多遍了，对每个网格仍会加载并生成一个新的纹理。这很快就会变成模型加载实现的性能瓶颈。

所以我们会对模型的代码进行调整，将所有加载过的纹理全局储存，每当我们想加载一个纹理的时候，首先去检查它有没有被加载过。如果有的话，我们会直接使用那个纹理，并跳过整个加载流程，来为我们省下很多处理能力。为了能够比较纹理，我们还需要储存它们的路径：

```
struct Texture {
    unsigned int id;
    string type;
    aiString path; // 我们储存纹理的路径用于与其它纹理进行比较
};
```

接下来我们将所有加载过的纹理储存在另一个vector中，在模型类的顶部声明为一个私有变量：

```
vector<Texture> textures_loaded;
```

之后，在`loadMaterialTextures`函数中，我们希望将纹理的路径与储存在`textures_loaded`这个vector中的所有纹理进行比较，看看当前纹理的路径是否与其中的一个相同。如果是的话，则跳过纹理加载/生成的部分，直接使用定位到的纹理结构体为网格的纹理。更新后的函数如下：

```
vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, string typeName)
{
    vector<Texture> textures;
    for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        bool skip = false;
        for(unsigned int j = 0; j < textures_loaded.size(); j++)
        {
            if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
            {
                textures.push_back(textures_loaded[j]);
                skip = true;
                break;
            }
        }
        if(!skip)
        { // 如果纹理还没有被加载，则加载它
            Texture texture;
            texture.id = TextureFromFile(str.C_Str(), directory);
            texture.type = typeName;
            texture.path = str.C_Str();
            textures.push_back(texture);
            textures_loaded.push_back(texture); // 添加到已加载的纹理中
        }
    }
    return textures;
}
```

所以现在我们不仅有了个灵活的模型加载系统，我们也获得了一个加载对象很快的优化版本。

有些版本的Assimp在使用调试版本或者使用IDE的调试模式下加载模型会非常缓慢，所以在你遇到缓慢的加载速度时，可以试试使用发布版本。

你可以在[这里](#)找到优化后Model类的完整源代码。

2.4.5 和箱子模型告别

所以，让我们导入一个由真正的艺术家所创造的模型，替代我这个天才的作品（你要承认，这些箱子可能是你看过的最漂亮的立方体了），测试一下我们的实现吧。由于我不想让我占太多的功劳，我会偶尔让别的艺术家也加入我们，这次我们将会加载Crytek的游戏孤岛危机(Crysis)中的原版[纳米装](#)(Nanosuit)。这个模型被输出为一个.obj文件以及一个.mtl文件，.mtl文件包含了模型的漫反射、镜面光和法线贴图（这个会在后面学习到），你可以在[这里](#)下载到（稍微修改之后的）模型，注意所有的纹理和模型文件应该位于同一个目录下，以供加载纹理。

你从本网站中下载到的版本是修改过的版本，每个纹理的路径都被修改为了一个本地的相对路径，而不是原资源的绝对路径。

现在在代码中，声明一个Model对象，将模型的文件位置传入。接下来模型应该会自动加载并（如果没有错误的话）在渲染循环中使用它的Draw函数来绘制物体，这样就可以了。不再需要缓冲分配、属性指针和渲染指令，只需要一行代码就可以了。接下来如果你创建一系列着色器，其中片段着色器仅仅输出物体的漫反射纹理颜色，最终的结果看上去会是这样的：



你可以在[这里](#)找到完整的源码。

我们可以变得更有创造力一点，根据我们之前在[光照](#)教程中学过的知识，引入两个点光源到渲染方程中，结合镜面光贴图，我们能得到很惊人的效果。



甚至我都必须要承认这个可能是比一直使用的箱子要好看多了。使用Assimp，你能够加载互联网上的无数模型。有很多资源网站都提供了多种格式的免费3D模型供你下载。但还是要注意，有些模型会不能正常地载入，纹理的路径会出现问题，或者Assimp并不支持它的格式。

2.5 高级OpenGL

2.5.1 深度测试

原文 [Depth testing](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

在[坐标系统](#)小节中，我们渲染了一个3D箱子，并且运用了深度缓冲(Depth Buffer)来防止被阻挡的面渲染到其它面的前面。在这一节中，我们将会更加深入地讨论这些储存在深度缓冲（或z缓冲(z-buffer)）中的深度值(Depth Value)，以及它们是如何确定一个片段是处于其它片段后方的。

深度缓冲就像颜色缓冲(Color Buffer)（储存所有的片段颜色：视觉输出）一样，在每个片段中储存了信息，并且（通常）和颜色缓冲有着一样的宽度和高度。深度缓冲是由窗口系统自动创建的，它会以16、24或32位float的形式储存它的深度值。在大部分的系统中，深度缓冲的精度都是24位的。

当深度测试(Depth Testing)被启用的时候，OpenGL会将一个片段的深度值与深度缓冲的内容进行对比。OpenGL会执行一个深度测试，如果这个测试通过了的话，深度缓冲将会更新为新的深度值。如果深度测试失败了，片段将会被丢弃。

深度缓冲是在片段着色器运行之后（以及模板测试(Stencil Testing)运行之后，我们将在[下一节](#)中讨论）在屏幕空间中运行的。屏幕空间坐标与通过OpenGL的`glViewport`所定义的视口密切相关，并且可以直接使用GLSL内建变量`gl_FragCoord`从片段着色器中直接访问。`gl_FragCoord`的x和y分量代表了片段的屏幕空间坐标（其中(0, 0)位于左下角）。`gl_FragCoord`中也包含了一个z分量，它包含了片段真正的深度值。z值就是需要与深度缓冲内容所对比的那个值。

现在大部分的GPU都提供一个叫做提前深度测试(Early Depth Testing)的硬件特性。提前深度测试允许深度测试在片段着色器之前运行。只要我们清楚一个片段永远不会是可见的（它在其他物体之后），我们就能提前丢弃这个片段。片段着色器通常开销都是很大的，所以我们应该尽可能避免运行它们。当使用提前深度测试时，片段着色器的一个限制是你不能写入片段的深度值。如果一个片段着色器对它的深度值进行了写入，提前深度测试是不可能的。OpenGL不能提前知道深度值。

深度测试默认是禁用的，所以如果要启用深度测试的话，我们需要用`GL_DEPTH_TEST`选项来启用它：

```
glEnable(GL_DEPTH_TEST);
```

当它启用的时候，如果一个片段通过了深度测试的话，OpenGL会在深度缓冲中储存该片段的z值；如果没有通过深度缓冲，则会丢弃该片段。如果你启用了深度缓冲，你还应该在每个渲染迭代之前使用`GL_DEPTH_BUFFER_BIT`来清除深度缓冲，否则你会仍在使用上一次渲染迭代中的写入的深度值：

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

可以想象，在某些情况下你会需要对所有片段都执行深度测试并丢弃相应的片段，但不希望更新深度缓冲。基本上来说，你在使用一个只读的(Read-only)深度缓冲。OpenGL允许我们禁用深度缓冲的写入，只需要设置它的深度掩码(Depth Mask)设置为 `GL_FALSE` 就可以了：

```
glDepthMask(GL_FALSE);
```

注意这只在深度测试被启用的时候才有效果。

深度测试函数

OpenGL允许我们修改深度测试中使用的比较运算符。这允许我们来控制OpenGL什么时候该通过或丢弃一个片段，什么时候去更新深度缓冲。我们可以调用 `glDepthFunc` 函数来设置比较运算符（或者说深度函数(Depth Function)）：

```
glDepthFunc(GL_LESS);
```

这个函数接受下面表格中的比较运算符：

函数	描述
<code>GL_ALWAYS</code>	永远通过深度测试
<code>GL_NEVER</code>	永远不通过深度测试
<code>GL_LESS</code>	在片段深度值小于缓冲的深度值时通过测试
<code>GL_EQUAL</code>	在片段深度值等于缓冲区的深度值时通过测试
<code>GL_EQUAL</code>	在片段深度值小于等于缓冲区的深度值时通过测试
<code>GL_GREATER</code>	在片段深度值大于缓冲区的深度值时通过测试
<code>GL_NOTEQUAL</code>	在片段深度值不等于缓冲区的深度值时通过测试
<code>GL_GEQUAL</code>	在片段深度值大于等于缓冲区的深度值时通过测试

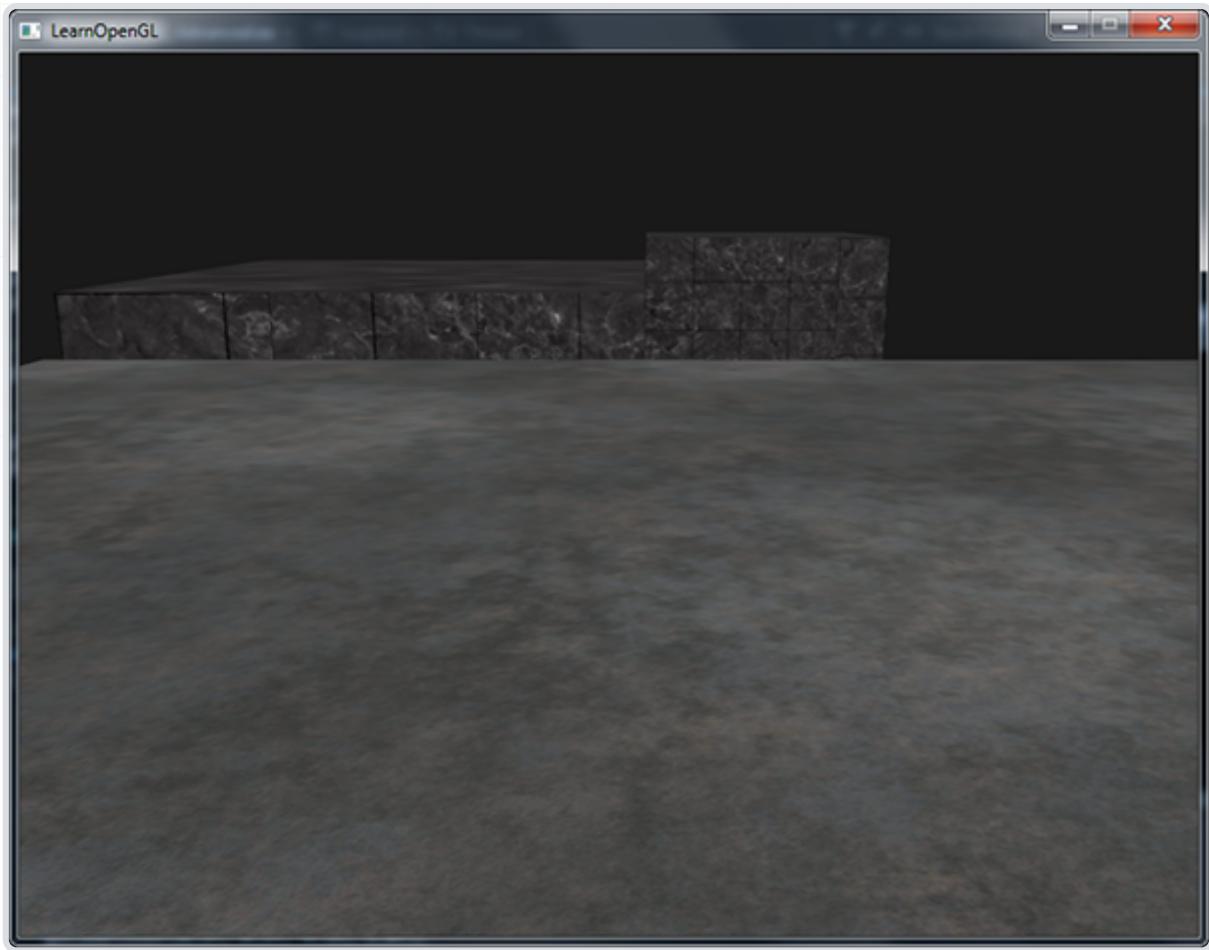
默认情况下使用的深度函数是 `GL_LESS`，它将会丢弃深度值大于等于当前深度缓冲值的所有片段。

让我们看看改变深度函数会对视觉输出有什么影响。我们将使用一个新的代码配置，它会显示一个没有光照的基本场景，里面有两个有纹理的立方体，放置在一个有纹理的地板上。你可以在[这里](#)找到源代码。

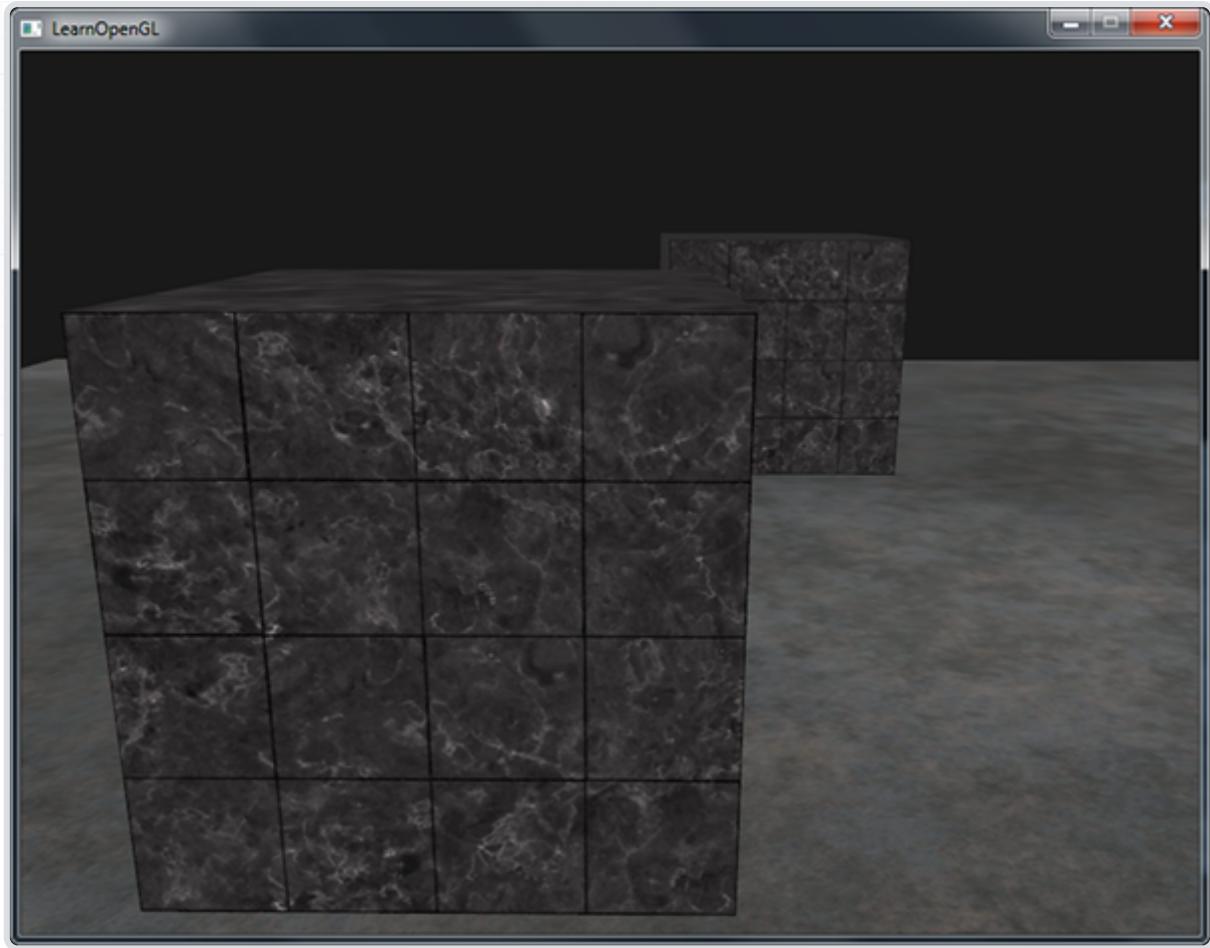
在源代码中，我们将深度函数改为 `GL_ALWAYS`：

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_ALWAYS);
```

这将会模拟我们没有启用深度测试时所得到的结果。深度测试将会永远通过，所以最后绘制的片段将会总是会渲染在之前绘制片段的上面，即使之前绘制的片段本就应该渲染在最前面。因为我们是最后渲染地板的，它会覆盖所有的箱子片段：



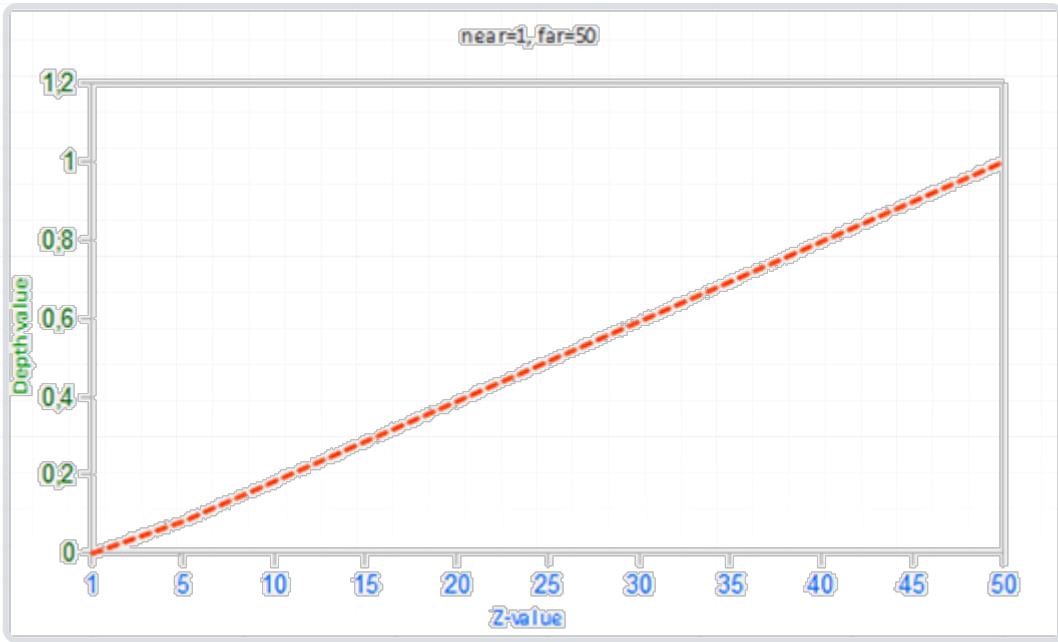
将它重新设置为 `GL_LESS`, 这会将场景还原为原有的样子:



深度值精度

深度缓冲包含了一个介于0.0和1.0之间的深度值，它将会与观察者视角所看见的场景中所有物体的z值进行比较。观察空间的z值可能是投影平截头体的近平面(Near)和远平面(Far)之间的任何值。我们需要一种方式来将这些观察空间的z值变换到[0, 1]范围之间，其中的一种方式就是将它们线性变换到[0, 1]范围之间。下面这个（线性）方程将z值变换到了0.0到1.0之间的深度值：

这里的 z 值是我们之前提供给投影矩阵设置可视平截头体的（见[坐标系统](#)）那个 *near* 和 *far* 值。这个方程需要平截头体中的一个 z 值，并将它变换到了[0, 1]的范围内。 z 值和对应的深度值之间的关系可以在下图中看到：

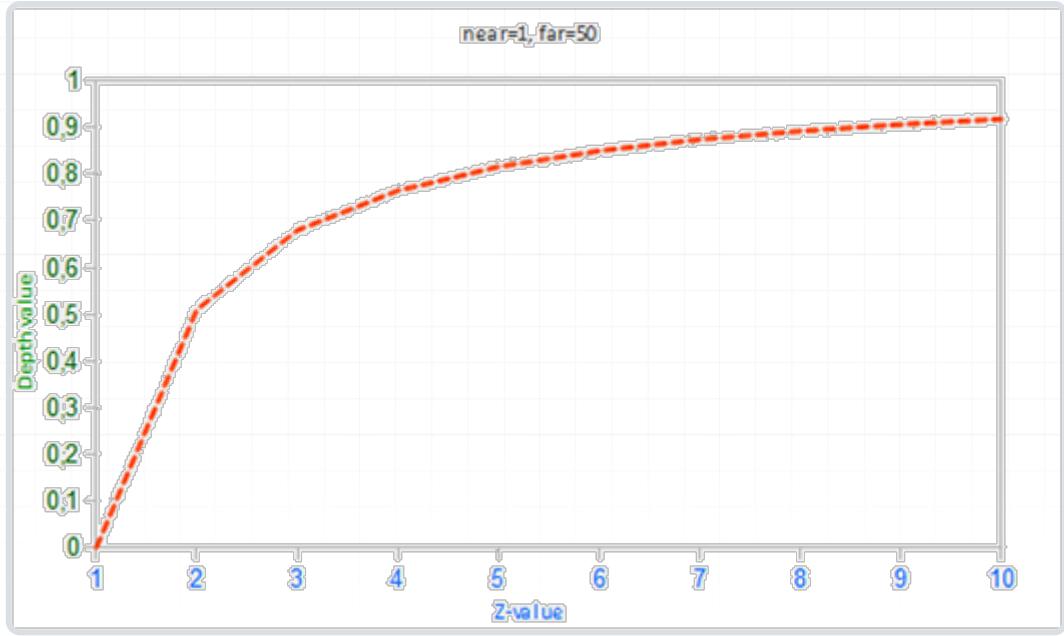


注意所有的方程都会将非常近的物体的深度值设置为接近0.0的值，而当物体非常接近远平面的时候，它的深度值会非常接近1.0。

然而，在实践中是几乎永远不会使用这样的线性深度缓冲(Linear Depth Buffer)的。要想有正确的投影性质，需要使用一个非线性的深度方程，它是与 $1/z$ 成正比的。它做的就是在z值很小的时候提供非常好的精度，而在z值很远的时候提供更少的精度。花时间想想这个：我们真的需要对1000单位远的深度值和只有1单位远的充满细节的物体使用相同的精度吗？线性方程并不会考虑这一点。

由于非线性方程与 $1/z$ 成正比，在1.0和2.0之间的z值将会变换至1.0到0.5之间的深度值，这就是一个float提供给我们的一半精度了，这在z值很小的情况下提供了非常大的精度。在50.0和100.0之间的z值将会只占2%的float精度，这正是我们所需要的。这样的一个考虑了远近距离的方程是这样的：

如果你不知道这个方程是怎么回事也不用担心。重要的是要记住深度缓冲中的值在屏幕空间中不是线性的（在透视矩阵应用之前在观察空间中是线性的）。深度缓冲中0.5的值并不代表着物体的z值是位于平截头体的中间了，这个顶点的z值实际上非常接近近平面！你可以在下图中看到z值和最终的深度缓冲值之间的非线性关系：



可以看到，深度值很大一部分是由很小的z值所决定的，这给了近处的物体很大的深度精度。这个（从观察者的视角）变换z值的方程是嵌入在投影矩阵中的，所以当我们想将一个顶点坐标从观察空间至裁剪空间的时候这个非线性方程就被应用了。如果你想深度了解投影矩阵究竟做了什么，我建议阅读[这篇文章](#)。

如果我们想要可视化深度缓冲的话，非线性方程的效果很快就会变得很清楚。

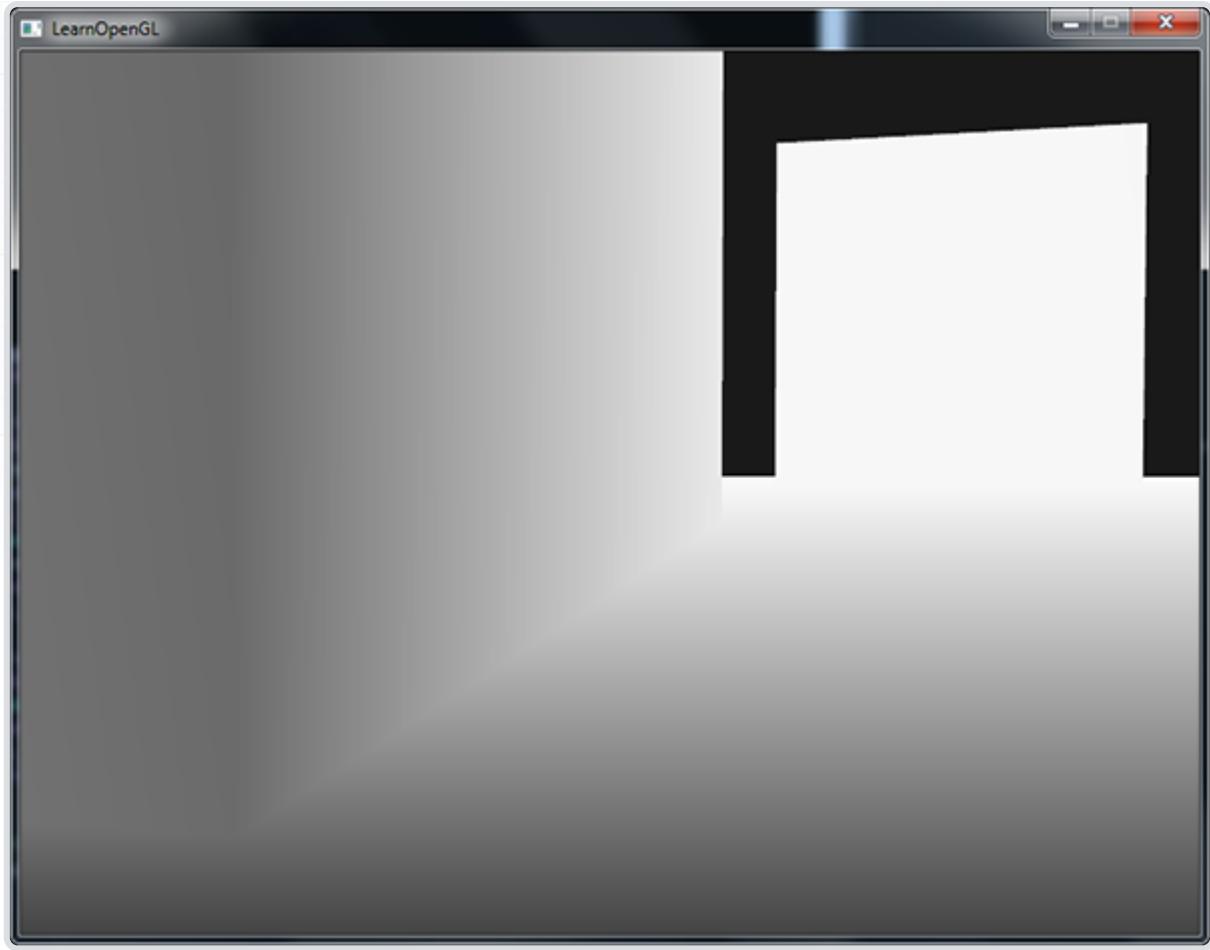
深度缓冲的可视化

我们知道片段着色器中，内建`gl_FragCoord`向量的z值包含了那个特定片段的深度值。如果我们将这个深度值输出为颜色，我们可以显示场景中所有片段的深度值。我们可以根据片段的深度值返回一个颜色向量来完成这一工作：

```
void main()
{
    FragColor = vec4(vec3(gl_FragCoord.z), 1.0);
}
```

如果你再次运行程序的话，你可能会注意到所有东西都是白色的，看起来就想我们所有的深度值都是最大的1.0。所以为什么没有靠近0.0（即变暗）的深度值呢？

你可能还记得在上一部分中说到，屏幕空间中的深度值是非线性的，即它在z值很小的时候有很高的精度，而z值很大的时候有较低的精度。片段的深度值会随着距离迅速增加，所以几乎所有的顶点的深度值都是接近于1.0的。如果我们小心地靠近物体，你可能会最终注意到颜色会渐渐变暗，显示它们的z值在逐渐变小：



这很清楚地展示了深度值的非线性性质。近处的物体比起远处的物体对深度值有着更大的影响。只需要移动几厘米就能让颜色从暗完全变白。

然而，我们也可以让片段非线性的深度值变换为线性的。要实现这个，我们需要仅仅反转深度值的投影变换。这也就意味着我们需要首先将深度值从 $[0, 1]$ 范围重新变换到 $[-1, 1]$ 范围的标准化设备坐标（裁剪空间）。接下来我们需要像投影矩阵那样反转这个非线性方程（方程2），并将这个反转的方程应用到最终的深度值上。最终的结果就是一个线性的深度值了。听起来是可行的，对吧？

首先我们将深度值变换为NDC，不是非常困难：

```
float z = depth * 2.0 - 1.0;
```

接下来使用获取到的z值，应用逆变换来获取线性的深度值：

```
float linearDepth = (2.0 * near * far) / (far + near - z * (far - near));
```

这个方程是用投影矩阵推导得出的，它使用了方程2来非线性化深度值，返回一个`near`与`far`之间的深度值。这篇注重数学的[文章](#)为感兴趣的读者详细解释了投影矩阵，它也展示了这些方程是怎么来的。

将屏幕空间中非线性的深度值变换至线性深度值的完整片段着色器如下：

```
#version 330 core
out vec4 FragColor;

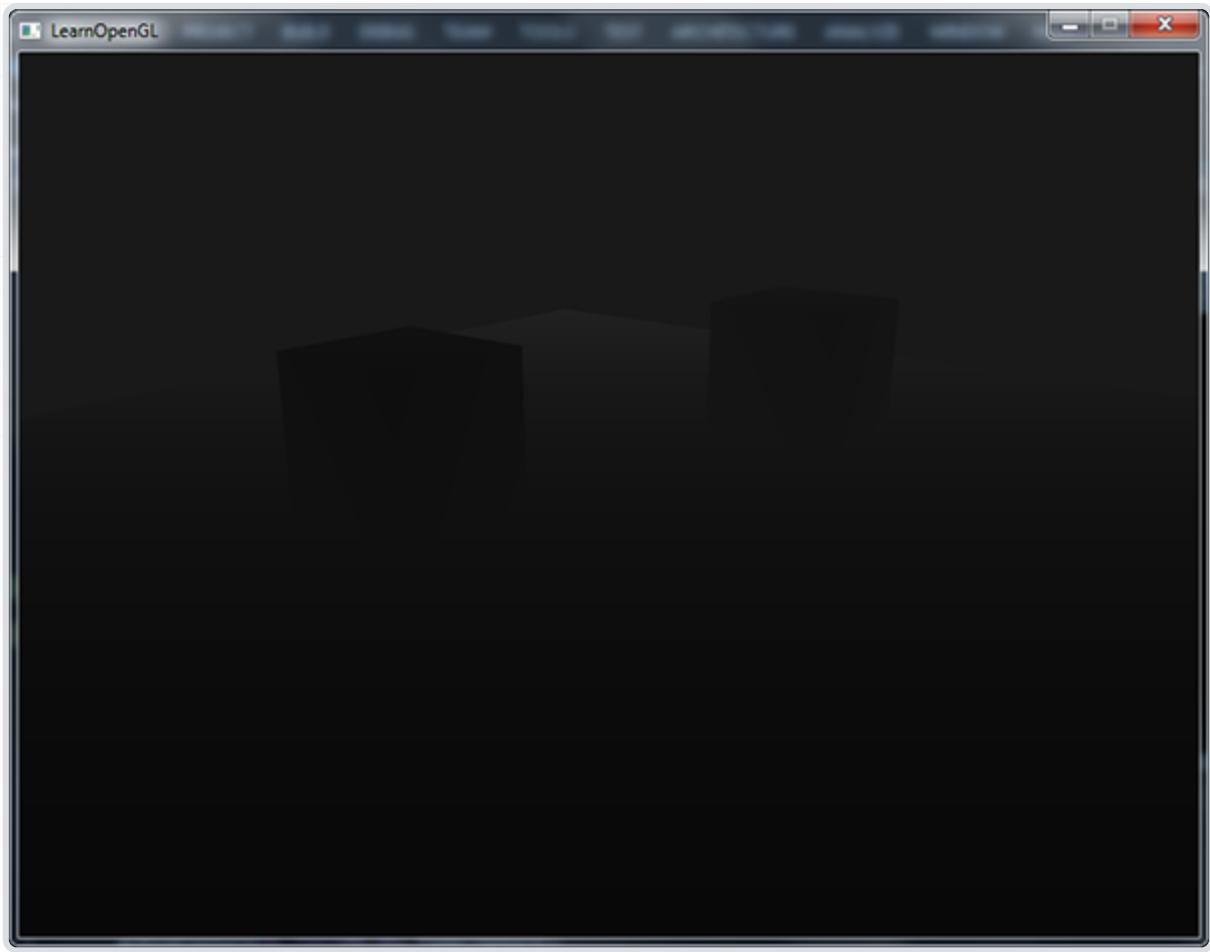
float near = 0.1;
float far = 100.0;

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // back to NDC
    return (2.0 * near * far) / (far + near - z * (far - near));
}

void main()
{
    float depth = LinearizeDepth(gl_FragCoord.z) / far; // 为了演示除以 far
    FragColor = vec4(vec3(depth), 1.0);
}
```

由于线性化的深度值处于`near`与`far`之间，它的大部分值都会大于1.0并显示为完全的白色。通过在`main`函数中将线性深度值除以`far`，我们近似地将线性深度值转化到[0, 1]的范围之间。这样子我们就能逐渐看到一个片段越接近投影平截头体的远平面，它就会变得越亮，更适用于展示目的。

如果我们现在运行程序，我们就能看见深度值随着距离增大是线性的了。尝试在场景中移动，看看深度值是怎样以线性变化的。



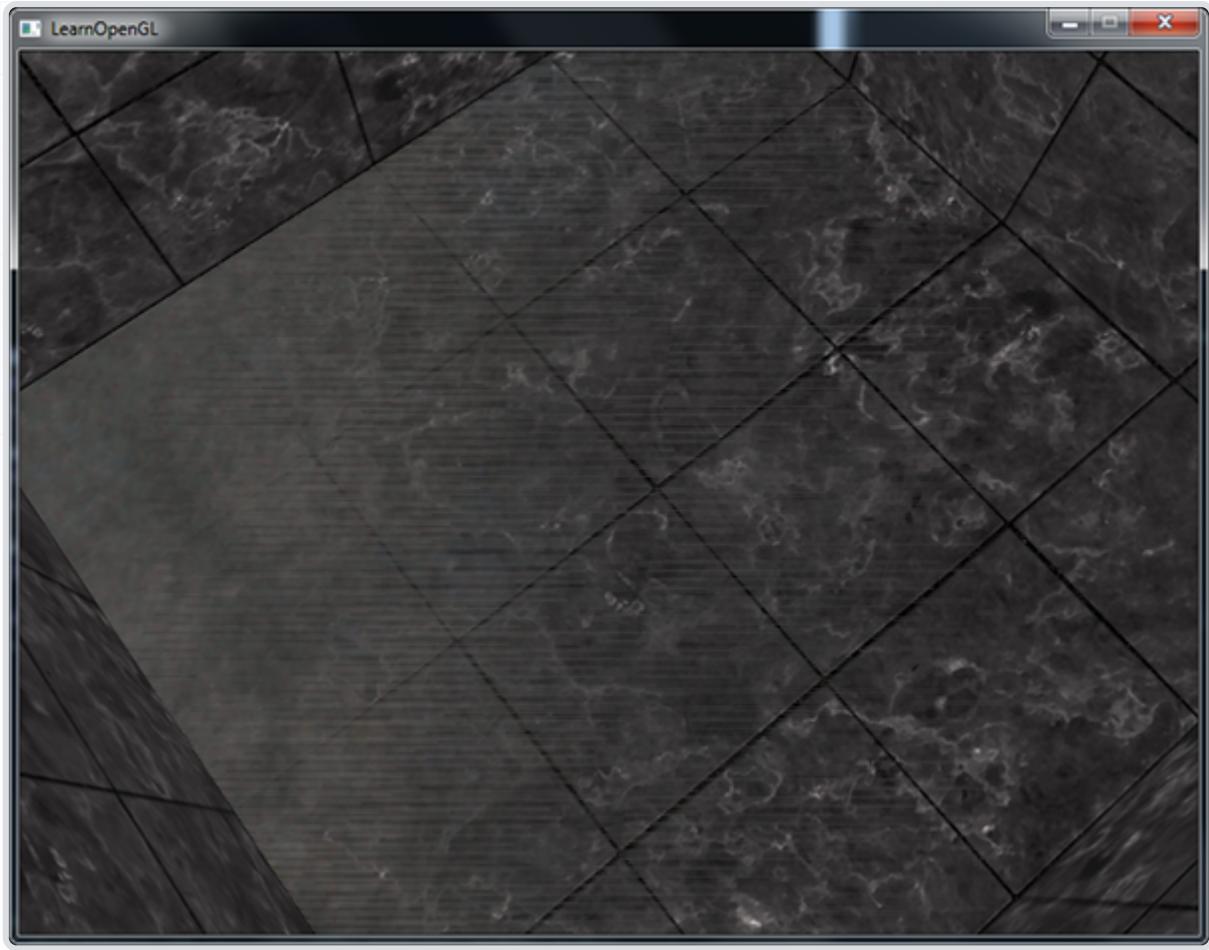
颜色大部分都是黑色，因为深度值的范围是0.1的近平面到100的远平面，它离我们还是非常远的。结果就是，我们相对靠近近平面，所以会得到更低的（更暗的）深度值。

深度冲突

一个很常见的视觉错误会在两个平面或者三角形非常紧密地平行排列在一起时会发生，深度缓冲没有足够的精度来决定两个形状哪个在前面。结果就是这两个形状不断地在切换前后顺序，这会导致很奇怪的花纹。这个现象叫做**深度冲突**(Z-fighting)，因为它看起来像是这两个形状在争夺(Fight)谁该处于顶端。

在我们一直使用的场景中，有几个地方的深度冲突还是非常明显的。箱子被放置在地板的同一高度上，这也就意味着箱子的底面和地板是共面的(Coplanar)。这两个面的深度值都是一样的，所以深度测试没有办法决定应该显示哪一个。

如果你将摄像机移动到其中一个箱子的内部，你就能清楚地看到这个效果的，箱子的底部不断地在箱子底面与地板之间切换，形成一个锯齿的花纹：



深度冲突是深度缓冲的一个常见问题，当物体在远处时效果会更明显（因为深度缓冲在z值比较大的时候有着更小的精度）。深度冲突不能够被完全避免，但一般会有一些技巧有助于在你的场景中减轻或者完全避免深度冲突。

防止深度冲突

第一个也是最重要的技巧是永远不要把多个物体摆得太靠近，以至于它们的一些三角形会重叠。通过在两个物体之间设置一个用户无法注意到的偏移值，你可以完全避免这两个物体之间的深度冲突。在箱子和地板的例子中，我们可以将箱子沿着正y轴稍微移动一点。箱子位置的这点微小改变将不太可能被注意到，但它能够完全减少深度冲突的发生。然而，这需要对每个物体都手动调整，并且需要进行彻底的测试来保证场景中没有物体会产生深度冲突。

第二个技巧是尽可能将近平面设置远一些。在前面我们提到了精度在靠近近平面时是非常高的，所以如果我们将近平面远离观察者，我们将会对整个平截头体有着更大的精度。然而，将近平面设置太远将会导致近处的物体被裁剪掉，所以这通常需要实验和微调来决定最适合你的场景的近平面距离。

另外一个很好的技巧是牺牲一些性能，使用更高精度的深度缓冲。大部分深度缓冲的精度都是24位的，但现在大部分的显卡都支持32位的深度缓冲，这将会极大地提高精度。所以，牺牲掉一些性能，你就能获得更高精度的深度测试，减少深度冲突。

我们上面讨论的三个技术是最普遍也很容易实现的抗深度冲突技术了。还有一些更复杂的技术，但它们依然不能完全消除深度冲突。深度冲突是一个常见的问题，但如果你组合使用了上面列举出来的技术，你可能不会再需要处理深度冲突了。

2.5.2 模板测试

原文 [Stencil testing](#)

作者 JoeyDeVries

翻译 Krasjet

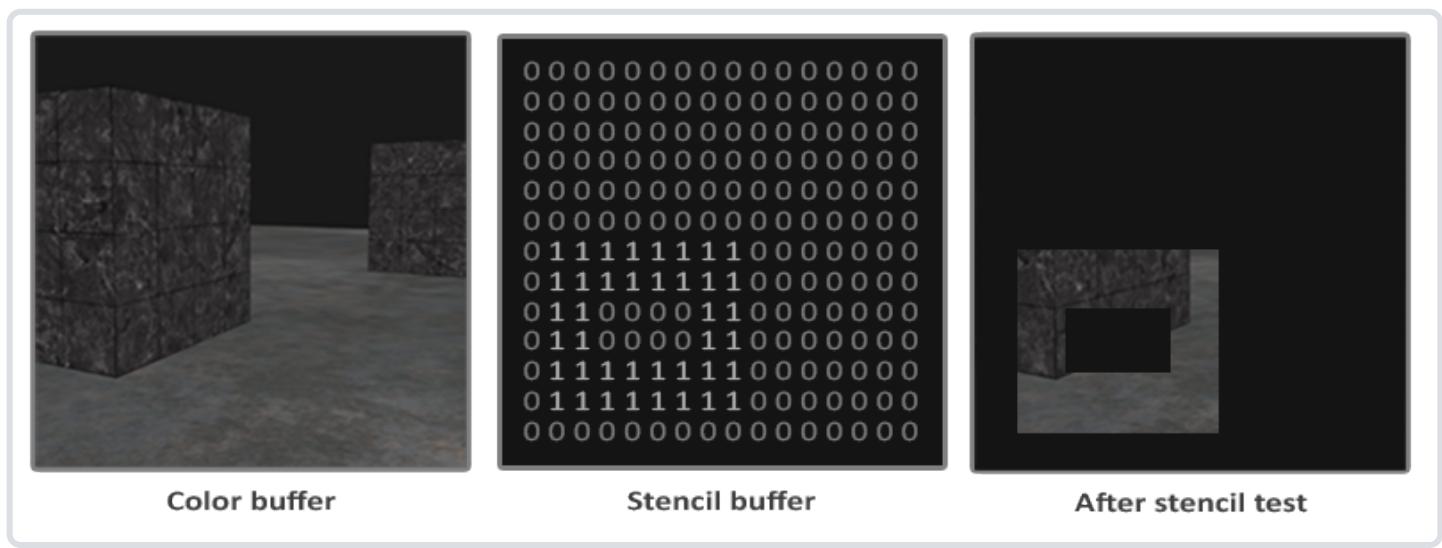
校对 暂未校对

当片段着色器处理完一个片段之后，**模板测试**(Stencil Test)会开始执行，和深度测试一样，它也可能会丢弃片段。接下来，被保留的片段会进入深度测试，它可能会丢弃更多的片段。模板测试是根据另一个缓冲来进行的，它叫做**模板缓冲**(Stencil Buffer)，我们可以在渲染的时候更新它来获得一些很有意思的效果。

一个模板缓冲中，(通常)每个**模板值**(Stencil Value)是8位的。所以每个像素/片段一共能有256种不同的模板值。我们可以将这些模板值设置为我们想要的值，然后当某一个片段有某一个模板值的时候，我们就可以选择丢弃或是保留这个片段了。

每个窗口库都需要为你配置一个模板缓冲。GLFW自动做了这件事，所以我们不需要告诉GLFW来创建一个，但其它的窗口库可能不会默认给你创建一个模板库，所以记得要查看库的文档。

模板缓冲的一个简单的例子如下：



模板缓冲首先会被清除为0，之后在模板缓冲中使用1填充了一个空心矩形。场景中的片段将会只在片段的模板值为1的时候会被渲染（其它的都被丢弃了）。

模板缓冲操作允许我们在渲染片段时将模板缓冲设定为一个特定的值。通过在渲染时修改模板缓冲的内容，我们写入了模板缓冲。在同一个（或者接下来的）渲染迭代中，我们可以读取这些值，来决定丢弃还是保留某个片段。使用模板缓冲的时候你可以尽情发挥，但大体的步骤如下：

- 启用模板缓冲的写入。
- 渲染物体，更新模板缓冲的内容。
- 禁用模板缓冲的写入。
- 渲染（其它）物体，这次根据模板缓冲的内容丢弃特定的片段。

所以，通过使用模板缓冲，我们可以根据场景中已绘制的其它物体的片段，来决定是否丢弃特定的片段。

你可以启用`GL_STENCIL_TEST`来启用模板测试。在这一行代码之后，所有的渲染调用都会以某种方式影响着模板缓冲。

```
glEnable(GL_STENCIL_TEST);
```

注意，和颜色和深度缓冲一样，你也需要在每次迭代之前清除模板缓冲。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

和深度测试的`glDepthMask`函数一样，模板缓冲也有一个类似的函数。`glStencilMask`允许我们设置一个位掩码(Bitmask)，它会与将要写入缓冲的模板值进行与(AND)运算。默认情况下设置的位掩码所有位都为1，不影响输出，但如果我们将它设置为`0x00`，写入缓冲的所有模板值最后都会变成0.这与深度测试中的`glDepthMask(GL_FALSE)`是等价的。

```
glStencilMask(0xFF); // 每一位写入模板缓冲时都保持原样  
glStencilMask(0x00); // 每一位在写入模板缓冲时都会变成0 (禁用写入)
```

大部分情况下你都只会使用`0x00`或者`0xFF`作为模板掩码(Stencil Mask)，但是知道有选项可以设置自定义的位掩码总是好的。

模板函数

和深度测试一样，我们对模板缓冲应该通过还是失败，以及它应该如何影响模板缓冲，也是有一定控制的。一共有两个函数能够用来配置模板测试：`glStencilFunc`和`glStencilOp`。

`glStencilFunc(GLenum func, GLint ref, GLuint mask)`一共包含三个参数：

- `func`：设置模板测试函数(Stencil Test Function)。这个测试函数将会应用到已储存的模板值上和`glStencilFunc`函数的`ref`值上。可用的选项有：`GL_NEVER`、`GL_LESS`、`GL_EQUAL`、`GL_GREATER`、`GL_GEQUAL`、`GL_NOTEQUAL`和`GL_ALWAYS`。它们的语义和深度缓冲的函数类似。
- `ref`：设置了模板测试的参考值(Reference Value)。模板缓冲的内容将会与这个值进行比较。
- `mask`：设置一个掩码，它将会与参考值和储存的模板值在测试比较它们之前进行与(AND)运算。初始情况下所有位都为1。

在一开始的那个简单的模板例子中，函数被设置为：

```
glStencilFunc(GL_EQUAL, 1, 0xFF)
```

这会告诉OpenGL，只要一个片段的模板值等于(`GL_EQUAL`)参考值1，片段将会通过测试并被绘制，否则会被丢弃。

但是`glStencilFunc`仅仅描述了OpenGL应该对模板缓冲内容做什么，而不是我们应该如何更新缓冲。这就需要`glStencilOp`这个函数了。

`glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)`一共包含三个选项，我们能够设定每个选项应该采取的行为：

- `sfail`：模板测试失败时采取的行为。
- `dpfail`：模板测试通过，但深度测试失败时采取的行为。
- `dppass`：模板测试和深度测试都通过时采取的行为。

每个选项都可以选用以下的其中一种行为：

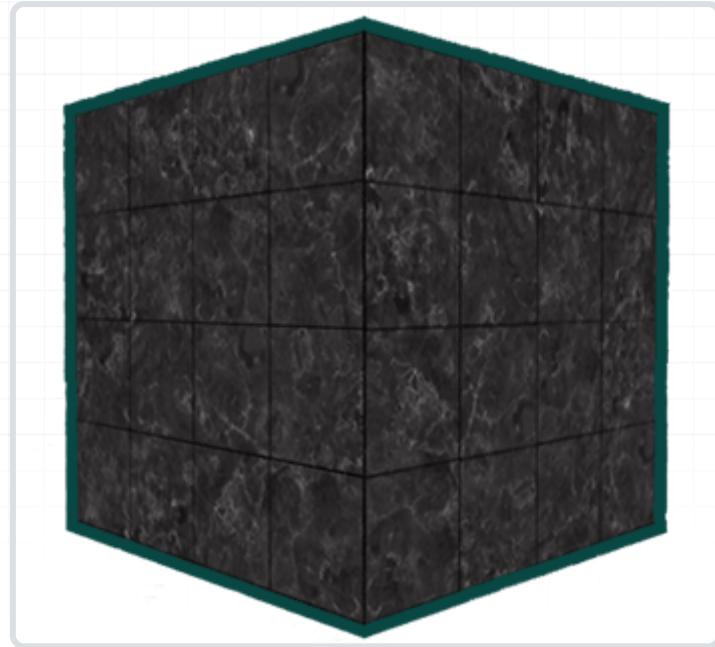
行为	描述
GL_KEEP	保持当前储存的模板值
GL_ZERO	将模板值设置为0
GL_REPLACE	将模板值设置为 <code>glStencilFunc</code> 函数设置的 <code>ref</code> 值
GL_INCR	如果模板值小于最大值则将模板值加1
GL_INCR_WRAP	与 <code>GL_INCR</code> 一样，但如果模板值超过了最大值则归零
GL_DECR	如果模板值大于最小值则将模板值减1
GL_DECWRAP	与 <code>GL_DECR</code> 一样，但如果模板值小于0则将其设置为最大值
GL_INVERT	按位翻转当前的模板缓冲值

默认情况下`glStencilOp`是设置为`(GL_KEEP, GL_KEEP, GL_KEEP)`的，所以不论任何测试的结果是如何，模板缓冲都会保留它的值。默认的行为不会更新模板缓冲，所以如果你想写入模板缓冲的话，你需要至少对其中一个选项设置不同的值。

所以，通过使用`glStencilFunc`和`glStencilOp`，我们可以精确地指定更新模板缓冲的时机与行为了，我们也可以指定什么时候该让模板缓冲通过，即什么时候片段需要被丢弃。

2.5.3 物体轮廓

仅仅看了前面的部分你还是不太可能能够完全理解模板测试的工作原理，所以我们将会展示一个使用模板测试就可以完成的有用特性，它叫做**物体轮廓**(Object Outlining)。



物体轮廓所能做的事情正如它名字所描述的那样。我们将会为每个（或者一个）物体在它的周围创建一个很小的有色边框。当你想要在策略游戏中选中一个单位进行操作的，想要告诉玩家选中的是哪个单位的时候，这个效果就非常有用了。为物体创建轮廓的步骤如下：

1. 在绘制（需要添加轮廓的）物体之前，将模板函数设置为 `GL_ALWAYS`，每当物体的片段被渲染时，将模板缓冲更新为1。
2. 渲染物体。
3. 禁用模板写入以及深度测试。
4. 将每个物体缩放一点点。
5. 使用一个不同的片段着色器，输出一个单独的（边框）颜色。
6. 再次绘制物体，但只在它们片段的模板值不等于1时才绘制。
7. 再次启用模板写入和深度测试。

这个过程将每个物体的片段的模板缓冲设置为1，当我们想要绘制边框的时候，我们主要绘制放大版本的物体中模板测试通过的部分，也就是物体的边框的位置。我们主要使用模板缓冲丢弃了放大版本中属于原物体片段的部分。

所以我们首先来创建一个很简单的片段着色器，它会输出一个边框颜色。我们简单地给它设置一个硬编码的颜色值，将这个着色器命名为 `shaderSingleColor`：

```
void main()
{
    FragColor = vec4(0.04, 0.28, 0.26, 1.0);
}
```

我们只想给那两个箱子加上边框，所以我们让地板不参与这个过程。我们希望首先绘制地板，再绘制两个箱子（并写入模板缓冲），之后绘制放大的箱子（并丢弃覆盖了之前绘制的箱子片段的那些片段）。

我们首先启用模板测试，并设置测试通过或失败时的行为：

```
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
```

如果其中的一个测试失败了，我们什么都不做，我们仅仅保留当前储存在模板缓冲中的值。如果模板测试和深度测试都通过了，那么我们希望将储存的模板值设置为参考值，参考值能够通过 `glStencilFunc` 来设置，我们之后会设置为1。

我们将模板缓冲清除为0，对箱子中所有绘制的片段，将模板值更新为1：

```
glStencilFunc(GL_ALWAYS, 1, 0xFF); // 所有的片段都应该更新模板缓冲
glStencilMask(0xFF); // 启用模板缓冲写入
normalShader.use();
DrawTwoContainers();
```

通过使用`GL_ALWAYS`模板测试函数，我们保证了箱子的每个片段都会将模板缓冲的模板值更新为1。因为片段永远会通过模板测试，在绘制片段的地方，模板缓冲会被更新为参考值。

现在模板缓冲在箱子被绘制的地方都更新为1了，我们将要绘制放大的箱子，但这次要禁用模板缓冲的写入：

```
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00); // 禁止模板缓冲的写入
glDisable(GL_DEPTH_TEST);
shaderSingleColor.use();
DrawTwoScaledUpContainers();
```

我们将模板函数设置为`GL_NOTEQUAL`，它会保证我们只绘制箱子上模板值不为1的部分，即只绘制箱子在之前绘制的箱子之外的部分。注意我们也禁用了深度测试，让放大的箱子，即边框，不会被地板所覆盖。

记得要在完成之后重新启用深度缓冲。

场景中物体轮廓的完整步骤会看起来像这样：

```
 glEnable(GL_DEPTH_TEST);
 glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

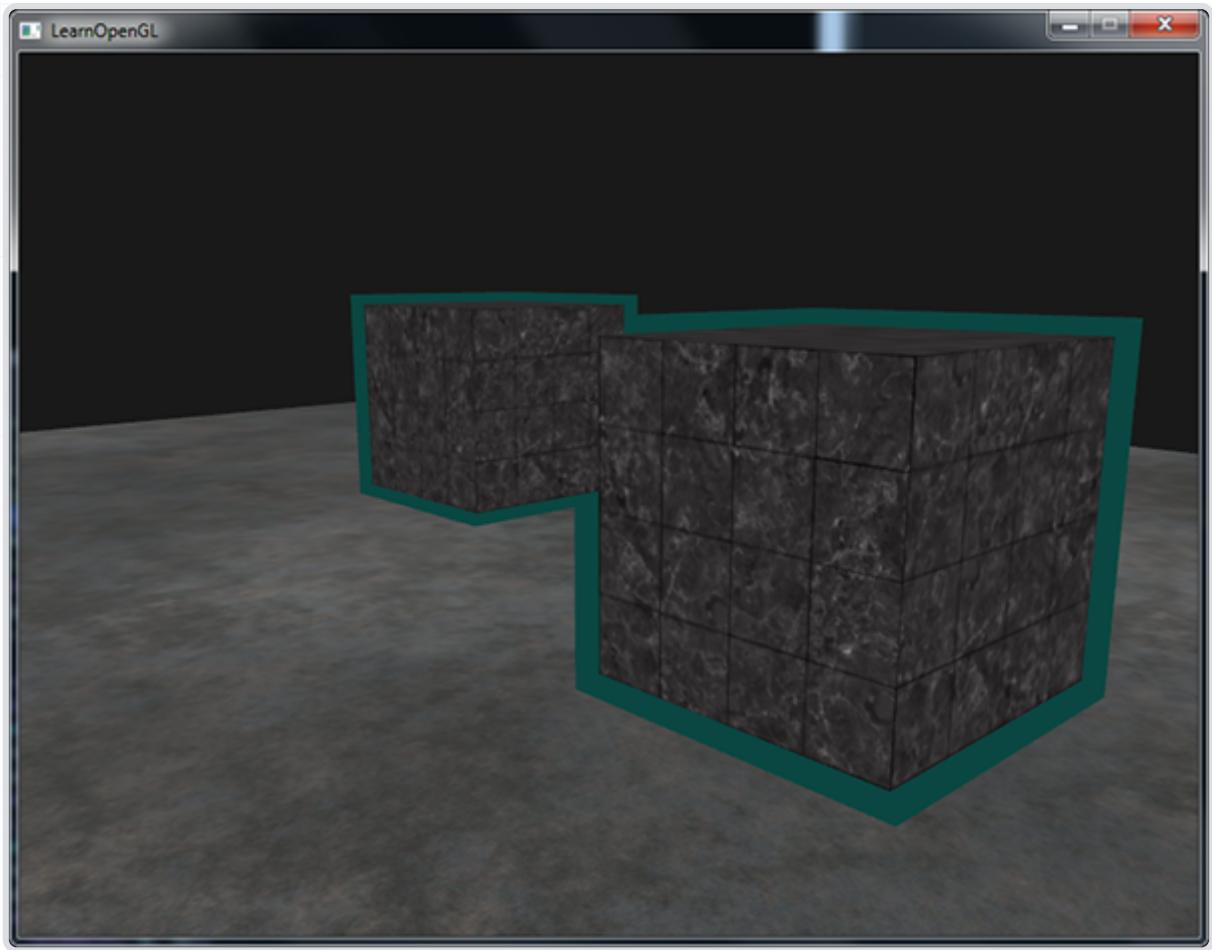
 glStencilMask(0x00); // 记得保证我们在绘制地板的时候不会更新模板缓冲
 normalShader.use();
 DrawFloor()

 glStencilFunc(GL_ALWAYS, 1, 0xFF);
 glStencilMask(0xFF);
 DrawTwoContainers();

 glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
 glStencilMask(0x00);
 glDisable(GL_DEPTH_TEST);
 shaderSingleColor.use();
 DrawTwoScaledUpContainers();
 glStencilMask(0xFF);
 glEnable(GL_DEPTH_TEST);
```

只要你理解了模板缓冲背后的大体思路，这个代码片段就不是那么难理解了。如果还是不能理解的话，尝试再次仔细阅读之前的部分，并尝试通过上面使用的范例，完全理解每个函数的功能。

在[深度测试](#)小节的场景中，这个轮廓算法的结果看起来会像是这样的：



可以在[这里](#)查看源代码，看看物体轮廓算法的完整代码。

你可以看到这两个箱子的边框重合了，这通常都是我们想要的结果（想想策略游戏中，我们希望选择10个单位，合并边框通常是我们想需要的结果）。如果你想让每个物体都有一个完整的边框，你需要对每个物体都清空模板缓冲，并有创意地利用深度缓冲。

你看到的物体轮廓算法在需要显示选中物体的游戏（想想策略游戏）中非常常见。这样的算法能够在一个模型类中轻松实现。你可以在模型类中设置一个boolean标记，来设置需不需要绘制边框。如果你有创造力的话，你也可以使用后期处理滤镜(Filter)，像是高斯模糊(Gaussian Blur)，让边框看起来更自然。

除了物体轮廓之外，模板测试还有很多用途，比如在一个后视镜中绘制纹理，让它能够绘制到镜子形状中，或者使用一个叫做阴影体积(Shadow Volume)的模板缓冲技术渲染实时阴影。模板缓冲为我们已经很丰富的OpenGL工具箱又提供了一个很好的工具。

2.5.4 混合

原文 [Blending](#)

作者 JoeyDeVries

翻译 Krasjet, [Django](#)

校对 暂无校对

OpenGL中，[混合](#)(Blending)通常是实现物体[透明度](#)(Transparency)的一种技术。透明就是说一个物体（或者其中的一部分）不是纯色(Solid Color)的，它的颜色是物体本身的颜色和它背后其它物体的颜色的不同强度结合。一个有色玻璃窗是一个透明的物体，玻璃有它自己的颜色，但它最终的颜色还包含了玻璃之后所有物体的颜色。这也是混合这一名字的出处，我们[混合](#)(Blend)（不同物体的）多种颜色为一种颜色。所以透明度能让我们看穿物体。



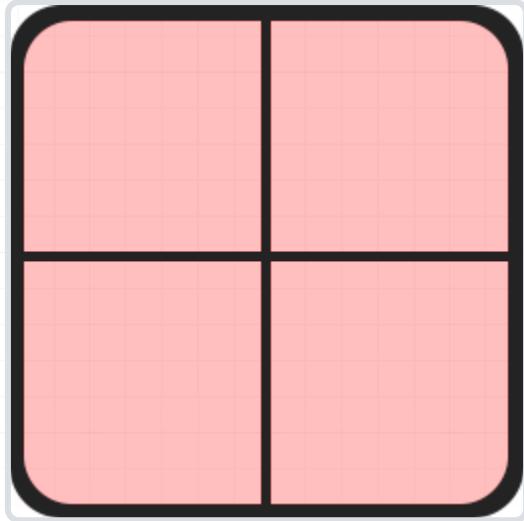
Full transparent window



Partially transparent window

透明的物体可以是完全透明的（让所有的颜色穿过），或者是半透明的（它让颜色通过，同时也会显示自身的颜色）。一个物体的透明度是通过它颜色的[alpha](#)值来决定的。Alpha颜色值是颜色向量的第四个分量，你可能已经看到过它很多遍了。在这个教程之前我们都将这个第四个分量设置为1.0，让这个物体的透明度为0.0，而当alpha值为0.0时物体将会是完全透明的。当alpha值为0.5时，物体的颜色有50%是来自物体自身的颜色，50%来自背后物体的颜色。

我们目前一直使用的纹理有三个颜色分量：红、绿、蓝。但一些材质会有一个内嵌的alpha通道，对每个纹素(Texel)都包含了一个alpha值。这个alpha值精确地告诉我们纹理各个部分的透明度。比如说，下面这个[窗户纹理](#)中的玻璃部分的alpha值为0.25（它在一般情况下是完全的红色，但由于它有75%的透明度，能让很大一部分的网站背景颜色穿过，让它看起来不那么红了），角落的alpha值是0.0。



我们很快就会将这个窗户纹理添加到场景中，但是首先我们需要讨论一个更简单的技术，来实现只有完全透明和完全不透明的纹理的透明度。

丢弃片段

有些图片并不需要半透明，只需要根据纹理颜色值，显示一部分，或者不显示一部分，没有中间情况。比如说草，如果想不太费劲地创建草这种东西，你需要将一个草的纹理贴在一个2D四边形(Quad)上，然后将这个四边形放到场景中。然而，草的形状和2D四边形的形状并不完全相同，所以你只想显示草纹理的某些部分，而忽略剩下的部分。

下面这个纹理正是这样的，它要么是完全不透明的（alpha值为1.0），要么是完全透明的（alpha值为0.0），没有中间情况。你可以看到，只要不是草的部分，这个图片显示的都是网站的背景颜色而不是它本身的颜色。



所以当添加像草这样的植被到场景中时，我们不希望看到草的方形图像，而是只显示草的部分，并能看透图像其余的部分。我们想要丢弃(Discard)显示纹理中透明部分的片段，不将这些片段存储到颜色缓冲中。在此之前，我们还要学习如何加载一个透明的纹理。

要想加载有alpha值的纹理，我们并不需要改很多东西，`stb_image`在纹理有alpha通道的时候会自动加载，但我们仍要在纹理生成过程中告诉OpenGL，我们的纹理现在使用alpha通道了：

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

同样，保证你在片段着色器中获取了纹理的全部4个颜色分量，而不仅仅是RGB分量：

```
void main()
{
    // FragColor = vec4(texture(texture1, TexCoords), 1.0);
    FragColor = texture(texture1, TexCoords);
}
```

既然我们已经知道该如何加载透明的纹理了，是时候将它带入实战了，我们将会在[深度测试](#)小节的场景中加入几棵草。

我们会创建一个vector，向里面添加几个 `glm::vec3` 变量来代表草的位置：

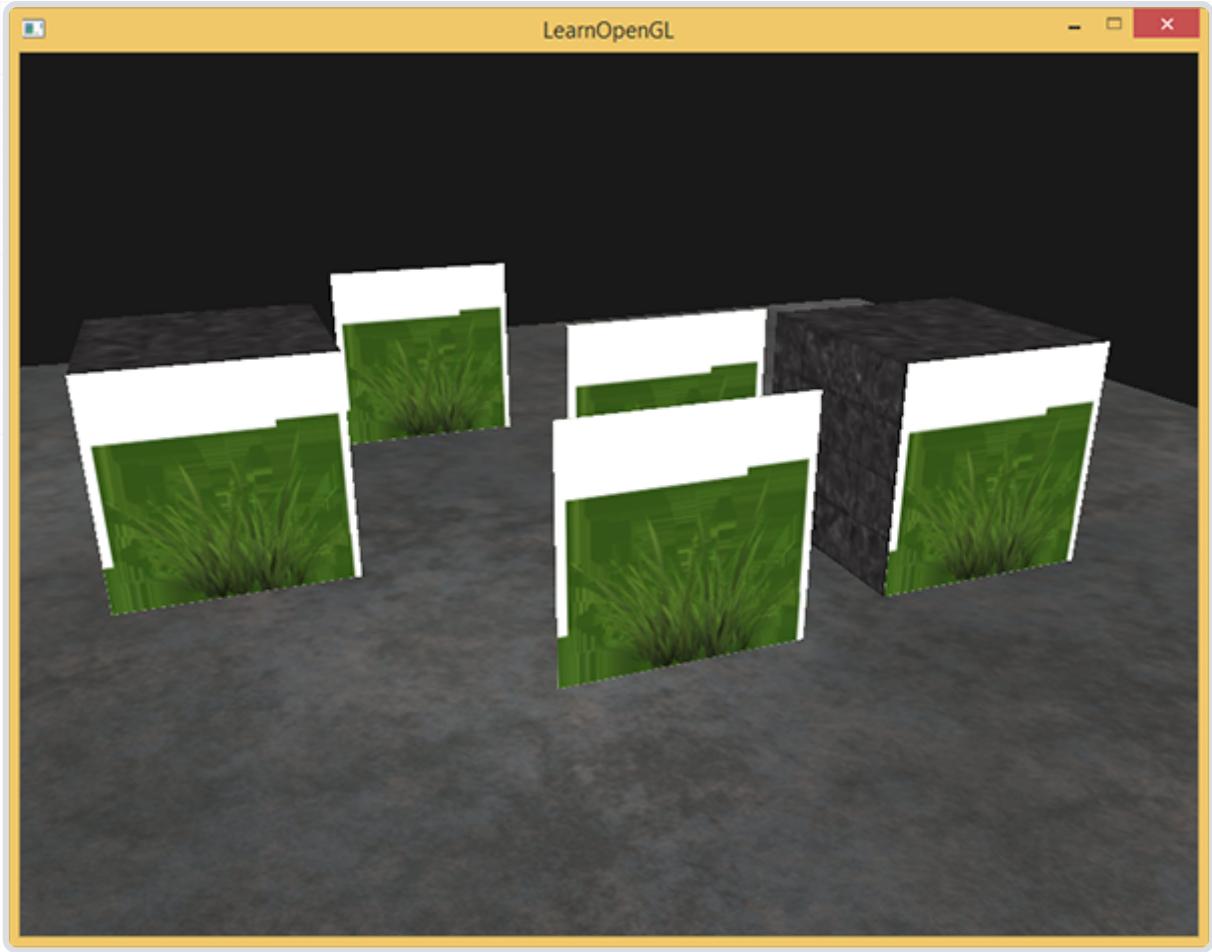
```
vector<glm::vec3> vegetation;
vegetation.push_back(glm::vec3(-1.5f, 0.0f, -0.48f));
vegetation.push_back(glm::vec3( 1.5f, 0.0f, 0.51f));
vegetation.push_back(glm::vec3( 0.0f, 0.0f, 0.7f));
vegetation.push_back(glm::vec3(-0.3f, 0.0f, -2.3f));
vegetation.push_back(glm::vec3( 0.5f, 0.0f, -0.6f));
```

每个草都被渲染到了一个四边形上，贴上草的纹理。这并不能完美地表示3D的草，但这比加载复杂的模型要快多了。使用一些小技巧，比如在同一个位置加入一些旋转后的草四边形，你仍然能获得比较好的结果的。

因为草的纹理是添加到四边形对象上的，我们还需要创建另外一个VAO，填充VBO，设置正确的顶点属性指针。接下来，在绘制完地板和两个立方体后，我们将会绘制草：

```
glBindVertexArray(vegetationVA0);
glBindTexture(GL_TEXTURE_2D, grassTexture);
for(unsigned int i = 0; i < vegetation.size(); i++)
{
    model = glm::mat4(1.0f);
    model = glm::translate(model, vegetation[i]);
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

运行程序你将看到：



出现这种情况是因为OpenGL默认是不知道怎么处理alpha值的，更不知道什么时候应该丢弃片段。我们需要自己手动来弄。幸运的是，有了着色器，这还是非常容易的。GLSL给了我们`discard`命令，一旦被调用，它就会保证片段不会被进一步处理，所以就不会进入颜色缓冲。有了这个指令，我们就能够在片段着色器中检测一个片段的alpha值是否低于某个阈值，如果是的话，则丢弃这个片段，就好像它不存在一样：

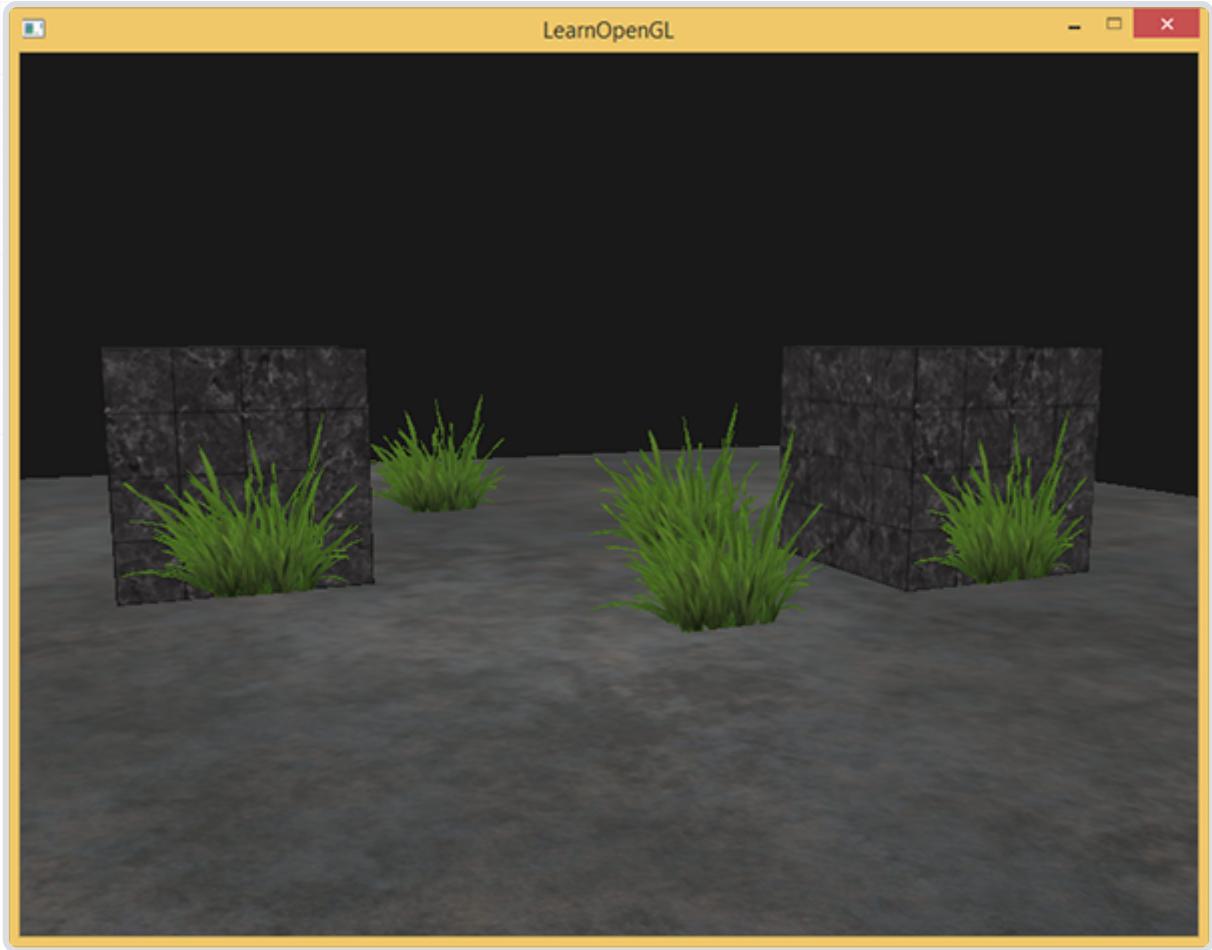
```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
}
```

这里，我们检测被采样的纹理颜色的alpha值是否低于0.1的阈值，如果是的话，则丢弃这个片段。片段着色器保证了它只会渲染不是（几乎）完全透明的片段。现在它看起来就正常了：



注意，当采样纹理的边缘的时候，OpenGL会对边缘的值和纹理下一个重复的值进行插值（因为我们将它的环绕方式设置为了`GL_REPEAT`。这通常是没问题的，但是由于我们使用了透明值，纹理图像的顶部将会与底部边缘的纯色值进行插值。这样的结果是一个半透明的有色边框，你可能会看见它环绕着你的纹理四边形。要想避免这个，每当你alpha纹理的时候，请将纹理的环绕方式设置为`GL_CLAMP_TO_EDGE`：

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

你可以在[这里](#)找到源码。

混合

虽然直接丢弃片段很好，但它不能让我们渲染半透明的图像。我们要么渲染一个片段，要么完全丢弃它。要想渲染有多个透明度级别的图像，我们需要启用**混合**(Blending)。和OpenGL大多数的功能一样，我们可以启用`GL_BLEND`来启用混合：

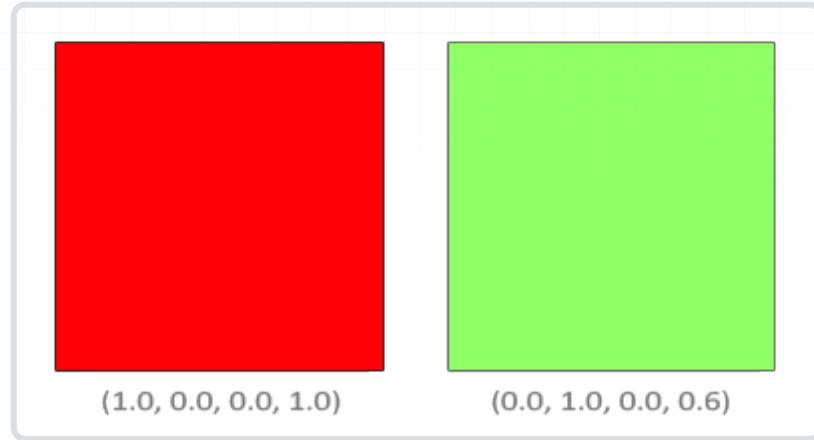
```
 glEnable(GL_BLEND);
```

启用了混合之后，我们需要告诉OpenGL它该如何混合。

OpenGL中的混合是通过下面这个方程来实现的：

- 源颜色向量。这是源自纹理的颜色向量。
- 目标颜色向量。这是当前储存在颜色缓冲中的颜色向量。
- 源因子值。指定了alpha值对源颜色的影响。
- 目标因子值。指定了alpha值对目标颜色的影响。

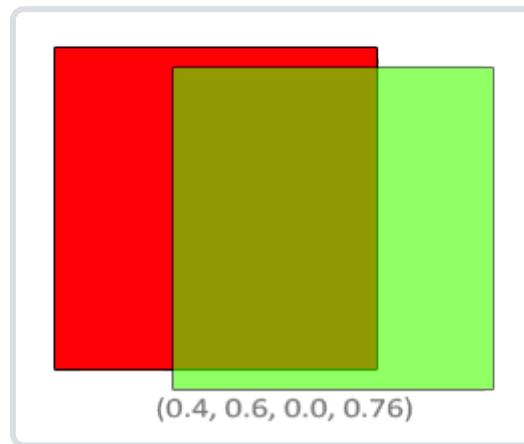
片段着色器运行完成后，并且所有的测试都通过之后，这个**混合方程**(Blend Equation)才会应用到片段颜色输出与当前颜色缓冲中的值(当前片段之前储存的之前片段的颜色)上。源颜色和目标颜色将会由OpenGL自动设定，但源因子和目标因子的值可以由我们来决定。我们先来看一个简单的例子：



我们有两个方形，我们希望将这个半透明的绿色方形绘制在红色方形之上。红色的方形将会是目标颜色（所以它应该先在颜色缓冲中），我们将要在这个红色方形之上绘制这个绿色方形。

问题来了：我们将因子值设置为什么？嘛，我们至少想让绿色方形乘以它的alpha值，所以我们想要将设置为源颜色向量的alpha值，也就是0.6。接下来就应该清楚了，目标方形的贡献应该为剩下的alpha值。如果绿色方形对最终颜色贡献了60%，那么红色方块应该对最终颜色贡献了40%，即 $1.0 - 0.6$ 。所以我们将设置为1减去源颜色向量的alpha值。这个方程变成了：

结果就是重叠方形的片段包含了一个60%绿色，40%红色的一种脏兮兮的颜色：



最终的颜色将会被储存到颜色缓冲中，替代之前的颜色。

这样子很不错，但我们该如何让OpenGL使用这样的因子呢？正好有一个专门的函数，叫做`glBlendFunc`。

`glBlendFunc(GLenum sfactor, GLenum dfactor)` 函数接受两个参数，来设置源和目标因子。OpenGL为我们定义了很多个选项，我们将在下面列出大部分最常用的选项。注意常数颜色向量可以通过`glBlendColor`函数来另外设置。

选项	值
<code>GL_ZERO</code>	因子等于
<code>GL_ONE</code>	因子等于
<code>GL_SRC_COLOR</code>	因子等于源颜色向量
<code>GL_ONE_MINUS_SRC_COLOR</code>	因子等于
<code>GL_DST_COLOR</code>	因子等于目标颜色向量
<code>GL_ONE_MINUS_DST_COLOR</code>	因子等于
<code>GL_SRC_ALPHA</code>	因子等于的分量
<code>GL_ONE_MINUS_SRC_ALPHA</code>	因子等于 的分量
<code>GL_DST_ALPHA</code>	因子等于的分量
<code>GL_ONE_MINUS_DST_ALPHA</code>	因子等于 的分量
<code>GL_CONSTANT_COLOR</code>	因子等于常数颜色向量
<code>GL_ONE_MINUS_CONSTANT_COLOR</code>	因子等于
<code>GL_CONSTANT_ALPHA</code>	因子等于的分量
<code>GL_ONE_MINUS_CONSTANT_ALPHA</code>	因子等于 的分量

为了获得之前两个方形的混合结果，我们需要使用源颜色向量的作为源因子，使用作为目标因子。这将会产生以下的`glBlendFunc`:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

也可以使用`glBlendFuncSeparate`为RGB和alpha通道分别设置不同的选项:

```
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

这个函数和我们之前设置的那样设置了RGB分量，但这样只能让最终的alpha分量被源颜色向量的alpha值所影响到。

OpenGL甚至给了我们更多的灵活性，允许我们改变方程中源和目标部分的运算符。当前源和目标是相加的，但如果愿意的话，我们也可以让它们相减。`glBlendEquation(GLenum mode)` 允许我们设置运算符，它提供了三个选项:

- `GL_FUNC_ADD`：默认选项，将两个分量相加：。
- `GL_FUNC_SUBTRACT`：将两个分量相减：。
- `GL_FUNC_REVERSE_SUBTRACT`：将两个分量相减，但顺序相反：。

通常我们都可以省略调用`glBlendEquation`，因为`GL_FUNC_ADD`对大部分的操作来说都是我们希望的混合方程，但如果你真的想打破主流，其它的方程也可能符合你的要求。

渲染半透明纹理

既然我们已经知道OpenGL是如何处理混合的了，是时候将我们的知识运用到实战中了，我们将会在场景中添加几个半透明的窗户。我们将使用本节开始的那个场景，但是这次不再是渲染草的纹理了，我们现在将使用本节开始时的那个[透明的窗户](#)纹理。

首先，在初始化时我们启用混合，并设定相应的混合函数:

```
 glEnable(GL_BLEND);
 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

由于启用了混合，我们就不需要丢弃片段了，所以我们把片段着色器还原：

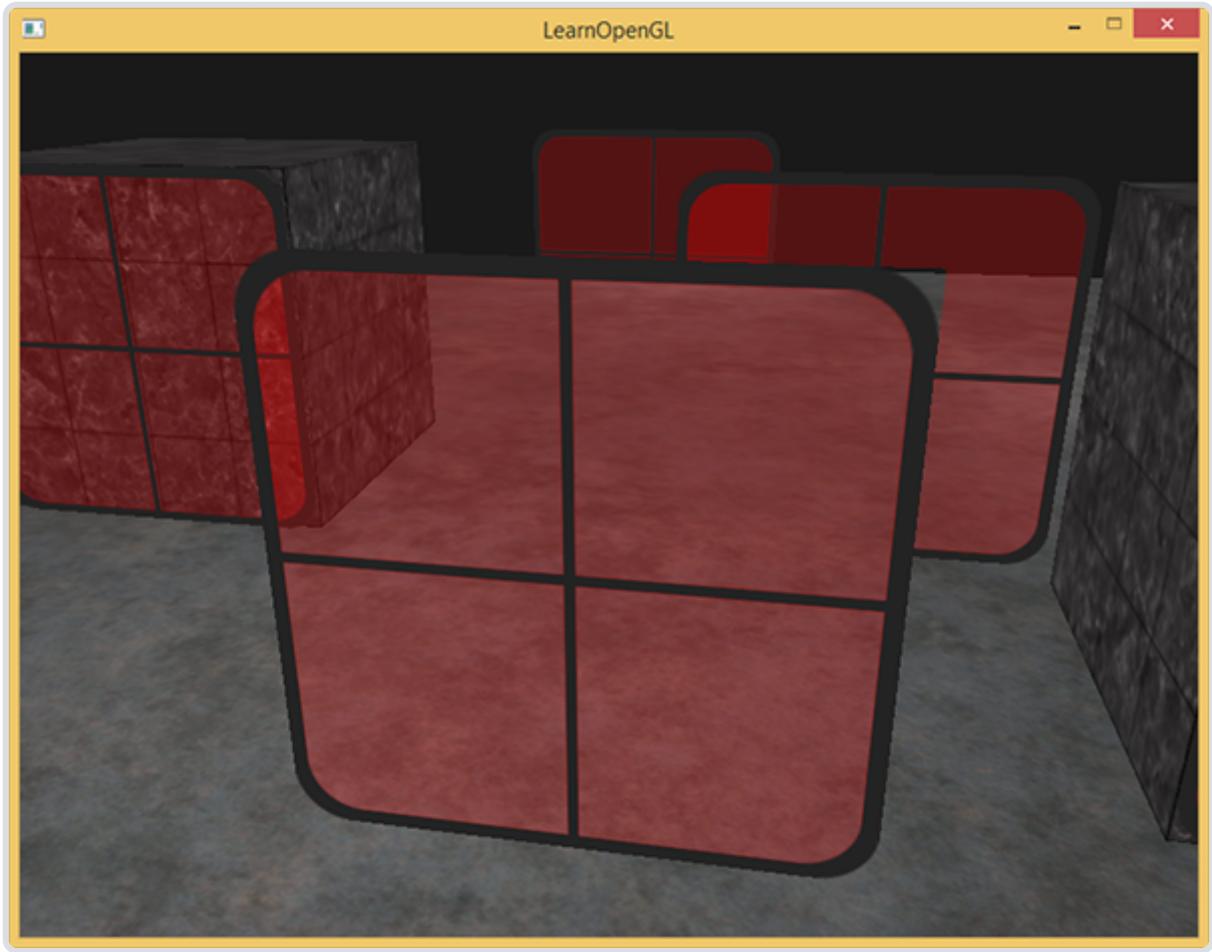
```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1, TexCoords);
}
```

现在（每当OpenGL渲染了一个片段时）它都会将当前片段的颜色和当前颜色缓冲中的片段颜色根据alpha值来进行混合。由于窗户纹理的玻璃部分是半透明的，我们应该能通窗户中看到背后的场景了。



如果你仔细看的话，你可能会注意到有些不对劲。最前面窗户的透明部分遮蔽了背后的窗户？这为什么会发生呢？

发生这一现象的原因是，深度测试和混合一起使用的话会产生一些麻烦。当写入深度缓冲时，深度缓冲不会检查片段是否是透明的，所以透明的部分会和其它值一样写入到深度缓冲中。结果就是窗户的整个四边形不论透明度都会进行深度测试。即使透明的部分应该显示背后的窗户，深度测试仍然丢弃了它们。

所以我们不能随意地决定如何渲染窗户，让深度缓冲解决所有的问题了。这也是混合变得有些麻烦的部分。要想保证窗户中能够显示它们背后的窗户，我们需要首先绘制背后的这部分窗户。这也就是说在绘制的时候，我们必须先手动将窗户按照最远到最近来排序，再按照顺序渲染。

注意，对于草这种全透明的物体，我们可以选择丢弃透明的片段而不是混合它们，这样就解决了这些头疼的问题（没有深度问题）。

不要打乱顺序

要想让混合在多个物体上工作，我们需要最先绘制最远的物体，最后绘制最近的物体。普通不需要混合的物体仍然可以使用深度缓冲正常绘制，所以它们不需要排序。但我们仍要保证它们在绘制（排序的）透明物体之前已经绘制完毕了。当绘制一个有不透明和透明物体的场景的时候，大体的原则如下：

1. 先绘制所有不透明的物体。
2. 对所有透明的物体排序。
3. 按顺序绘制所有透明的物体。

排序透明物体的一种方法是，从观察者视角获取物体的距离。这可以通过计算摄像机位置向量和物体的位置向量之间的距离所获得。接下来我们把距离和它对应的位置向量存储到一个STL库的map数据结构中。`map`会自动根据键值(Key)对它的值排序，所以只要我们添加了所有的位置，并以它的距离作为键，它们就会自动根据距离值排序了。

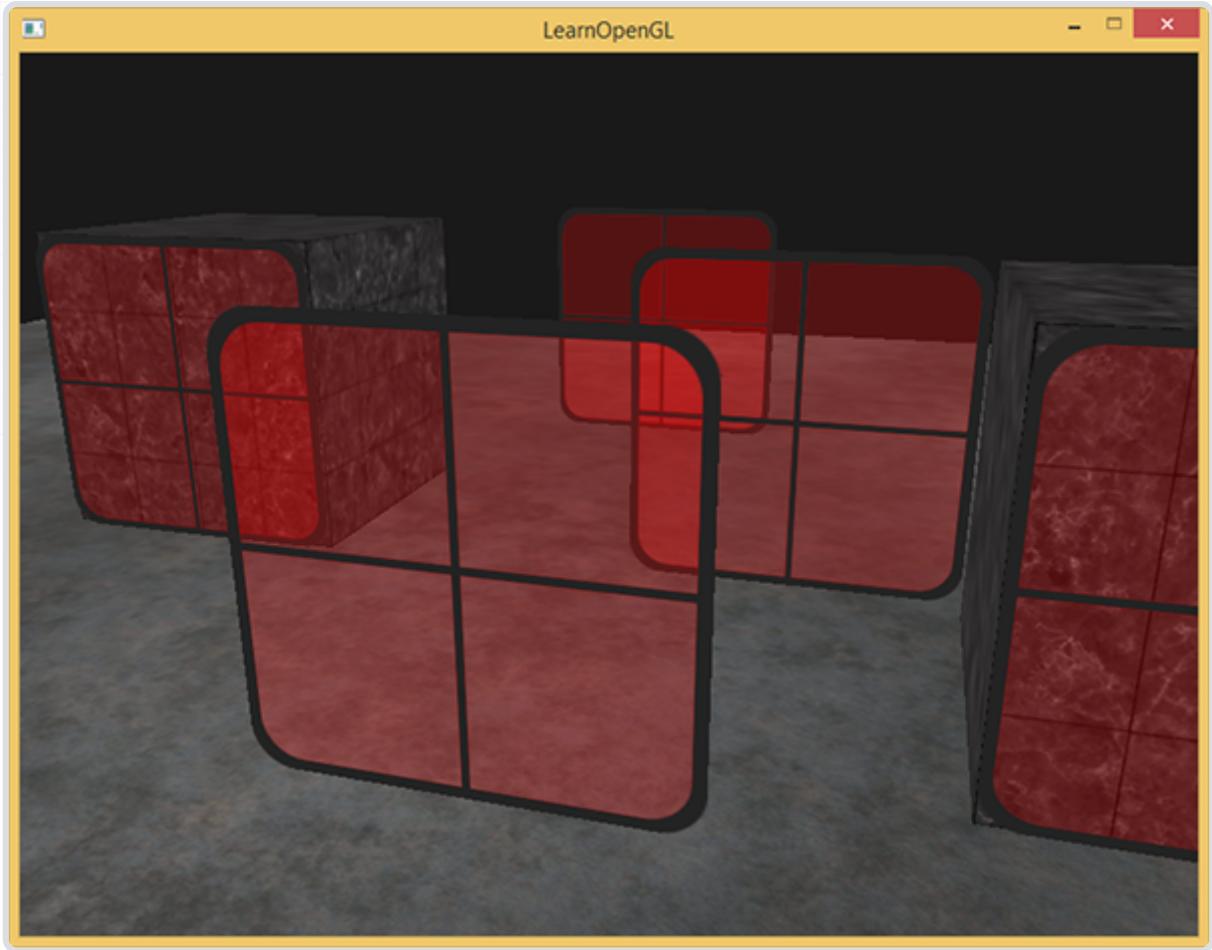
```
std::map<float, glm::vec3> sorted;
for (unsigned int i = 0; i < windows.size(); i++)
{
    float distance = glm::length(camera.Position - windows[i]);
    sorted[distance] = windows[i];
}
```

结果就是一个排序后的容器对象，它根据`distance`键值从低到高储存了每个窗户的位置。

之后，这次在渲染的时候，我们将以逆序（从远到近）从map中获取值，之后以正确的顺序绘制对应的窗户：

```
for(std::map<float,glm::vec3>::reverse_iterator it = sorted.rbegin(); it != sorted.rend(); ++it)
{
    model = glm::mat4();
    model = glm::translate(model, it->second);
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

我们使用了`map`的一个反向迭代器(Reverse Iterator)，反向遍历其中的条目，并将每个窗户四边形位移到对应的窗户位置上。这是排序透明物体的一个比较简单的实现，它能够修复之前的问题，现在场景看起来是这样的：



你可以在[这里](#)找到带有排序的完整源代码。

虽然按照距离排序物体这种方法对我们这个场景能够正常工作，但它并没有考虑旋转、缩放或者其它的变换，奇怪形状的物体需要一个不同的计量，而不是仅仅一个位置向量。

在场景中排序物体是一个很困难的技术，很大程度上由你场景的类型所决定，更别说它额外需要消耗的处理能力了。完整渲染一个包含不透明和透明物体的场景并不是那么容易。更高级的技术还有**次序无关透明度**(Order Independent Transparency, OIT)，但这超出本教程的范围了。现在，你还是要普通地混合你的物体，但如果你很小心，并且知道目前方法的限制的话，你仍然能够获得一个比较不错的混合实现。

2.5.5 面剔除

原文 [Face culling](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

尝试在脑子中想象一个3D立方体，数数你从任意方向最多能同时看到几个面。如果你的想象力不是过于丰富了，你应该能得出最大的面数是3。你可以从任意位置和任意方向看向这个球体，但你永远不能看到3个以上的面。所以我们为什么要浪费时间绘制我们不能看见的那3个面呢？如果我们能够以某种方式丢弃这几个看不见的面，我们能省下超过50%的片段着色器执行数！

我说的是超过50%而不是50%，因为从特定角度来看的话只能看见2个甚至是1个面。在这种情况下，我们就能省下超过50%了。

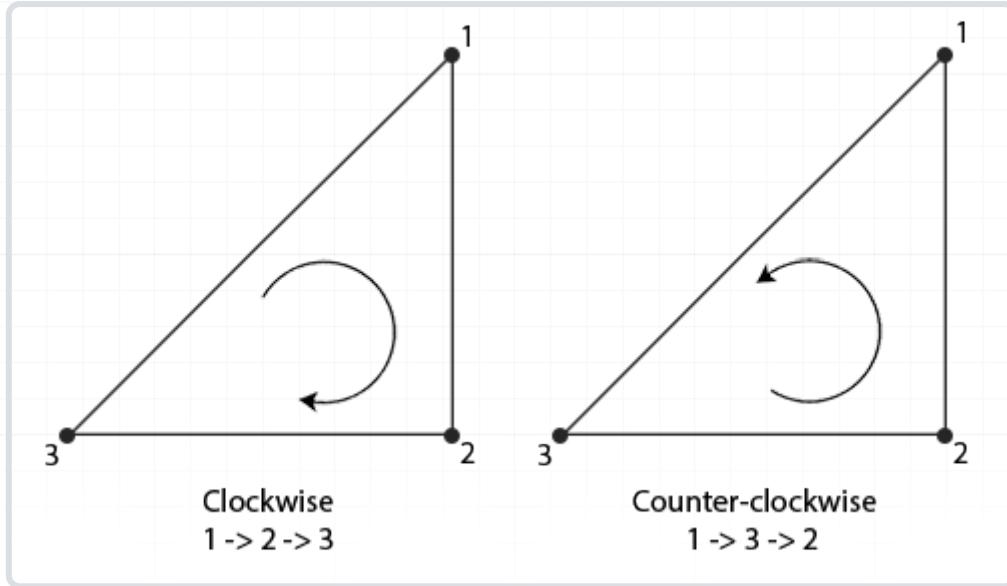
这是一个很好的主意，但我们仍有一个问题需要解决：我们如何知道一个物体的某一个面不能从观察者视角看到呢？

如果我们想象任何一个闭合形状，它的每一个面都有两侧，每一侧要么面向用户，要么背对用户。如果我们能够只绘制面向观察者的面呢？

这正是面剔除(Face Culling)所做的。OpenGL能够检查所有面向(Front Facing)观察者的面，并渲染它们，而丢弃那些背向(Back Facing)的面，节省我们很多的片段着色器调用（它们的开销很大！）。但我们仍要告诉OpenGL哪些面是正向面(Front Face)，哪些面是背向面(Back Face)。OpenGL使用了一个很聪明的技巧，分析顶点数据的环绕顺序(Winding Order)。

环绕顺序

当我们定义一组三角形顶点时，我们会以特定的环绕顺序来定义它们，可能是顺时针(Clockwise)的，也可能是逆时针(Counter-clockwise)的。每个三角形由3个顶点所组成，我们会从三角形中间来看，为这3个顶点设定一个环绕顺序。



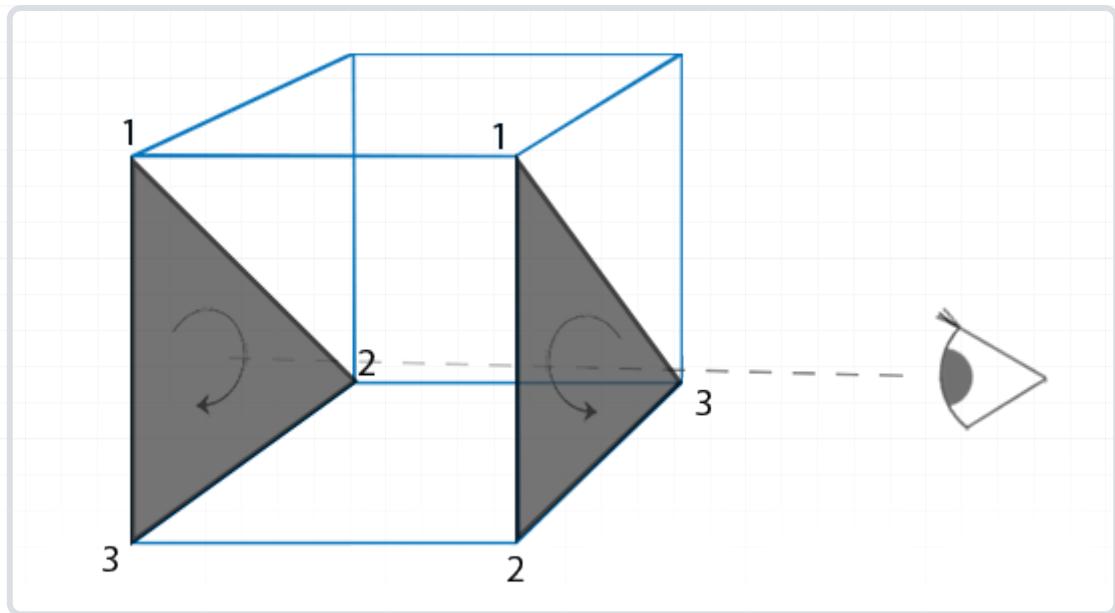
可以看到，我们首先定义了顶点1，之后我们可以选择定义顶点2或者顶点3，这个选择将定义了这个三角形的环绕顺序。下面的代码展示了这点：

```
float vertices[] = {
    // 顺时针
    vertices[0], // 顶点1
    vertices[1], // 顶点2
    vertices[2], // 顶点3
    // 逆时针
    vertices[0], // 顶点1
    vertices[2], // 顶点3
    vertices[1] // 顶点2
};
```

每组组成三角形图元的三个顶点就包含了一个环绕顺序。OpenGL在渲染图元的时候将使用这个信息来决定一个三角形是一个[正向](#)三角形还是[背向](#)三角形。默认情况下，逆时针顶点所定义的三角形将会被处理为正向三角形。

当你定义顶点顺序的时候，你应该想象对应的三角形是面向你的，所以你定义的三角形从正面看去应该是逆时针的。这样定义顶点很棒的一点是，实际的环绕顺序是在光栅化阶段进行的，也就是顶点着色器运行之后。这些顶点就是从观察者视角所见的了。

观察者所面向的所有三角形顶点就是我们所指定的正确环绕顺序了，而立方体另一面的三角形顶点则是以相反的环绕顺序所渲染的。这样的结果就是，我们所面向的三角形将会是正向三角形，而背面的三角形则是背向三角形。下面这张图显示了这个效果：



在顶点数据中，我们将两个三角形都以逆时针顺序定义（正面的三角形是1、2、3，背面的三角形也是1、2、3（如果我们从正面看这个三角形的话））。然而，如果从观察者当前视角使用1、2、3的顺序来绘制的话，从观察者的方向来看，背面的三角形将会是以顺时针顺序渲染的。虽然背面的三角形是以逆时针定义的，它现在是以顺时针顺序渲染的了。这正是我们想要**剔除**（Cull，丢弃）的不可见面！

在顶点数据中，我们定义的是两个逆时针顺序的三角形。然而，从观察者的方面看，后面的三角形是顺时针的，如果我们仍以1、2、3的顺序以观察者当面的视野看的话。即使我们以逆时针顺序定义后面的三角形，它现在还是变为顺时针。它正是我们打算剔除（丢弃）的不可见的面！

面剔除

在本节的开头我们就说过，OpenGL能够丢弃那些渲染为背向三角形的三角形图元。既然已经知道如何设置顶点的环绕顺序了，我们就可以使用OpenGL的**面剔除**选项了，它默认是禁用状态的。

在之前教程中使用的立方体顶点数据并不是按照逆时针环绕顺序定义的，所以我更新了顶点数据，来反映逆时针的环绕顺序，你可以从[这里](#)复制它们。尝试想象这些顶点，确认在每个三角形中它们都是以逆时针定义的，这是一个很好的习惯。

要想启用面剔除，我们只需要启用OpenGL的`GL_CULL_FACE`选项：

```
glEnable(GL_CULL_FACE);
```

从这一句代码之后，所有背向面都将被丢弃（尝试飞进立方体内部，看看所有的内面是不是都被丢弃了）。目前我们在渲染片段的时候能够节省50%以上的性能，但注意这只对像立方体这样的封闭形状有效。当我们想要绘制[上一节](#)中的草时，我们必须要再次禁用面剔除，因为它们的正向面和背向面都应该是可见的。

OpenGL允许我们改变需要剔除的面的类型。如果我们只想剔除正向面而不是背向面会怎么样？我们可以调用`glCullFace`来定义这一行为：

```
glCullFace(GL_FRONT);
```

`glCullFace`函数有三个可用的选项：

- `GL_BACK`：只剔除背向面。
- `GL_FRONT`：只剔除正向面。
- `GL_FRONT_AND_BACK`：剔除正向面和背向面。

`glCullFace`的初始值是`GL_BACK`。除了需要剔除的面之外，我们也可以通过调用`glFrontFace`，告诉OpenGL我们希望将顺时针的面（而不是逆时针的面）定义为正向面：

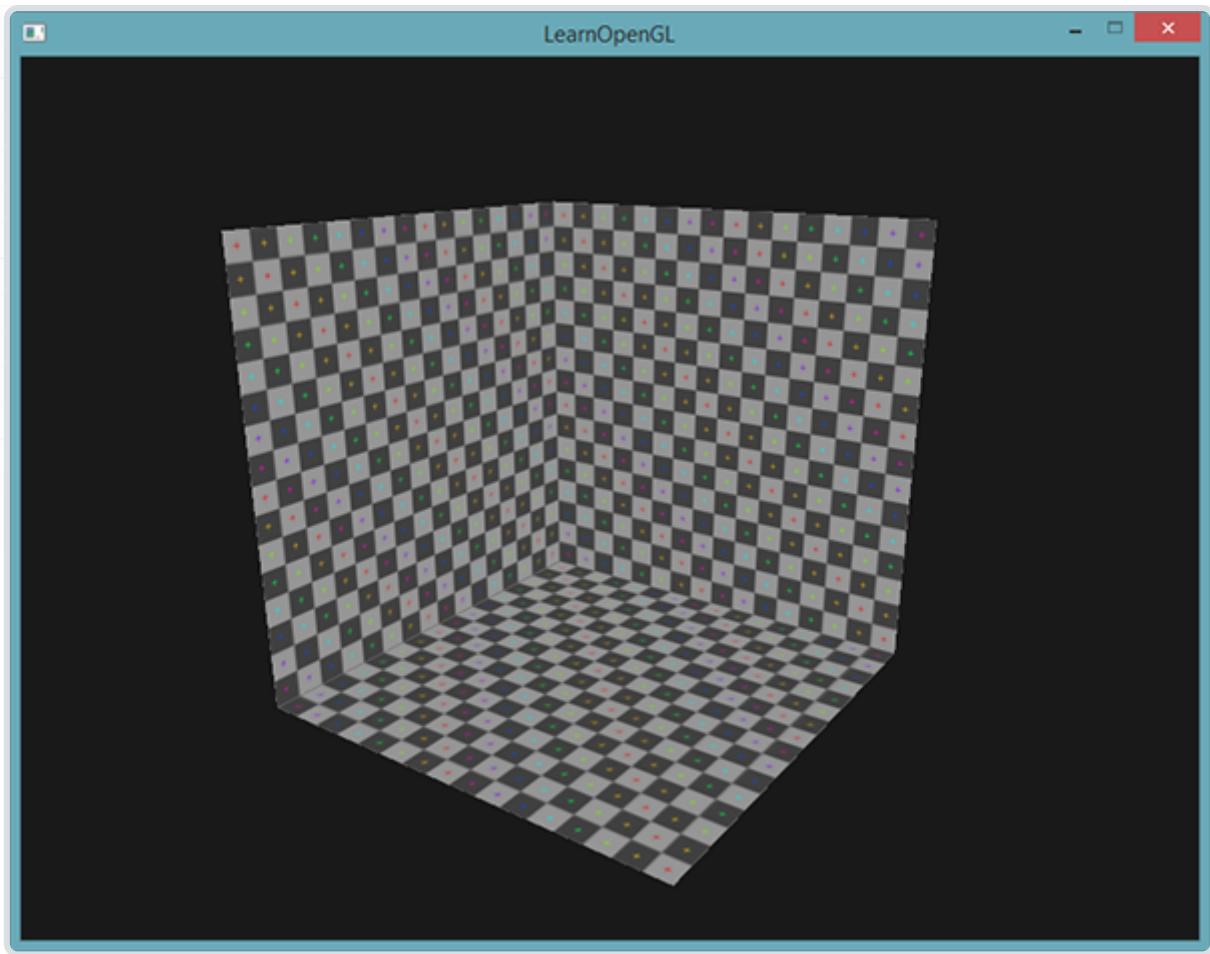
```
glFrontFace(GL_CCW);
```

默认值是`GL_CCW`，它代表的是逆时针的环绕顺序，另一个选项是`GL_CW`，它（显然）代表的是顺时针顺序。

我们可以来做一个实验，告诉OpenGL现在顺时针顺序代表的是正向面：

```
 glEnable(GL_CULL_FACE);
 glCullFace(GL_BACK);
 glFrontFace(GL_CW);
```

这样的结果是只有背向面被渲染了：



注意你可以仍使用默认的逆时针环绕顺序，但剔除正向面，来达到相同的效果：

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

可以看到，面剔除是一个提高OpenGL程序性能的很棒的工具。但你需要记住哪些物体能够从面剔除中获益，而哪些物体不应该被剔除。

练习

- 你能够重新定义顶点数据，将每个三角形设置为顺时针顺序，并将顺时针的三角形设置为正向面，仍将场景渲染出来吗？：[参考解答](#)

2.5.6 帧缓冲

原文 [Framebuffers](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

到目前为止，我们已经使用了很多屏幕缓冲了：用于写入颜色值的颜色缓冲、用于写入深度信息的深度缓冲和允许我们根据一些条件丢弃特定片段的模板缓冲。这些缓冲结合起来叫做帧缓冲(Framebuffer)，它被储存在内存中。OpenGL允许我们定义我们自己的帧缓冲，也就是说我们能够定义我们自己的颜色缓冲，甚至是深度缓冲和模板缓冲。

我们目前所做的所有操作都是在默认帧缓冲的渲染缓冲上进行的。默认的帧缓冲是在你创建窗口的时候生成和配置的（GLFW帮我们做了这些）。有了我们自己的帧缓冲，我们就能够有更多方式来渲染了。

你可能不能很快理解帧缓冲的应用，但渲染你的场景到不同的帧缓冲能够让我们在场景中加入类似镜子的东西，或者做出很酷的后期处理效果。首先我们会讨论它是如何工作的，之后我们将实现这些炫酷的后期处理效果。

创建一个帧缓冲

和OpenGL中的其它对象一样，我们会使用一个叫做`glGenFramebuffers`的函数来创建一个帧缓冲对象(Framebuffer Object, FBO)：

```
unsigned int fbo;
 glGenFramebuffers(1, &fbo);
```

这种创建和使用对象的方式我们已经见过很多次了，所以它的使用函数也和其他的对象类似。首先我们创建一个帧缓冲对象，将它绑定为激活的(Active)帧缓冲，做一些操作，之后解绑帧缓冲。我们使用`glBindFramebuffer`来绑定帧缓冲。

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

在绑定到`GL_FRAMEBUFFER`目标之后，所有的读取和写入帧缓冲的操作将会影响当前绑定的帧缓冲。我们也可以使用`GL_READ_FRAMEBUFFER`或`GL_DRAW_FRAMEBUFFER`，将一个帧缓冲分别绑定到读取目标或写入目标。绑定到`GL_READ_FRAMEBUFFER`的帧缓冲将会使用在所有像是`glReadPixels`的读取操作中，而绑定到`GL_DRAW_FRAMEBUFFER`的帧缓冲将会被用作渲染、清除等写入操作的目标。大部分情况你都不需要区分它们，通常都会使用`GL_FRAMEBUFFER`，绑定到两个上。

不幸的是，我们现在还不能使用我们的帧缓冲，因为它还不完整(Complete)，一个完整的帧缓冲需要满足以下的条件：

- 附加至少一个缓冲（颜色、深度或模板缓冲）。
- 至少有一个颜色附件(Attachment)。
- 所有的附件都必须是完整的（保留了内存）。
- 每个缓冲都应该有相同的样本数。

如果你不知道什么是样本，不要担心，我们将在之后的教程中讲到。

从上面的条件中可以知道，我们需要为帧缓冲创建一些附件，并将附件附加到帧缓冲上。在完成所有的条件之后，我们可以以`GL_FRAMEBUFFER`为参数调用`glCheckFramebufferStatus`，检查帧缓冲是否完整。它将会检测当前绑定的帧缓冲，并返回规范中这些值的其中之一。如果它返回的是`GL_FRAMEBUFFER_COMPLETE`，帧缓冲就是完整的了。

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) == GL_FRAMEBUFFER_COMPLETE)
 // 执行胜利的舞蹈
```

之后所有的渲染操作将会渲染到当前绑定帧缓冲的附件中。由于我们的帧缓冲不是默认帧缓冲，渲染指令将不会对窗口的视觉输出有任何影响。出于这个原因，渲染到一个不同的帧缓冲被叫做离屏渲染(Off-screen Rendering)。要保证所有的渲染操作在主窗口中有视觉效果，我们需要再次激活默认帧缓冲，将它绑定到 0。

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

在完成所有的帧缓冲操作之后，不要忘记删除这个帧缓冲对象：

```
glDeleteFramebuffers(1, &fbo);
```

在完整性检查执行之前，我们需要给帧缓冲附加一个附件。附件是一个内存位置，它能够作为帧缓冲的一个缓冲，可以将它想象为一个图像。当创建一个附件的时候我们有两个选项：纹理或渲染缓冲对象(Renderbuffer Object)。

纹理附件

当把一个纹理附加到帧缓冲的时候，所有的渲染指令将会写入到这个纹理中，就像它是一个普通的颜色/深度或模板缓冲一样。使用纹理的优点是，所有渲染操作的结果将会被储存在一个纹理图像中，我们之后可以在着色器中很方便地使用它。

为帧缓冲创建一个纹理和创建一个普通的纹理差不多：

```
unsigned int texture;
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);

 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);

 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

主要的区别就是，我们将维度设置为了屏幕大小（尽管这不是必须的），并且我们给纹理的 `data` 参数传递了 `NULL`。对于这个纹理，我们仅仅分配了内存而没有填充它。填充这个纹理将会在我们渲染到帧缓冲之后来进行。同样注意我们并不关心环绕方式或多级渐远纹理，我们在大多数情况下都不会需要它们。

如果你想将你的屏幕渲染到一个更小或更大的纹理上，你需要（在渲染到你的帧缓冲之前）再次调用 `glViewport`，使用纹理的新维度作为参数，否则只有一小部分的纹理或屏幕会被渲染到这个纹理上。

现在我们已经创建好一个纹理了，要做的最后一件事就是将它附加到帧缓冲上了：

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

`glFramebufferTexture2D` 有以下的参数：

- `target`：帧缓冲的目标（绘制、读取或者两者皆有）
- `attachment`：我们想要附加的附件类型。当前我们正在附加一个颜色附件。注意最后的 0 意味着我们可以附加多个颜色附件。我们将在之后的教程中提到。
- `textarget`：你希望附加的纹理类型
- `texture`：要附加的纹理本身
- `level`：多级渐远纹理的级别。我们将它保留为0。

除了颜色附件之外，我们还可以附加一个深度和模板缓冲纹理到帧缓冲对象中。要附加深度缓冲的话，我们将附件类型设置为 `GL_DEPTH_ATTACHMENT`。注意纹理的格式(Format)和内部格式(Internalformat)类型将变为 `GL_DEPTH_COMPONENT`，来反映深度缓冲的储存格式。要附加模板缓冲的话，你要将第二个参数设置为 `GL_STENCIL_ATTACHMENT`，并将纹理的格式设定为 `GL_STENCIL_INDEX`。

也可以将深度缓冲和模板缓冲附加为一个单独的纹理。纹理的每32位数值将包含24位的深度信息和8位的模板信息。要将深度和模板缓冲附加为一个纹理的话，我们使用 `GL_DEPTH_STENCIL_ATTACHMENT` 类型，并配置纹理的格式，让它包含合并的深度和模板值。将一个深度和模板缓冲附加为一个纹理到帧缓冲的例子可以在下面找到：

```
glTexImage2D(
    GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, 800, 600, 0,
    GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL
);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_TEXTURE_2D, texture, 0);
```

渲染缓冲对象附件

渲染缓冲对象(Renderbuffer Object)是在纹理之后引入到OpenGL中，作为一个可用的帧缓冲附件类型的，所以在过去纹理是唯一可用的附件。和纹理图像一样，渲染缓冲对象是一个真正的缓冲，即一系列的字节、整数、像素等。渲染缓冲对象附加的好处是，它会将数据储存为OpenGL原生的渲染格式，它是为离屏渲染到帧缓冲优化过的。

渲染缓冲对象直接将所有的渲染数据储存到它的缓冲中，不会做任何针对纹理格式的转换，让它变为一个更快的可写储存介质。然而，渲染缓冲对象通常都是只写的，所以你不能读取它们（比如使用纹理访问）。当然你仍然还是能够使用 `glReadPixels` 来读取它，这会从当前绑定的帧缓冲，而不是附件本身，中返回特定区域的像素。

因为它的数据已经是原生的格式了，当写入或者复制它的数据到其它缓冲中时是非常快的。所以，交换缓冲这样的操作在使用渲染缓冲对象时会非常快。我们在每个渲染迭代最后使用的 `glfwSwapBuffers`，也可以通过渲染缓冲对象实现：只需要写入一个渲染缓冲图像，并在最后交换到另外一个渲染缓冲就可以了。渲染缓冲对象对这种操作非常完美。

创建一个渲染缓冲对象的代码和帧缓冲的代码很类似：

```
unsigned int rbo;
 glGenRenderbuffers(1, &rbo);
```

类似，我们需要绑定这个渲染缓冲对象，让之后所有的渲染缓冲操作影响当前的 `rbo`：

```
 glBindRenderbuffer(GL_RENDERBUFFER, rbo);
```

由于渲染缓冲对象通常都是只写的，它们会经常用于深度和模板附件，因为大部分时间我们都不需要从深度和模板缓冲中读取值，只关心深度和模板测试。我们需要深度和模板值用于测试，但不需要对它们进行采样，所以渲染缓冲对象非常适合它们。当我们不需要从这些缓冲中采样的时候，通常都会选择渲染缓冲对象，因为它会更优化一点。

创建一个深度和模板渲染缓冲对象可以通过调用 `glRenderbufferStorage` 函数来完成：

```
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
```

创建一个渲染缓冲对象和纹理对象类似，不同的是这个对象是专门被设计作为帧缓冲附件使用的，而不是纹理那样的通用数据缓冲(General Purpose Data Buffer)。这里我们选择 `GL_DEPTH24_STENCIL8` 作为内部格式，它封装了24位的深度和8位的模板缓冲。

最后一件事就是附加这个渲染缓冲对象：

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

渲染缓冲对象能为你的帧缓冲对象提供一些优化，但知道什么时候使用渲染缓冲对象，什么时候使用纹理是很重要的。通常的规则是，如果你不需要从一个缓冲中采样数据，那么对这个缓冲使用渲染缓冲对象会是明智的选择。如果你需要从缓冲中采样颜色或深度值等数据，那么你应该选择纹理附件。性能方面它不会产生非常大的影响的。

渲染到纹理

既然我们已经知道帧缓冲（大概）是怎么工作的了，是时候实践它们了。我们将会将场景渲染到一个附加到帧缓冲对象上的颜色纹理中，之后将在一个横跨整个屏幕的四边形上绘制这个纹理。这样视觉输出和没使用帧缓冲时是完全一样的，但这次是打印到了一个四边形上。这为什么很有用呢？我们会在下一部分中知道原因。

首先要创建一个帧缓冲对象，并绑定它，这些都很直观：

```
unsigned int framebuffer;
 glGenFramebuffers(1, &framebuffer);
 glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

接下来我们需要创建一个纹理图像，我们将它作为一个颜色附件附加到帧缓冲上。我们将纹理的维度设置为窗口的宽度和高度，并且不初始化它的数据：

```
// 生成纹理
unsigned int texColorBuffer;
 glGenTextures(1, &texColorBuffer);
 glBindTexture(GL_TEXTURE_2D, texColorBuffer);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
 glBindTexture(GL_TEXTURE_2D, 0);

// 将它附加到当前绑定的帧缓冲对象
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texColorBuffer, 0);
```

我们还希望OpenGL能够进行深度测试（如果你需要的话还有模板测试），所以我们还需要添加一个深度（和模板）附件到帧缓冲中。由于我们只希望采样颜色缓冲，而不是其它的缓冲，我们可以为它们创建一个渲染缓冲对象。还记得当我们不需要采样缓冲的时候，渲染缓冲对象是更好的选择吗？

创建一个渲染缓冲对象不是非常复杂。我们需要记住的唯一事情是，我们将它创建为一个深度和模板附件渲染缓冲对象。我们将它的内部格式设置为`GL_DEPTH24_STENCIL8`，对我们来说这个精度已经足够了。

```
unsigned int rbo;
 glGenRenderbuffers(1, &rbo);
 glBindRenderbuffer(GL_RENDERBUFFER, rbo);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
 glBindRenderbuffer(GL_RENDERBUFFER, 0);
```

当我们为渲染缓冲对象分配了足够的内存之后，我们可以解绑这个渲染缓冲。

接下来，作为完成帧缓冲之前的最后一步，我们将渲染缓冲对象附加到帧缓冲的深度和模板附件上：

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

最后，我们希望检查帧缓冲是否是完整的，如果不是，我们将打印错误信息。

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

记得要解绑帧缓冲，保证我们不会不小心渲染到错误的帧缓冲上。

现在这个帧缓冲就完整了，我们只需要绑定这个帧缓冲对象，让渲染到帧缓冲的缓冲中而不是默认的帧缓冲中。之后的渲染指令将会影响当前绑定的帧缓冲。所有的深度和模板操作都会从当前绑定的帧缓冲的深度和模板附件中（如果有的话）读取。如果你忽略了深度缓冲，那么所有的深度测试操作将不再工作，因为当前绑定的帧缓冲中不存在深度缓冲。

所以，要想绘制场景到一个纹理上，我们需要采取以下的步骤：

1. 将新的帧缓冲绑定为激活的帧缓冲，和往常一样渲染场景
2. 绑定默认的帧缓冲
3. 绘制一个横跨整个屏幕的四边形，将帧缓冲的颜色缓冲作为它的纹理。

我们将会绘制深度测试小节中的场景，但这次使用的是旧的箱子纹理。

为了绘制这个四边形，我们将会新创建一套简单的着色器。我们将不会包含任何花哨的矩阵变换，因为我们提供的是标准化设备坐标的顶点坐标，所以我们可以直接将它们设定为顶点着色器的输出。顶点着色器是这样的：

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    TexCoords = aTexCoords;
}
```

并没有太复杂的东西。片段着色器会更加基础，我们做的唯一一件事就是从纹理中采样：

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D screenTexture;

void main()
{
    FragColor = texture(screenTexture, TexCoords);
```

接着就靠你来为屏幕四边形创建并配置一个VAO了。帧缓冲的一个渲染迭代将会有以下的结构：

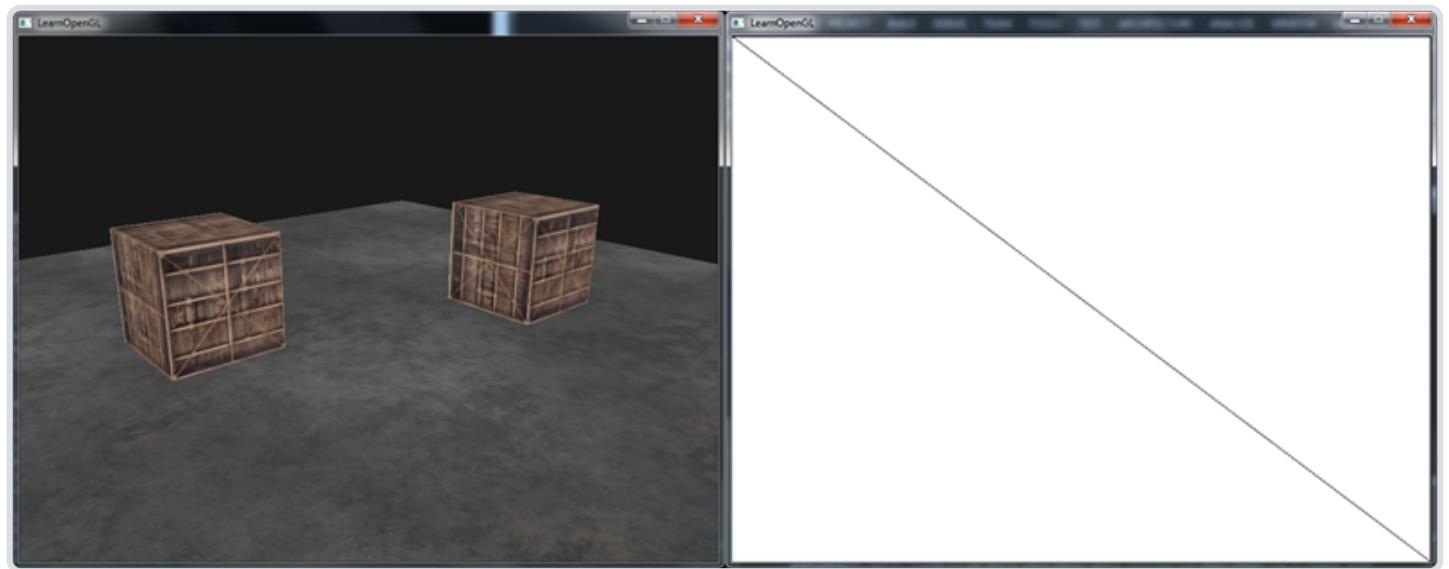
```
// 第一处理阶段(Pass)
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 我们现在不使用模板缓冲
 glEnable(GL_DEPTH_TEST);
DrawScene();

// 第二处理阶段
glBindFramebuffer(GL_FRAMEBUFFER, 0); // 返回默认
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

screenShader.use();
 glBindVertexArray(quadVAO);
 glDisable(GL_DEPTH_TEST);
 glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
 glDrawArrays(GL_TRIANGLES, 0, 6);
```

要注意一些事情。第一，由于我们使用的每个帧缓冲都有它自己一套缓冲，我们希望设置合适的位，调用`glClear`，清除这些缓冲。第二，当绘制四边形时，我们将禁用深度测试，因为我们是在绘制一个简单的四边形，并不需要关系深度测试。在绘制普通场景的时候我们将会重新启用深度测试。

有很多步骤都可能会出错，所以如果你没有得到输出的话，尝试调试程序，并重新阅读本节的相关部分。如果所有的东西都能够正常工作，你将会得到下面这样的视觉输出：



左边展示的是视觉输出，它和[深度测试](#)中是完全一样的，但这次是渲染在一个简单的四边形上。如果我们使用线框模式渲染场景，就会变得很明显，我们在默认的帧缓冲中只绘制了一个简单的四边形。

你可以在[这里](#)找到程序的源代码。

所以这个有什么用处呢？因为我们能够以一个纹理图像的方式访问已渲染场景中的每个像素，我们可以在片段着色器中创建出非常有趣的效果。这些有趣效果统称为[后期处理](#)(Post-processing)效果。

2.5.7 后期处理

既然整个场景都被渲染到了一个纹理上，我们可以简单地通过修改纹理数据创建出一些非常有意思的效果。在这一部分中，我们将会向你展示一些流行的后期处理效果，并告诉你该如何使用创造力创建你自己的效果。

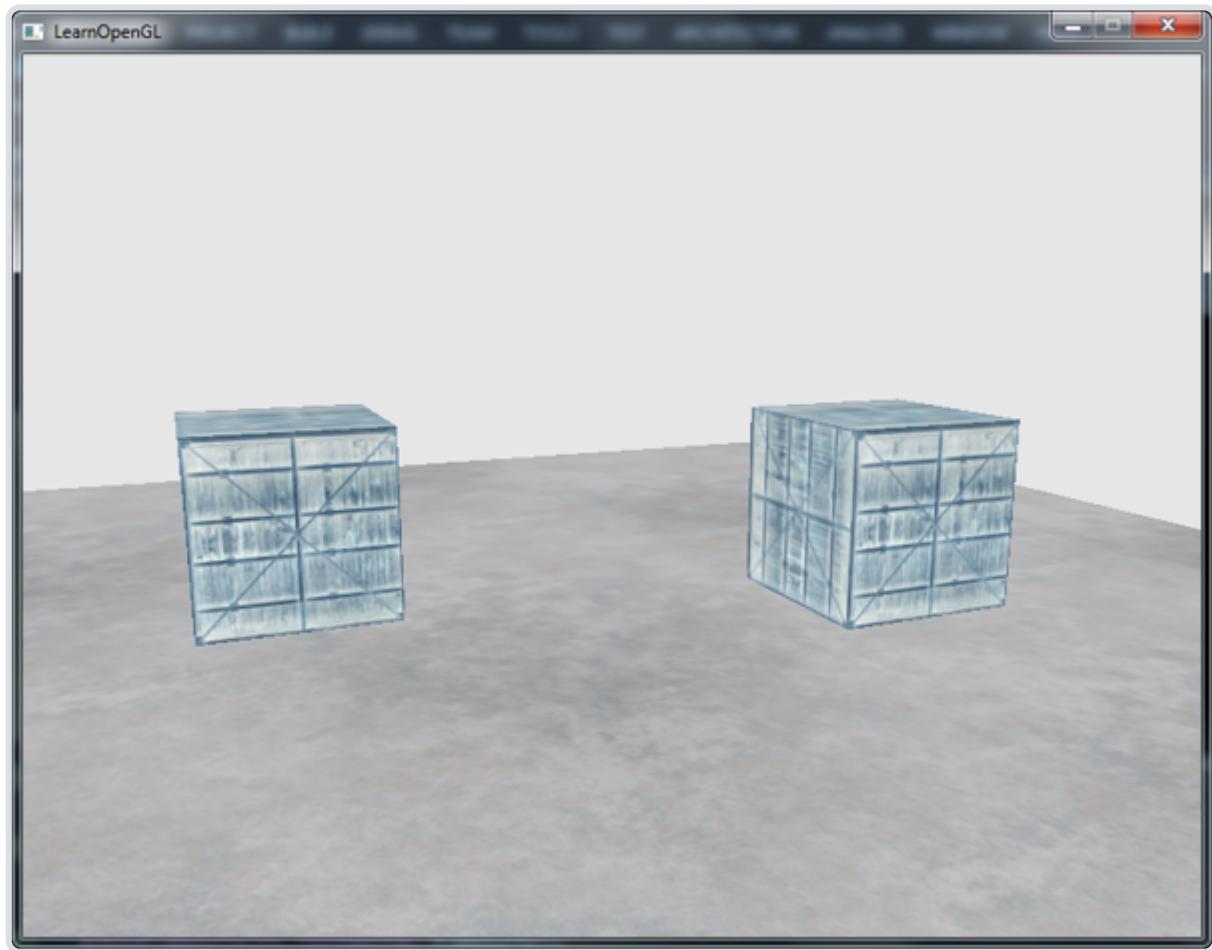
让我们先从最简单的后期处理效果开始。

反相

我们现在能够访问渲染输出的每个颜色，所以在（译注：屏幕的）片段着色器中返回这些颜色的反相(Inversion)并不是很难。我们将会从屏幕纹理中取颜色值，然后用1.0减去它，对它进行反相：

```
void main()
{
    FragColor = vec4(1.0 - texture(screenTexture, TexCoords), 1.0);
}
```

尽管反相是一个相对简单的后期处理效果，它已经能创造一些奇怪的效果了：



在片段着色器中仅仅使用一行代码，就能让整个场景的颜色都反相了。很酷吧？

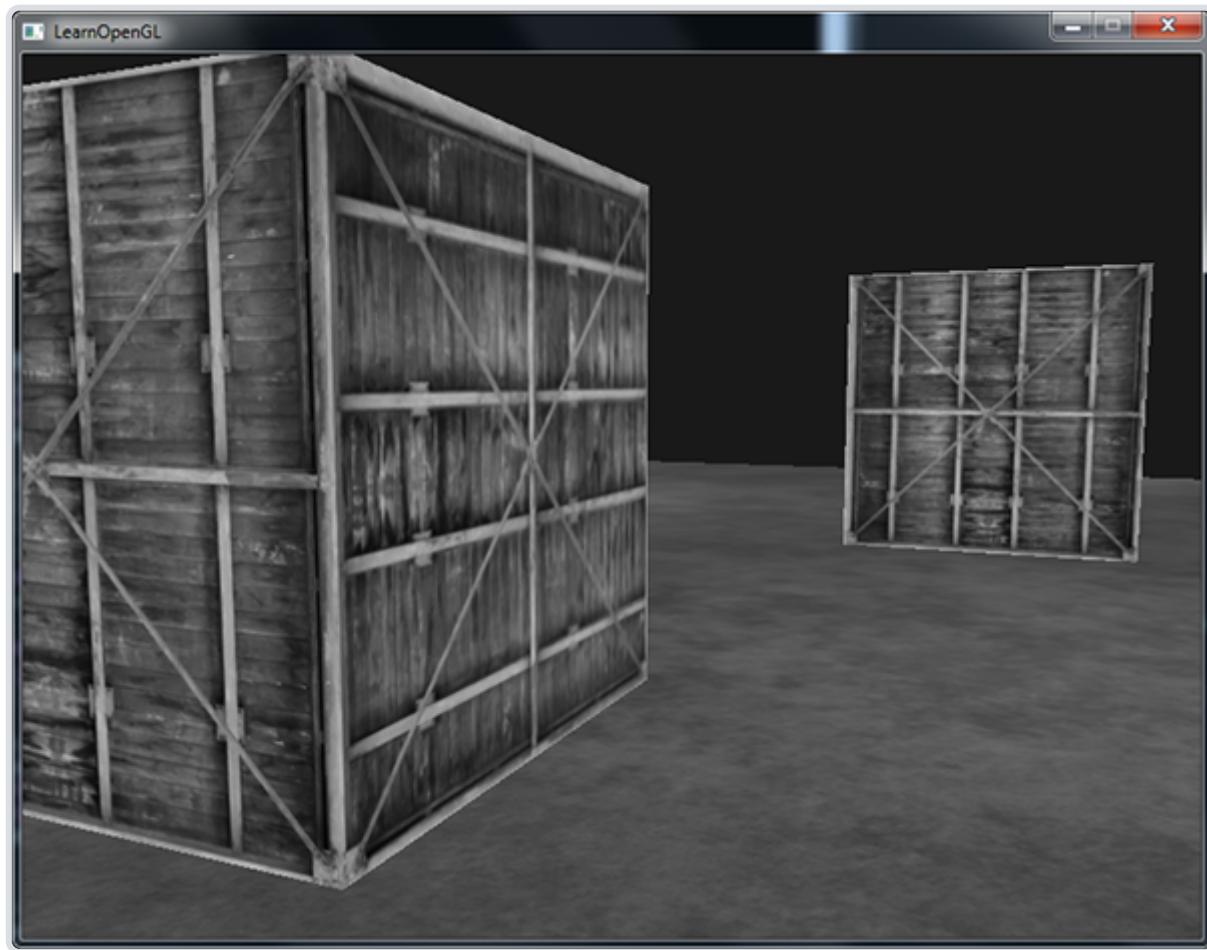
灰度

另外一个很有趣的效果是，移除场景中除了黑白灰以外所有的颜色，让整个图像灰度化(Grayscale)。很简单的实现方式是，取所有的颜色分量，将它们平均化：

```
void main()
{
    FragColor = texture(screenTexture, TexCoords);
    float average = (FragColor.r + FragColor.g + FragColor.b) / 3.0;
    FragColor = vec4(average, average, average, 1.0);
}
```

这已经能创造很好的结果了，但人眼会对绿色更加敏感一些，而对蓝色不那么敏感，所以为了获取物理上更精确的效果，我们需要使用加权的(Weighted)通道：

```
void main()
{
    FragColor = texture(screenTexture, TexCoords);
    float average = 0.2126 * FragColor.r + 0.7152 * FragColor.g + 0.0722 * FragColor.b;
    FragColor = vec4(average, average, average, 1.0);
}
```



你可能不会立刻发现有什么差别，但在更复杂的场景中，这样的加权灰度效果会更真实一点。

核效果

在一个纹理图像上做后期处理的另外一个好处是，我们可以从纹理的其它地方采样颜色值。比如说我们可以在当前纹理坐标的周围取一小块区域，对当前纹理值周围的多个纹理值进行采样。我们可以结合它们创建出很有意思的效果。

核(Kernel)（或卷积矩阵(Convolution Matrix)）是一个类矩阵的数值数组，它的中心为当前的像素，它会用它的核值乘以周围的像素值，并将结果相加变成一个值。所以，基本上我们是在对当前像素周围的纹理坐标添加一个小的偏移量，并根据核将结果合并。下面是核的一个例子：

这个核取了8个周围像素值，将它们乘以2，而把当前的像素乘以-15。这个核的例子将周围的像素乘上了一个权重，并将当前像素乘以一个比较大的负权重来平衡结果。

你在网上找到的大部分核将所有的权重加起来之后都应该会等于1，如果它们加起来不等于1，这就意味着最终的纹理颜色将会比原纹理值更亮或者更暗了。

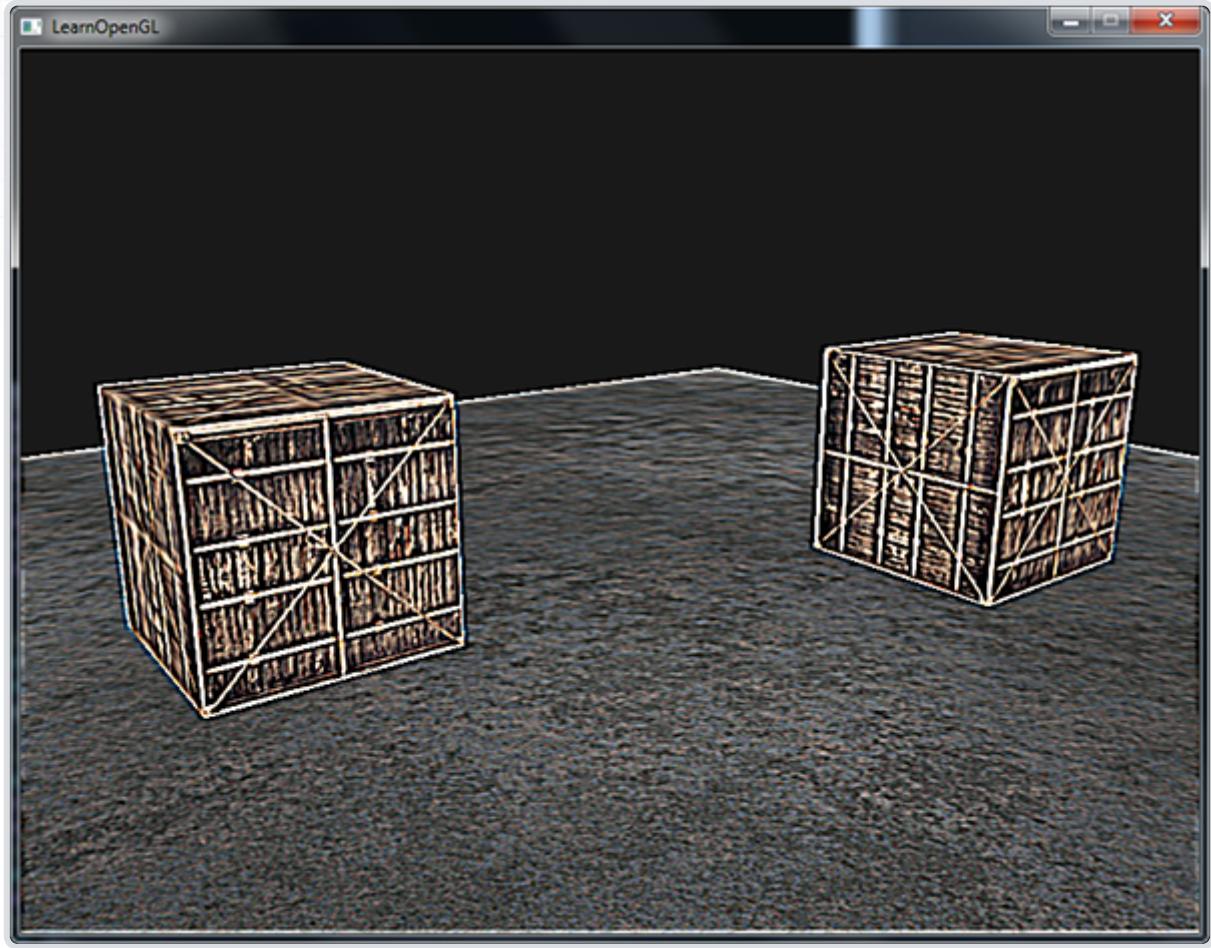
核是后期处理一个非常有用的工具，它们使用和实验起来都很简单，网上也能找到很多例子。我们需要稍微修改一下片段着色器，让它能够支持核。我们假设使用的核都是 3×3 核（实际上大部分核都是）：

```
const float offset = 1.0 / 300.0;

void main()
{
    vec2 offsets[9] = vec2[]( 
        vec2(-offset, -offset), // 左上
        vec2( 0.0f, -offset), // 正上
        vec2( offset, -offset), // 右上
        vec2(-offset,  0.0f), // 左
        vec2( 0.0f,  0.0f), // 中
        vec2( offset,  0.0f), // 右
        vec2(-offset, -offset), // 左下
        vec2( 0.0f, -offset), // 正下
        vec2( offset, -offset) // 右下
    );
    float kernel[9] = float[](
        -1, -1, -1,
        -1,  9, -1,
        -1, -1, -1
    );
    vec3 sampleTex[9];
    for(int i = 0; i < 9; i++)
    {
        sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));
    }
    vec3 col = vec3(0.0);
    for(int i = 0; i < 9; i++)
        col += sampleTex[i] * kernel[i];
    FragColor = vec4(col, 1.0);
}
```

在片段着色器中，我们首先为周围的纹理坐标创建了一个9个`vec2`偏移量的数组。偏移量是一个常量，你可以按照你的喜好自定义它。之后我们定义一个核，在这个例子中是一个锐化(Sharpen)核，它会采样周围的所有像素，锐化每个颜色值。最后，在采样时我们将每个偏移量加到当前纹理坐标上，获取需要采样的纹理，之后将这些纹理值乘以加权的核值，并将它们加到一起。

这个锐化核看起来是这样的：



这能创建一些很有趣的效果，比如说你的玩家打了麻醉剂所感受到的效果。

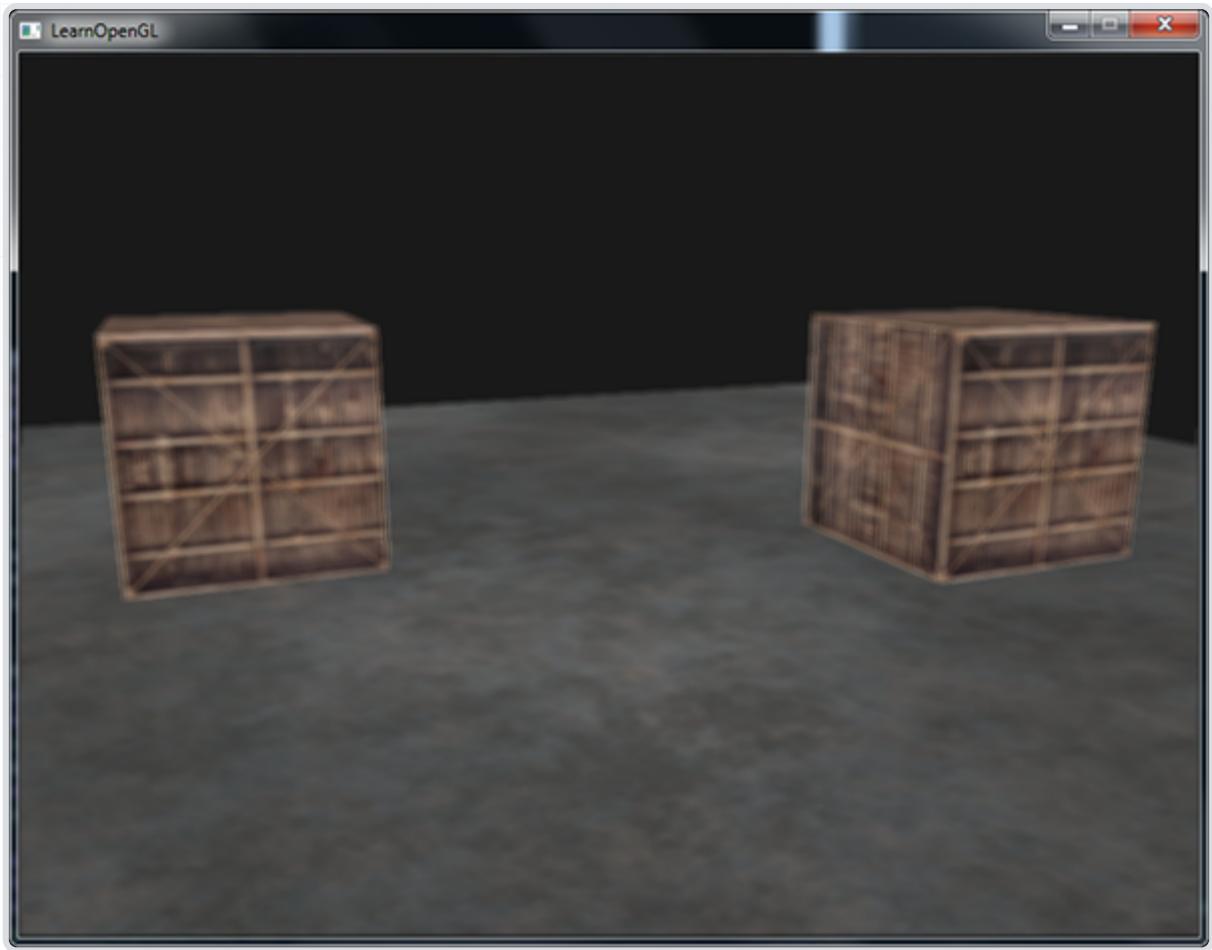
模糊

创建模糊(Blur)效果的核是这样的：

由于所有值的和是16，所以直接返回合并的采样颜色将产生非常亮的颜色，所以我们需要将核的每个值都除以16。最终的核数组将会是：

```
float kernel[9] = float[](
    1.0 / 16, 2.0 / 16, 1.0 / 16,
    2.0 / 16, 4.0 / 16, 2.0 / 16,
    1.0 / 16, 2.0 / 16, 1.0 / 16
);
```

通过在片段着色器中改变核的`float`数组，我们完全改变了后期处理效果。它现在看起来是这样子的：



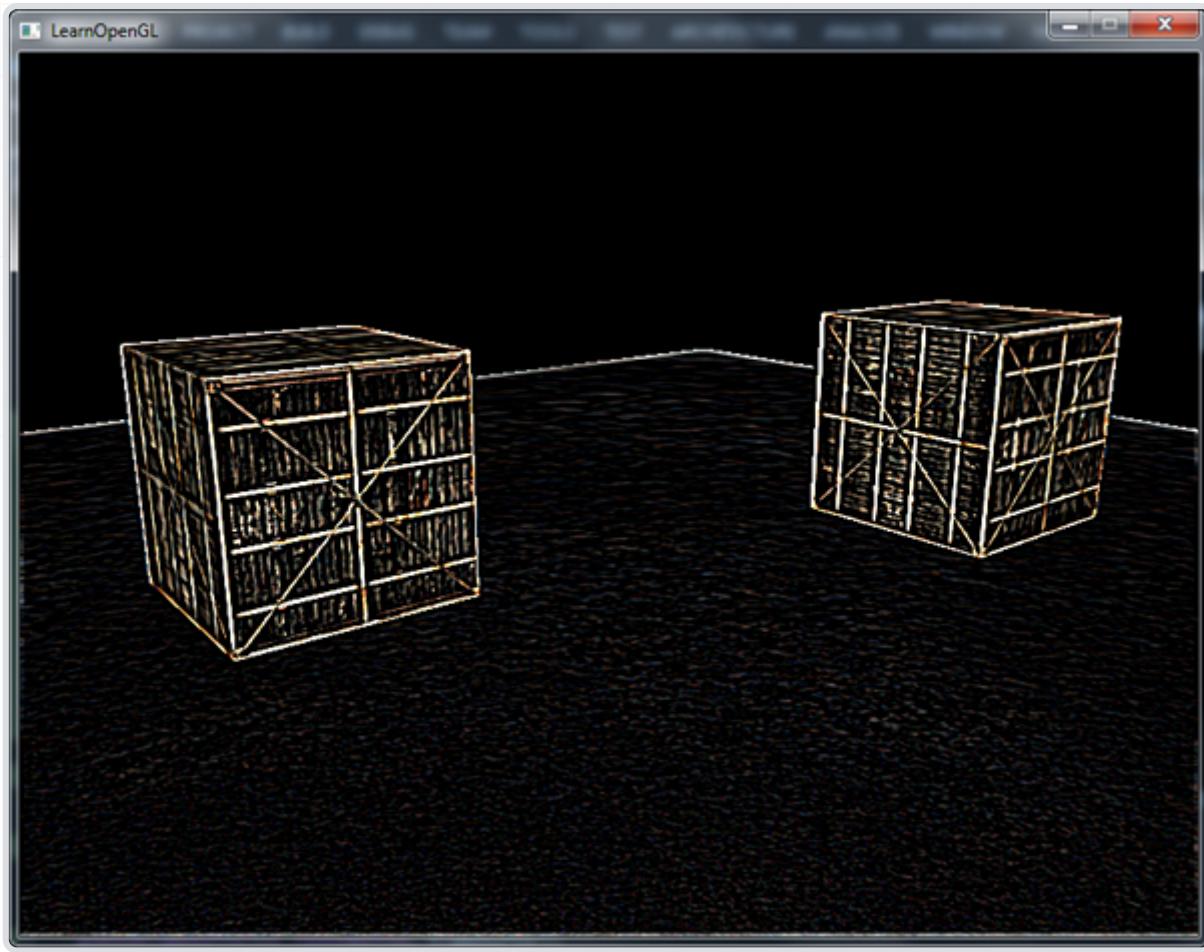
这样的模糊效果创造了很多的可能性。我们可以随着时间修改模糊的量，创造出玩家醉酒时的效果，或者在主角没带眼镜的时候增加模糊。模糊也能够让我们来平滑颜色值，我们将在之后教程中使用到。

你可以看到，只要我们有了这个核的实现，创建炫酷的后期处理特效是非常容易的事。我们再来看最后一个很流行的效果来结束本节的讨论。

边缘检测

下面的[边缘检测](#)(Edge-detection)核和锐化核非常相似：

这个核高亮了所有的边缘，而暗化了其它部分，在我们只关心图像的边角的时候是非常有用的。



你可能不会奇怪，像是Photoshop这样的图像修改工具/滤镜使用的也是这样的核。因为显卡处理片段的时候有着极强的并行处理能力，我们可以很轻松地在实时的情况下逐像素对图像进行处理。所以图像编辑工具在图像处理的时候会更倾向于使用显卡。

译注

注意，核在对屏幕纹理的边缘进行采样的时候，由于还会对中心像素周围的8个像素进行采样，其实会取到纹理之外的像素。由于环绕方式默认是`GL_REPEAT`，所以在没有设置的情况下取到的是屏幕另一边的像素，而另一边的像素本不应该对中心像素产生影响，这就可能会在屏幕边缘产生很奇怪的条纹。为了消除这一问题，我们可以将屏幕纹理的环绕方式都设置为`GL_CLAMP_TO_EDGE`。这样子在取到纹理外的像素时，就能够重复边缘的像素来更精确地估计最终的值了。

2.5.8 立方体贴图

原文 [Cubemaps](#)

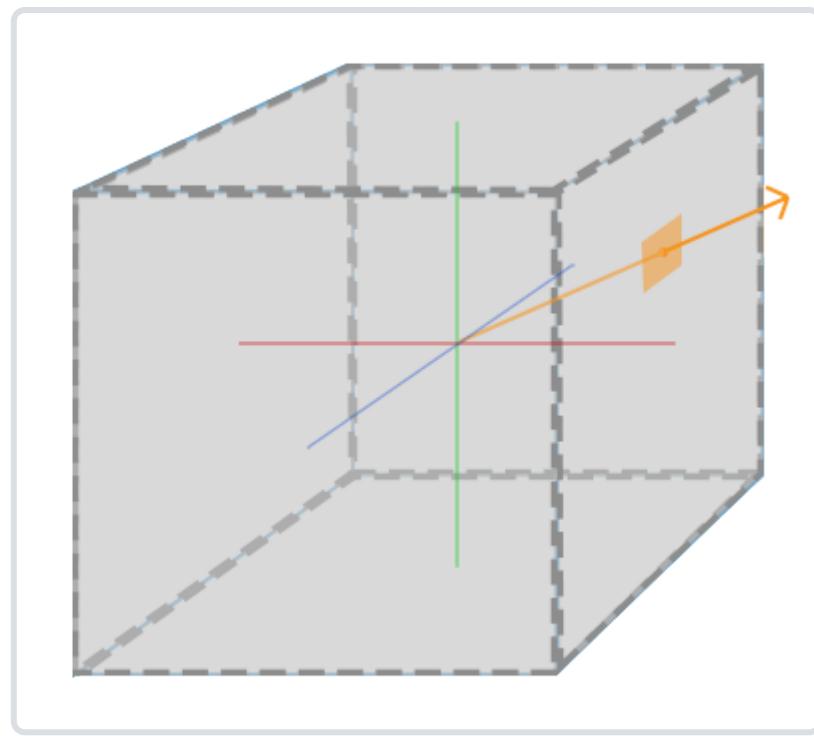
作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

我们已经使用2D纹理很长时间了，但除此之外仍有更多的纹理类型等着我们探索。在本节中，我们将讨论的是将多个纹理组合起来映射到一张纹理上的一种纹理类型：**立方体贴图**(Cube Map)。

简单来说，立方体贴图就是一个包含了6个2D纹理的纹理，每个2D纹理都组成了立方体的一个面：一个有纹理的立方体。你可能会奇怪，这样一个立方体有什么用途呢？为什么要把6张纹理合并到一张纹理中，而不是直接使用6个单独的纹理呢？立方体贴图有一个非常有用的特性，它可以通过一个方向向量来进行索引/采样。假设我们有一个 $1 \times 1 \times 1$ 的单位立方体，方向向量的原点位于它的中心。使用一个橘黄色的方向向量来从立方体贴图上采样一个纹理值会像是这样：



方向向量的大小并不重要，只要提供了方向，OpenGL就会获取方向向量（最终）所击中的纹素，并返回对应的采样纹理值。

如果我们假设将这样的立方体贴图应用到一个立方体上，采样立方体贴图所使用的方向向量将和立方体（插值的）顶点位置非常相像。这样子，只要立方体的中心位于原点，我们就能使用立方体的实际位置向量来对立方体贴图进行采样了。接下来，我们可以将所有顶点的纹理坐标当做是立方体的顶点位置。最终得到的结果就是可以访问立方体贴图上正确面(Face)纹理的一个纹理坐标。

创建立方体贴图

立方体贴图是和其它纹理一样的，所以如果想创建一个立方体贴图的话，我们需要生成一个纹理，并将其绑定到纹理目标上，之后再做其它的纹理操作。这次要绑定到`GL_TEXTURE_CUBE_MAP`：

```
unsigned int textureID;
 glGenTextures(1, &textureID);
 glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

因为立方体贴图包含有6个纹理，每个面一个，我们需要调用`glTexImage2D`函数6次，参数和之前教程中很类似。但这一次我们将纹理目标(target)参数设置为立方体贴图的一个特定的面，告诉OpenGL我们在对立方体贴图的哪一个面创建纹理。这就意味着我们需要对立方体贴图的每一个面都调用一次`glTexImage2D`。

由于我们有6个面，OpenGL给我们提供了6个特殊的纹理目标，专门对应立方体贴图的一个面。

纹理目标 方位

<code>GL_TEXTURE_CUBE_MAP_POSITIVE_X</code>	右
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_X</code>	左
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_Y</code>	上
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_Y</code>	下
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_Z</code>	后
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_Z</code>	前

和OpenGL的很多枚举(Enum)一样，它们背后的int值是线性递增的，所以如果我们有一个纹理位置的数组或者vector，我们就可以从`GL_TEXTURE_CUBE_MAP_POSITIVE_X`开始遍历它们，在每个迭代中对枚举值加1，遍历了整个纹理目标：

```
int width, height, nrChannels;
unsigned char *data;
for(unsigned int i = 0; i < textures_faces.size(); i++)
{
    data = stbi_load(textures_faces[i].c_str(), &width, &height, &nrChannels, 0);
    glTexImage2D(
        GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
        0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
    );
}
```

这里我们有一个叫做`textures_faces`的vector，它包含了立方体贴图所需的所有纹理路径，并以表中的顺序排列。这将为当前绑定的立方体贴图中的每个面生成一个纹理。

因为立方体贴图和其它纹理没什么不同，我们也需要设定它的环绕和过滤方式：

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

不要被`GL_TEXTURE_WRAP_R`吓到，它仅仅是为纹理的R坐标设置了环绕方式，它对应的是纹理的第三个维度（和位置的z一样）。我们将环绕方式设置为`GL_CLAMP_TO_EDGE`，这是因为正好处于两个面之间的纹理坐标可能不能击中一个面（由于一些硬件限制），所以通过使用`GL_CLAMP_TO_EDGE`，OpenGL将在我们对两个面之间采样的时候，永远返回它们的边界值。

在绘制使用立方体贴图的物体之前，我们要先激活对应的纹理单元，并绑定立方体贴图，这和普通的2D纹理没什么区别。

在片段着色器中，我们使用了一个不同类型的采样器，`samplerCube`，我们将使用`texture`函数使用它进行采样，但这次我们将使用一个`vec3`的方向向量而不是`vec2`。使用立方体贴图的片段着色器会像是这样的：

```
in vec3 textureDir; // 代表3D纹理坐标的方向向量
uniform samplerCube cubemap; // 立方体贴图的纹理采样器

void main()
{
    FragColor = texture(cubemap, textureDir);
}
```

看起来很棒，但为什么要用它呢？恰巧有一些很有意思的技术，使用立方体贴图来实现的话会简单多了。其中一个技术就是创建一个天空盒(Skybox)。

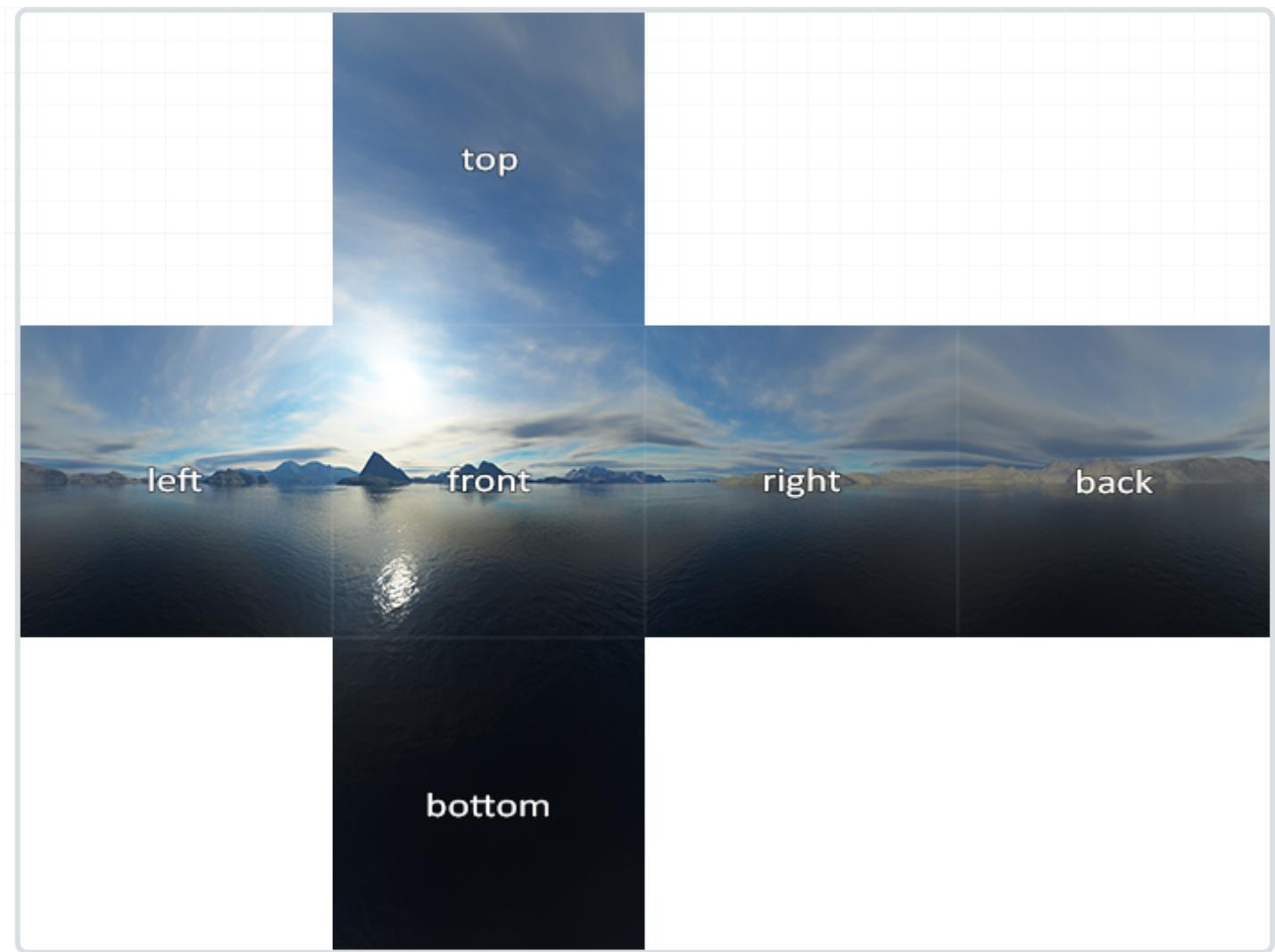
2.5.9 天空盒

天空盒是一个包含了整个场景的（大）立方体，它包含周围环境的6个图像，让玩家以为他处在一个比实际大得多的环境当中。游戏中使用天空盒的例子有群山、白云或星空。下面这张截图中展示的是星空的天空盒，它来自于『上古卷轴3』：



你可能现在已经猜到了，立方体贴图能完美满足天空盒的需求：我们有一个6面的立方体，每个面都需要一个纹理。在上面的图片中，他们使用了夜空的几张图片，让玩家产生其位于广袤宇宙中的错觉，但实际上他只是在一个小小的盒子当中。

你可以在网上找到很多像这样的天空盒资源。比如说这个[网站](#)就提供了很多天空盒。天空盒图像通常有以下的形式：



如果你将这六个面折成一个立方体，你就会得到一个完全贴图的立方体，模拟一个巨大的场景。一些资源可能会提供了这样格式的天空盒，你必须手动提取六个面的图像，但在大部分情况下它们都是6张单独的纹理图像。

之后我们将在场景中使用这个（高质量的）天空盒，它可以在[这里](#)下载到。

加载天空盒

因为天空盒本身就是一个立方体贴图，加载天空盒和之前加载立方体贴图时并没有什么不同。为了加载天空盒，我们将使用下面的函数，它接受一个包含6个纹理路径的vector：

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                         0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
                         );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }

    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

    return textureID;
}
```

函数本身应该很熟悉了。它基本就是上一部分中立方体贴图的代码，只不过合并到了一个便于管理的函数中。

之后，在调用这个函数之前，我们需要将合适的纹理路径按照立方体贴图枚举指定的顺序加载到一个vector中。

```
vector<std::string> faces
{
    "right.jpg",
    "left.jpg",
    "top.jpg",
    "bottom.jpg",
    "front.jpg",
    "back.jpg"
};

unsigned int cubemapTexture = loadCubemap(faces);
```

现在我们就将这个天空盒加载为一个立方体贴图了，它的id是cubemapTexture。我们可以将它绑定到一个立方体中，替换掉用了很长时期的难看的纯色背景。

显示天空盒

由于天空盒是绘制在一个立方体上的，和其它物体一样，我们需要另一个VAO、VBO以及新的一组顶点。你可以在[这里](#)找到它的顶点数据。

用于贴图3D立方体的立方体贴图可以使用立方体的位置作为纹理坐标来采样。当立方体处于原点(0, 0, 0)时，它的每一个位置向量都是从原点出发的方向向量。这个方向向量正是获取立方体上特定位置的纹理值所需要的。正是因为这个，我们只需要提供位置向量而不用纹理坐标了。

要渲染天空盒的话，我们需要一组新的着色器，它们都不是很复杂。因为我们只有一个顶点属性，顶点着色器非常简单：

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
}
```

注意，顶点着色器中很有意思的部分是，我们将输入的位置向量作为输出给片段着色器的纹理坐标。片段着色器会将它作为输入来采样

`samplerCube`：

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```

片段着色器非常直观。我们将顶点属性的位置向量作为纹理的方向向量，并使用它从立方体贴图中采样纹理值。

有了立方体贴图纹理，渲染天空盒现在就非常简单了，我们只需要绑定立方体贴图纹理，`skybox`采样器就会自动填充上天空盒立方体贴图了。绘制天空盒时，我们需要将它变为场景中的第一个渲染的物体，并且禁用深度写入。这样子天空盒就会永远被绘制在其它物体的背后了。

```
glDepthMask(GL_FALSE);
skyboxShader.use();
// ... 设置观察和投影矩阵
 glBindVertexArray(skyboxVA0);
 glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
 glDrawArrays(GL_TRIANGLES, 0, 36);
 glDepthMask(GL_TRUE);
// ... 绘制剩下的场景
```

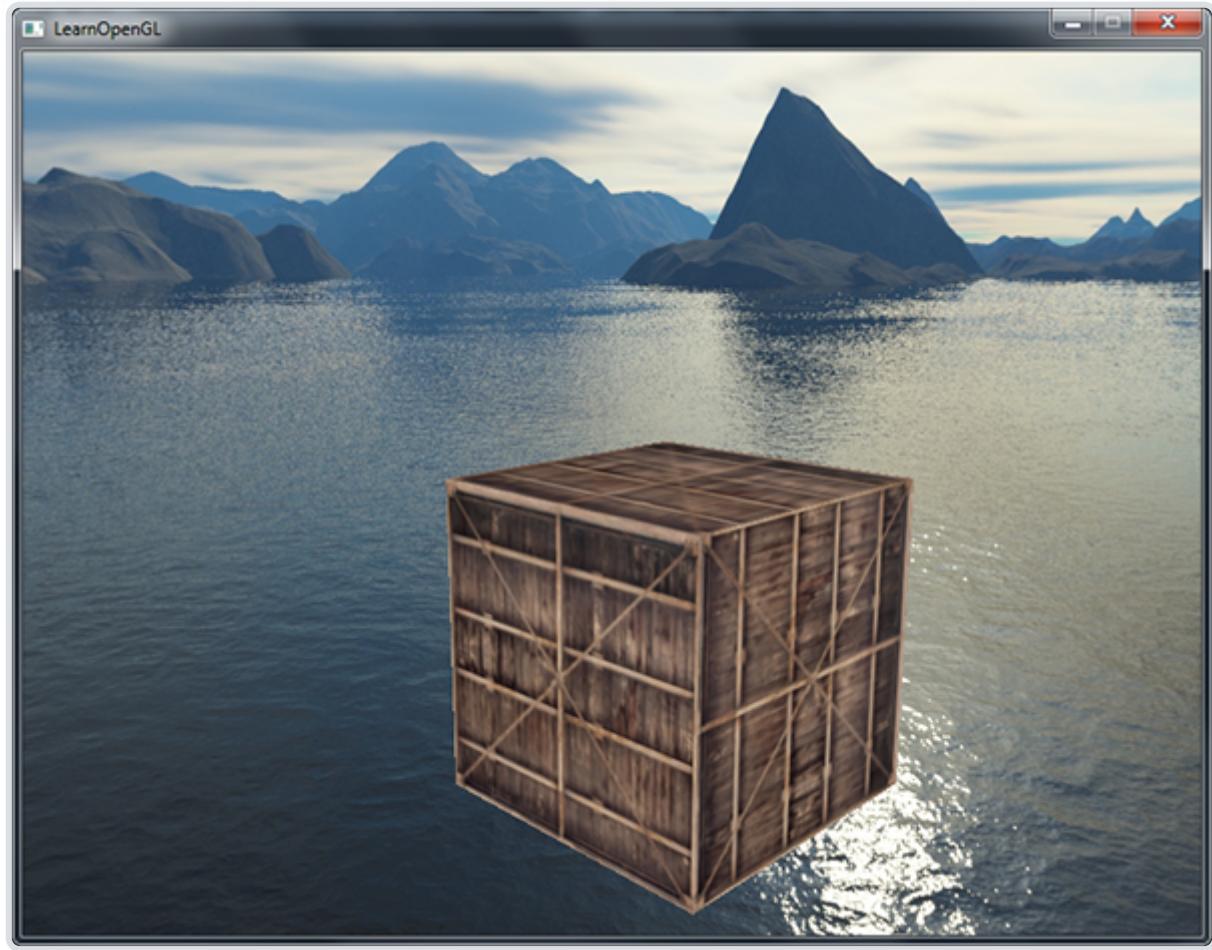
如果你运行一下的话你就会发现出现了一些问题。我们希望天空盒是以玩家为中心的，这样不论玩家移动了多远，天空盒都不会变近，让玩家产生周围环境非常大的印象。然而，当前的观察矩阵会旋转、缩放和位移来变换天空盒的所有位置，所以当玩家移动的时候，立方体贴图也会移动！我们希望移除观察矩阵中的位移部分，让移动不会影响天空盒的位置向量。

你可能还记得在[基础光照](#)小节中，我们通过取 4×4 矩阵左上角的 3×3 矩阵来移除变换矩阵的位移部分。我们可以将观察矩阵转换为 3×3 矩阵（移除位移），再将其转换回 4×4 矩阵，来达到类似的效果。

```
glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

这将移除任何的位移，但保留旋转变换，让玩家仍然能够环顾场景。

有了天空盒，最终的效果就是一个看起来巨大的场景了。如果你在箱子周围转一转，你就能立刻感受到距离感，极大地提升了场景的真实度。最终的结果看起来是这样的：



试一试不同的天空盒，看看它们是怎样对场景的观感产生巨大影响的。

优化

目前我们是首先渲染天空盒，之后再渲染场景中的其它物体。这样子能够工作，但不是非常高效。如果我们先渲染天空盒，我们就会对屏幕上的每一个像素运行一遍片段着色器，即便只有一小部分的天空盒最终是可见的。可以使用[提前深度测试](#)(Early Depth Testing)轻松丢弃掉的片段能够节省我们很多宝贵的带宽。

所以，我们将会最后渲染天空盒，以获得轻微的性能提升。这样子的话，深度缓冲就会填充满所有物体的深度值了，我们只需要在提前深度测试通过的地方渲染天空盒的片段就可以了，很大程度上减少了片段着色器的调用。问题是，天空盒只是一个 $1 \times 1 \times 1$ 的立方体，它很可能会不通过大部分的深度测试，导致渲染失败。不用深度测试来进行渲染不是解决方案，因为天空盒将会复写场景中的其它物体。我们需要欺骗深度缓冲，让它认为天空盒有着最大的深度值1.0，只要它前面有一个物体，深度测试就会失败。

在[坐标系统](#)小节中我们说过，透视线除法是在顶点着色器运行之后执行的，将`gl_Position`的`xyz`坐标除以w分量。我们又从[深度测试](#)小节中知道，相除结果的z分量等于顶点的深度值。使用这些信息，我们可以将输出位置的z分量等于它的w分量，让z分量永远等于1.0，这样子的话，当透视线除法执行之后，z分量会变为`w / w = 1.0`。

```
void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

最终的标准化设备坐标将永远会有一个等于1.0的z值：最大的深度值。结果就是天空盒只会在没有可见物体的地方渲染了（只有这样才能通过深度测试，其它所有的东西都在天空盒前面）。

我们还要改变一下深度函数，将它从默认的`GL_LESS`改为`GL_LEQUAL`。深度缓冲将会填充上天空盒的1.0值，所以我们需要保证天空盒在值小于或等于深度缓冲而不是小于时通过深度测试。

你可以在[这里](#)找到优化后的源代码。

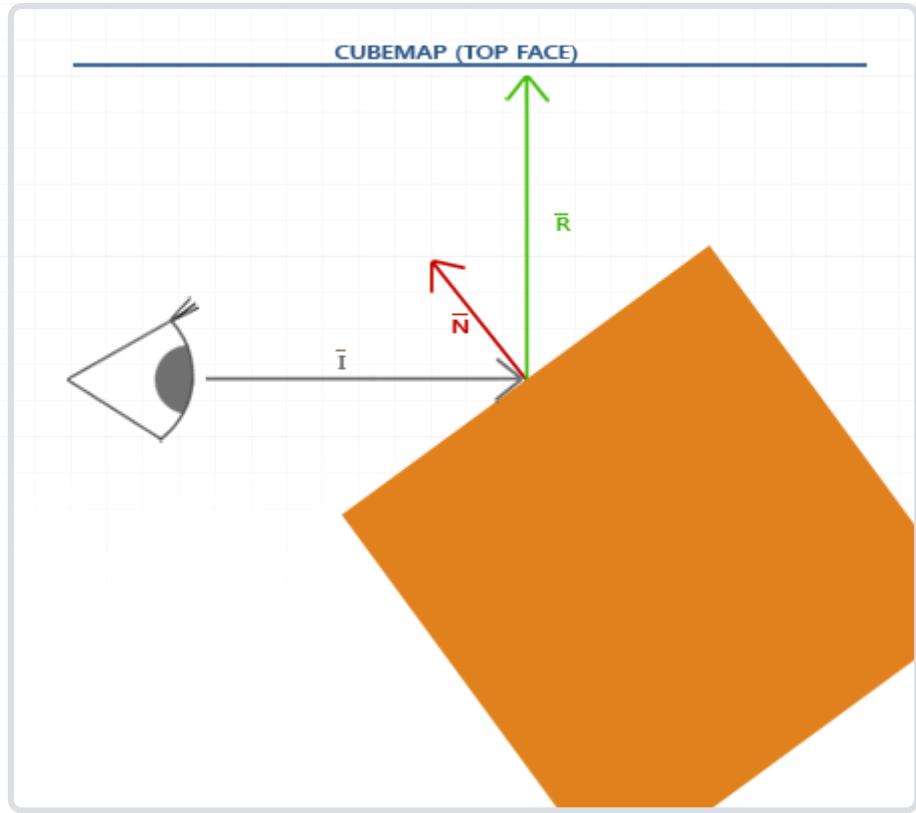
2.5.10 环境映射

我们现在将整个环境映射到了一个纹理对象上了，能利用这个信息的不仅仅只有天空盒。通过使用环境的立方体贴图，我们可以给物体反射和折射的属性。这样使用环境立方体贴图的技术叫做[环境映射](#)(Environment Mapping)，其中最流行的两个是[反射](#)(Reflection)和[折射](#)(Refraction)。

反射

反射这个属性表现为物体（或物体的一部分）[反射](#)它周围环境，即根据观察者的视角，物体的颜色或多或少等于它的环境。镜子就是一个反射性物体：它会根据观察者的视角反射它周围的环境。

反射的原理并不难。下面这张图展示了我们如何计算反射向量，并如何使用这个向量来从立方体贴图中采样：



我们根据观察方向向量和物体的法向量，来计算反射向量。我们可以使用GLSL内建的`reflect`函数来计算这个反射向量。最终的向量将会作为索引/采样立方体贴图的方向向量，返回环境的颜色值。最终的结果是物体看起来反射了天空盒。

因为我们已经在场景中配置好天空盒了，创建反射效果并不会很难。我们将会改变箱子的片段着色器，让箱子有反射性：

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 Position;

uniform vec3 cameraPos;
uniform samplerCube skybox;

void main()
{
    vec3 I = normalize(Position - cameraPos);
    vec3 R = reflect(I, normalize(Normal));
    FragColor = vec4(texture(skybox, R.rgb, 1.0);
}
```

我们先计算了观察/摄像机方向向量 I ，并使用它来计算反射向量 R ，之后我们将使用 R 来从天空盒立方体贴图中采样。注意，我们现在又有了片段的插值 `Normal` 和 `Position` 变量，所以我们需要更新一下顶点着色器。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 Normal;
out vec3 Position;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    Normal = mat3(transpose(inverse(model))) * aNormal;
    Position = vec3(model * vec4(aPos, 1.0));
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

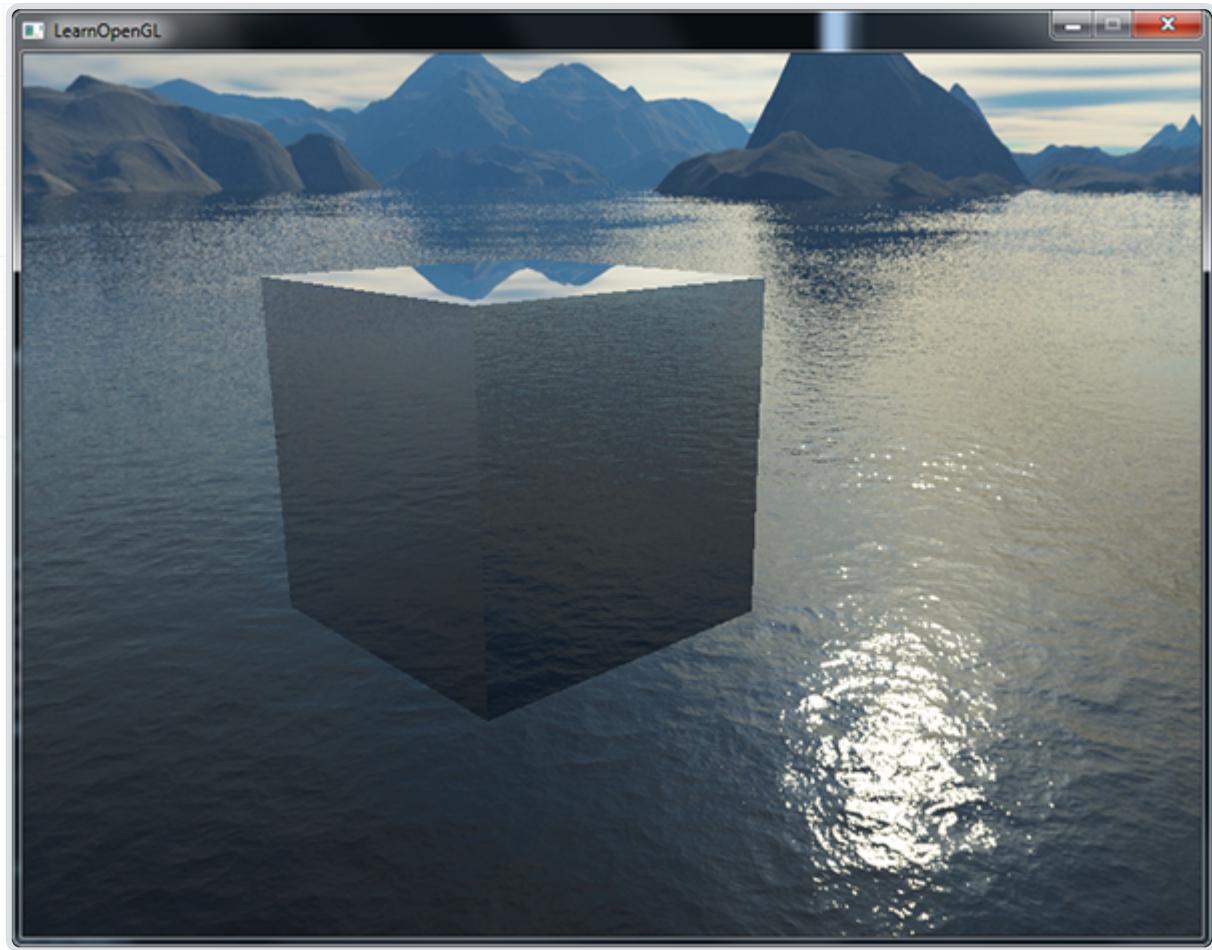
我们现在使用了一个法向量，所以我们将再次使用法线矩阵(Normal Matrix)来变换它们。`Position`输出向量是一个世界空间的位置向量。顶点着色器的这个`Position`输出将用来在片段着色器内计算观察方向向量。

因为我们使用了法线，你还需要更新一下[顶点数据](#)，并更新属性指针。还要记得去设置`cameraPos`这个uniform。

接下来，我们在渲染箱子之前先绑定立方体贴图纹理：

```
glBindVertexArray(cubeVA0);
glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

编译并运行代码，你将会得到一个像是镜子一样的箱子。周围的天空盒被完美地反射在箱子上。



你可以在[这里](#)找到完整的源代码。

当反射应用到一整个物体上（像是箱子）时，这个物体看起来就像是钢或者铬这样的高反射性材质。如果我们加载[模型加载](#)小节中的纳米装模型，我们会得到一种整个套装都是使用铬做成的效果：

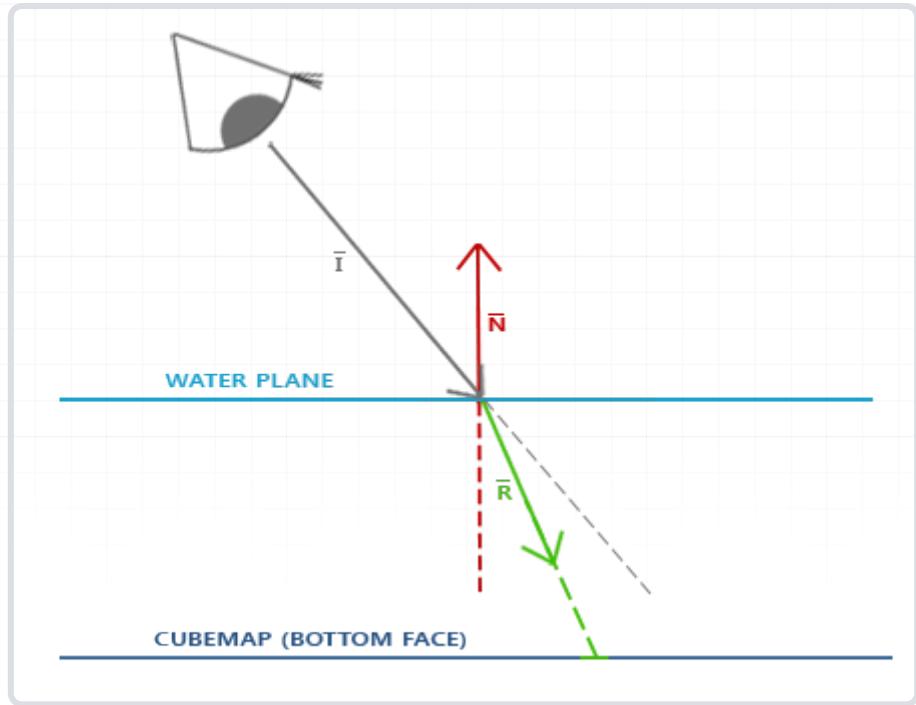


这看起来非常棒，但在现实中大部分的模型都不具有完全反射性。我们可以引入[反射贴图](#)(Reflection Map)，来给模型更多的细节。与漫反射和镜面光贴图一样，反射贴图也是可以采样的纹理图像，它决定这片段的反射性。通过使用反射贴图，我们可以知道模型的哪些部分该以什么强度显示反射。在本节的练习中，将由你来为我们之前创建的模型加载器中引入反射贴图，显著提升纳米装模型的细节。

折射

环境映射的另一种形式是[折射](#)，它和反射很相似。折射是光线由于传播介质的改变而产生的方向变化。在常见的类水表面上所产生的现象就是折射，光线不是直直地传播，而是弯曲了一点。将你的半只胳膊伸进水里，观察出来的就是这种效果。

折射是通过[斯涅尔定律](#)(Snell's Law)来描述的，使用环境贴图的话看起来像是这样：



同样，我们有一个观察向量，一个法向量，而这次是折射向量。可以看到，观察向量的方向轻微弯曲了。弯折后的向量将会用来从立方体贴图中采样。

折射可以使用GLSL的内建`refract`函数来轻松实现，它需要一个法向量、一个观察方向和两个材质之间的折射率(Refractive Index)。

折射率决定了材质中光线弯曲的程度，每个材质都有自己的折射率。一些最常见的折射率可以在下表中找到：

材质 折射率

空气 1.00

水 1.33

冰 1.309

玻璃 1.52

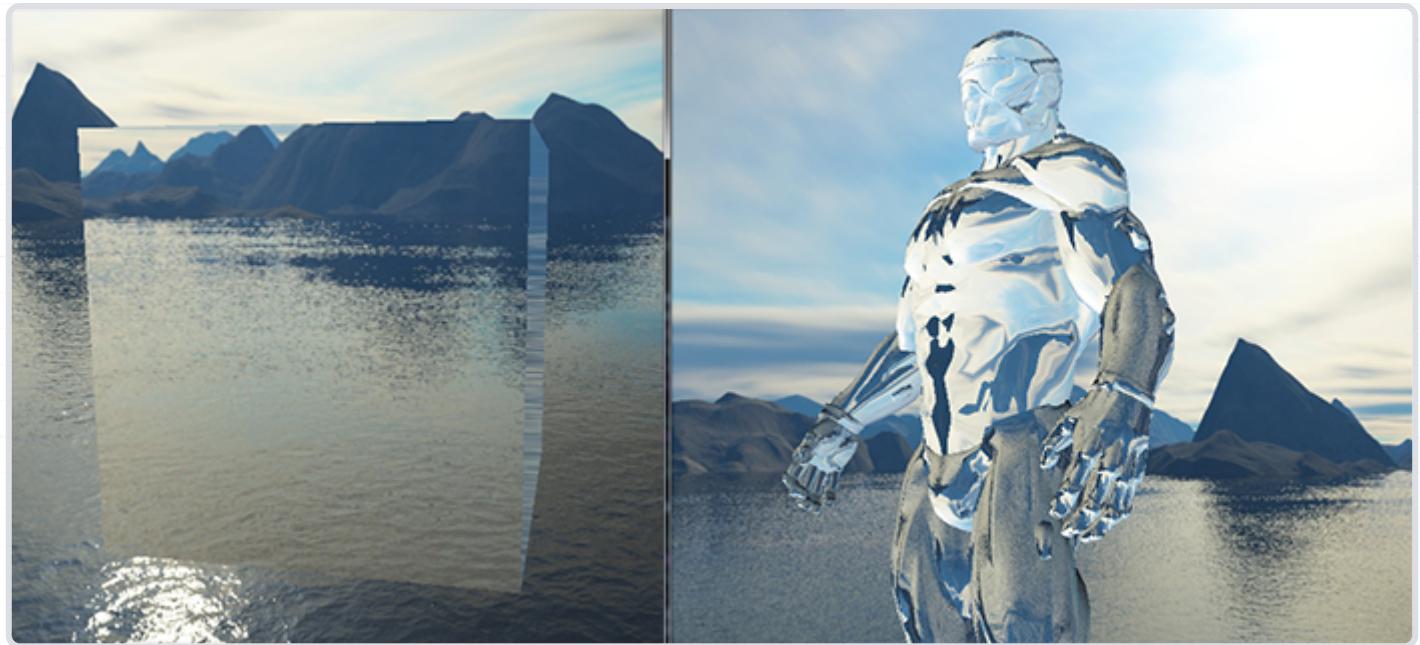
钻石 2.42

我们使用这些折射率来计算光传播的两种材质间的比值。在我们的例子中，光线/视线从空气进入玻璃（如果我们假设箱子是玻璃制的），所以比值为。

我们已经绑定了立方体贴图，提供了顶点数据和法线，并设置了摄像机位置的uniform。唯一要修改的就是片段着色器：

```
void main()
{
    float ratio = 1.00 / 1.52;
    vec3 I = normalize(Position - cameraPos);
    vec3 R = refract(I, normalize(Normal), ratio);
    FragColor = vec4(texture(skybox, R).rgb, 1.0);
}
```

通过改变折射率，你可以创建完全不同的视觉效果。编译程序并运行，但结果并不是很有趣，因为我们只使用了一个简单的箱子，它不太能显示折射的效果，现在看起来只是有点像一个放大镜。对纳米装使用相同的着色器却能够展现出了我们期待的效果：一个类玻璃的物体。



你可以想象出有了光照、反射、折射和顶点移动的正确组合，你可以创建出非常漂亮的水。注意，如果要想获得物理上精确的结果，我们还需要在光线离开物体的时候再次折射，现在我们使用的只是单面折射(Single-side Refraction)，但它对大部分场合都是没问题的。

动态环境贴图

现在我们使用的都是静态图像的组合来作为天空盒，看起来很不错，但它没有在场景中包括可移动的物体。我们一直都没有注意到这一点，因为我们只使用了一个物体。如果我们有一个镜子一样的物体，周围还有多个物体，镜子中可见的只有天空盒，看起来就像它是场景中唯一一个物体一样。

通过使用帧缓冲，我们能够为物体的6个不同角度创建出场景的纹理，并在每个渲染迭代中将它们储存到一个立方体贴图中。之后我们就可以使用这个（动态生成的）立方体贴图来创建出更真实的，包含其它物体的，反射和折射表面了。这就叫做**动态环境映射**(Dynamic Environment Mapping)，因为我们动态创建了物体周围的立方体贴图，并将其用作环境贴图。

虽然它看起来很棒，但它有一个很大的缺点：我们需要为使用环境贴图的物体渲染场景6次，这是对程序是非常大的性能开销。现代的程序通常会尽可能使用天空盒，并在可能的时候使用预编译的立方体贴图，只要它们能产生一点动态环境贴图的效果。虽然动态环境贴图是一个很棒的技术，但是要想在不降低性能的情况下让它工作还是需要非常多的技巧的。

练习

- 尝试在我们之前在[模型加载](#)小节中创建的模型加载器中引入反射贴图。你可以在[这里](#)找到升级后有反射贴图的纳米装模型。仍有几点要注意的：
 - Assimp在大多数格式中都不太喜欢反射贴图，所以我们需要欺骗一下它，将反射贴图储存为漫反射贴图。你可以在加载材质的时候将反射贴图的纹理类型设置为`aiTextureType_AMBIENT`。
 - 我偷懒直接使用镜面光纹理图像来创建了反射贴图，所以反射贴图在模型的某些地方不能准确地映射:)。
 - 由于模型加载器本身就已经在着色器中占用了3个纹理单元了，你需要将天空盒绑定到第4个纹理单元上，因为我们要从同一个着色器中对天空盒采样。
- 如果你都做对了，它会看起来像[这样](#)。

2.5.11 高级数据

原文 [Advanced Data](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

我们在OpenGL中大量使用缓冲来储存数据已经有很长时间了。操作缓冲其实还有更有意思的方式，而且使用纹理将大量数据传入着色器也有更有趣的方法。这一节中，我们将讨论一些更有意思的缓冲函数，以及我们该如何使用纹理对象来储存大量的数据（纹理的部分还没有完成）。

OpenGL中的缓冲只是一个管理特定内存块的对象，没有其它更多的功能了。在我们将它绑定到一个**缓冲目标**(Buffer Target)时，我们才赋予了其意义。当我们绑定一个缓冲到**GL_ARRAY_BUFFER**时，它就是一个顶点数组缓冲，但我们也很容易地将其绑定到**GL_ELEMENT_ARRAY_BUFFER**。OpenGL内部会为每个目标储存一个缓冲，并且会根据目标的不同，以不同的方式处理缓冲。

到目前为止，我们一直是调用**glBufferData**函数来填充缓冲对象所管理的内存，这个函数会分配一块内存，并将数据添加到这块内存中。如果我们将它的**data**参数设置为**NULL**，那么这个函数将只会分配内存，但不进行填充。这在我们需要预留(Reserve)特定大小的内存，之后回到这个缓冲一点一点填充的时候会很有用。

除了使用一次函数调用填充整个缓冲之外，我们也可以使用**glBufferSubData**，填充缓冲的特定区域。这个函数需要一个缓冲目标、一个偏移量、数据的大小和数据本身作为它的参数。这个函数不同的地方在于，我们可以提供一个偏移量，指定从何处开始填充这个缓冲。这能够让我们插入或者更新缓冲内存的某一部分。要注意的是，缓冲需要有足够的已分配内存，所以对一个缓冲调用**glBufferSubData**之前必须要先调用**glBufferData**。

```
glBufferSubData(GL_ARRAY_BUFFER, 24, sizeof(data), &data); // 范围: [24, 24 + sizeof(data)]
```

将数据导入缓冲的另外一种方法是，请求缓冲内存的指针，直接将数据复制到缓冲当中。通过调用**glMapBuffer**函数，OpenGL会返回当前绑定缓冲的内存指针，供我们操作：

```
float data[] = {
    0.5f, 1.0f, -0.35f
    ...
};

glBindBuffer(GL_ARRAY_BUFFER, buffer);
// 获取指针
void *ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
// 复制数据到内存
memcpy(ptr, data, sizeof(data));
// 记得告诉OpenGL我们不再需要这个指针了
glUnmapBuffer(GL_ARRAY_BUFFER);
```

当我们使用**glUnmapBuffer**函数，告诉OpenGL我们已经完成指针操作之后，OpenGL就会知道你已经完成了。在解除映射(Unmapping)之后，指针将会不再可用，并且如果OpenGL能够成功将您的数据映射到缓冲中，这个函数将会返回**GL_TRUE**。

如果要直接映射数据到缓冲，而不事先将其存储到临时内存中，**glMapBuffer**这个函数会很有用。比如说，你可以从文件中读取数据，并直接将它们复制到缓冲内存中。

分批顶点属性

通过使用`glVertexAttribPointer`, 我们能够指定顶点数组缓冲内容的属性布局。在顶点数组缓冲中, 我们对属性进行了交错(Interleave)处理, 也就是说, 我们将每一个顶点的位置、法线和/或纹理坐标紧密放置在一起。既然我们现在已经对缓冲有了更多的了解, 我们可以采取另一种方式。

我们可以做的是, 将每一种属性类型的向量数据打包(Batch)为一个大的区块, 而不是对它们进行交错储存。与交错布局123123123123不同, 我们将采用分批(Batched)的方式111122223333。

当从文件中加载顶点数据的时候, 你通常获取到的是一个位置数组、一个法线数组和/或一个纹理坐标数组。我们需要花点力气才能将这些数组转化为一个大的交错数据数组。使用分批的方式会是更简单的解决方案, 我们可以很容易使用`glBufferSubData`函数实现:

```
float positions[] = { ... };
float normals[] = { ... };
float tex[] = { ... };
// 填充缓冲
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(positions), &positions);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions), sizeof(normals), &normals);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions) + sizeof(normals), sizeof(tex), &tex);
```

这样子我们就能直接将属性数组作为一个整体传递给缓冲, 而不需要事先处理它们了。我们仍可以将它们合并为一个大的数组, 再使用`glBufferData`来填充缓冲, 但对于这种工作, 使用`glBufferSubData`会更合适一点。

我们还需要更新顶点属性指针来反映这些改变:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)(sizeof(positions)));
glVertexAttribPointer(
    2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)(sizeof(positions) + sizeof(normals)));
```

注意`stride`参数等于顶点属性的大小, 因为下一个顶点属性向量能在3个(或2个)分量之后找到。

这给了我们设置顶点属性的另一种方法。使用哪种方法都不会对OpenGL有什么立刻的好处, 它只是设置顶点属性的一种更整洁的方式。具体使用的方法将完全取决于你的喜好与程序类型。

复制缓冲

当你的缓冲已经填充好数据之后, 你可能会想与其它的缓冲共享其中的数据, 或者想要将缓冲的内容复制到另一个缓冲当中。

`glCopyBufferSubData`能够让我们相对容易地从一个缓冲中复制数据到另一个缓冲中。这个函数的原型如下:

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr readoffset,
                        GLintptr writeoffset, GLsizeiptr size);
```

`readtarget`和`writetarget`参数需要填入复制源和复制目标的缓冲目标。比如说, 我们可以将`VERTEX_ARRAY_BUFFER`缓冲复制到`VERTEX_ELEMENT_ARRAY_BUFFER`缓冲, 分别将这些缓冲目标设置为读和写的目标。当前绑定到这些缓冲目标的缓冲将会被影响到。

但如果我们想读写数据的两个不同缓冲都为顶点数组缓冲该怎么办呢? 我们不能同时将两个缓冲绑定到同一个缓冲目标上。正是出于这个原因, OpenGL提供给我们另外两个缓冲目标, 叫做`GL_COPY_READ_BUFFER`和`GL_COPY_WRITE_BUFFER`。我们接下来就可以将需要的缓冲绑定到这两个缓冲目标上, 并将这两个目标作为`readtarget`和`writetarget`参数。

接下来`glCopyBufferSubData`会从`readtarget`中读取`size`大小的数据，并将其写入`writetarget`缓冲的`writeoffset`偏移量处。下面这个例子展示了如何复制两个顶点数组缓冲：

```
float vertexData[] = { ... };
glBindBuffer(GL_COPY_READ_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0, sizeof(vertexData));
```

我们也可以只将`writetarget`缓冲绑定为新的缓冲目标类型之一：

```
float vertexData[] = { ... };
glBindBuffer(GL_ARRAY_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_ARRAY_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0, sizeof(vertexData));
```

有了这些关于如何操作缓冲的额外知识，我们已经能够以更有意思的方式使用它们了。当你越深入OpenGL时，这些新的缓冲方法将会变得更加有用。在[下一节](#)中，在我们讨论Uniform缓冲对象(Uniform Buffer Object)时，我们将会充分利用`glBufferSubData`。

2.5.12 高级GLSL

原文 [Advanced GLSL](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

这一小节并不会向你展示非常先进非常酷的新特性，也不会对场景的视觉质量有显著的提高。但是，这一节会或多或少涉及GLSL的一些有趣的地方以及一些很棒的技巧，它们可能在今后会帮助到你。简单来说，它们就是在组合使用OpenGL和GLSL创建程序时的一些最好要知道的东西，和一些会让你生活更加轻松的特性。

我们将会讨论一些有趣的**内建变量**(Built-in Variable)，管理着色器输入和输出的新方式以及一个叫做**Uniform缓冲对象**(Uniform Buffer Object)的有用工具。

2.5.13 GLSL的内建变量

着色器都是最简化的，如果需要当前着色器以外地方的数据的话，我们必须要将数据传进来。我们已经学会使用顶点属性、uniform和采样器来完成这一任务了。然而，除此之外，GLSL还定义了另外几个以`gl_`为前缀的变量，它们能提供给我们更多的方式来读取/写入数据。我们已经在前面教程中接触过其中的两个了：顶点着色器的输出向量`gl_Position`，和片段着色器的`gl_FragCoord`。

我们将会讨论几个有趣的GLSL内建输入和输出变量，并会解释它们能够怎样帮助你。注意，我们将不会讨论GLSL中存在的所有内建变量，如果你想知道所有的内建变量的话，请查看OpenGL的[wiki](#)。

顶点着色器变量

我们已经见过`gl_Position`了，它是顶点着色器的裁剪空间输出位置向量。如果你想在屏幕上显示任何东西，在顶点着色器中设置`gl_Position`是必须的步骤。这已经是它的全部功能了。

`gl_PointSize`

我们能够选用的其中一个图元是`GL_POINTS`，如果使用它的话，每一个顶点都是一个图元，都会被渲染为一个点。我们可以通过OpenGL的`glPointSize`函数来设置渲染出来的点的大小，但我们也可以在顶点着色器中修改这个值。

GLSL定义了一个叫做`gl_PointSize`输出变量，它是一个`float`变量，你可以使用它来设置点的宽高（像素）。在顶点着色器中修改点的大小的话，你就能对每个顶点设置不同的值了。

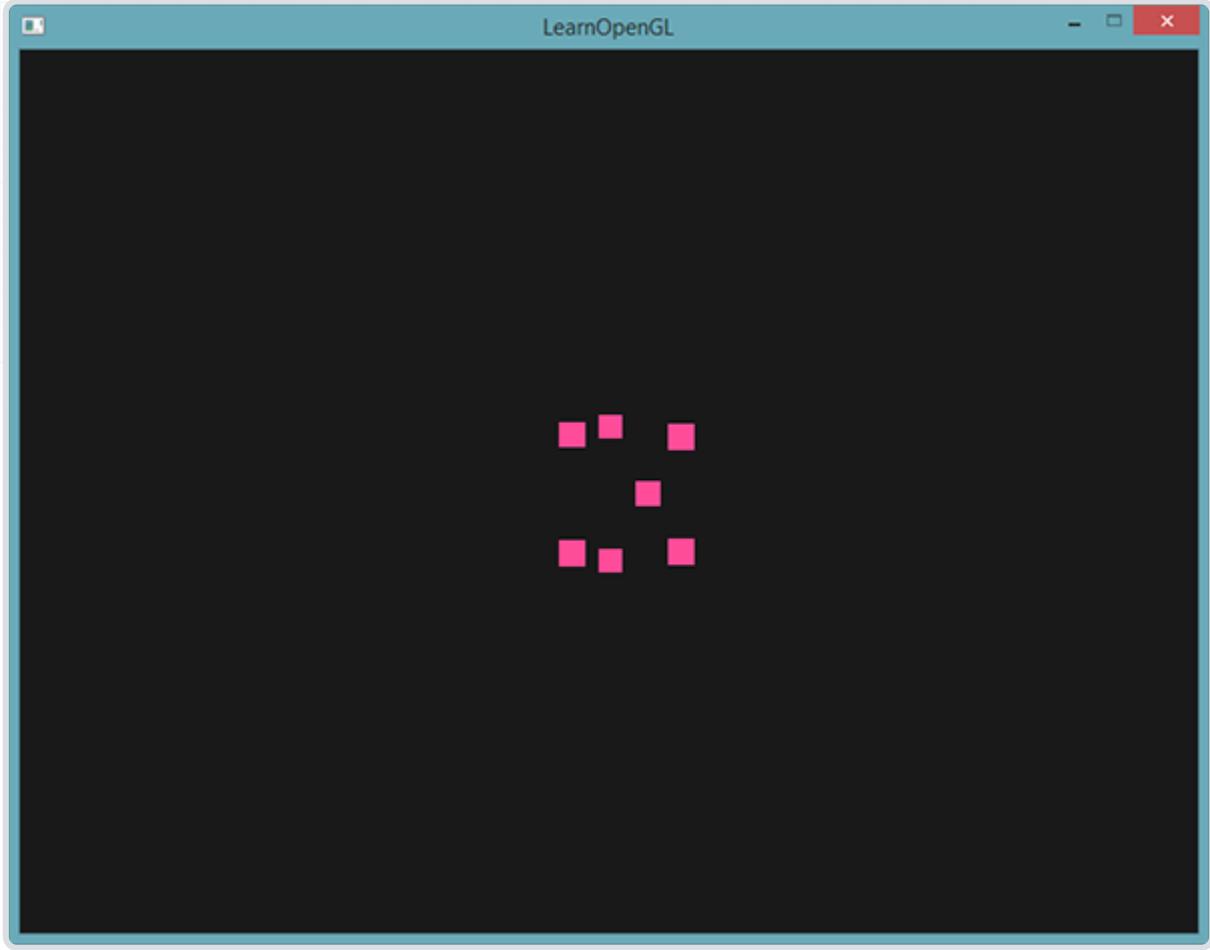
在顶点着色器中修改点大小的功能默认是禁用的，如果你需要启用它的话，你需要启用OpenGL的`GL_PROGRAM_POINT_SIZE`：

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

一个简单的例子就是将点的大小设置为裁剪空间位置的`z`值，也就是顶点距观察者的距离。点的大小会随着观察者距顶点距离变远而增大。

```
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    gl_PointSize = gl_Position.z;
}
```

结果就是，当我们远离这些点的时候，它们会变得更大：



你可以想到，对每个顶点使用不同的点大小，会在粒子生成之类的技术中很有意思。

gl_VertexID

`gl_Position`和`gl_PointSize`都是输出变量，因为它们的值是作为顶点着色器的输出被读取的。我们可以对它们进行写入，来改变结果。顶点着色器还为我们提供了一个有趣的输入变量，我们只能对它进行读取，它叫做`gl_VertexID`。

整型变量`gl_VertexID`储存了正在绘制顶点的当前ID。当（使用`glDrawElements`）进行索引渲染的时候，这个变量会存储正在绘制顶点的当前索引。当（使用`glDrawArrays`）不使用索引进行绘制的时候，这个变量会储存从渲染调用开始的已处理顶点数量。

虽然现在它没有什么具体的用途，但知道我们能够访问这个信息总是好的。

片段着色器变量

在片段着色器中，我们也能访问到一些有趣的变量。GLSL提供给我们两个有趣的输入变量：`gl_FragCoord`和`gl_FrontFacing`。

gl_FragCoord

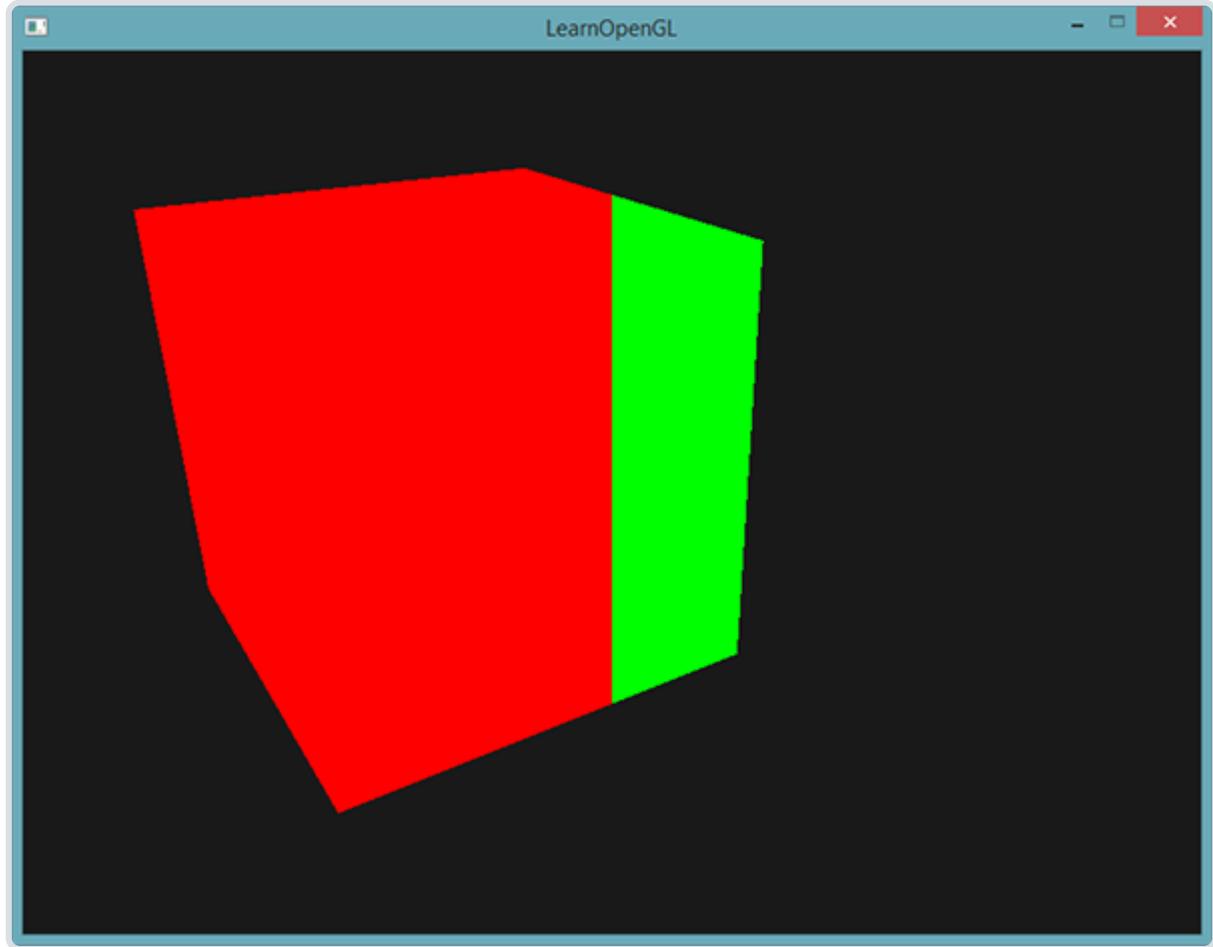
在讨论深度测试的时候，我们已经见过`gl_FragCoord`很多次了，因为`gl_FragCoord`的z分量等于对应片段的深度值。然而，我们也能使用它的x和y分量来实现一些有趣的效果。

`gl_FragCoord`的x和y分量是片段的窗口空间(Window-space)坐标，其原点为窗口的左下角。我们已经使用`glViewport`设定了一个800x600的窗口了，所以片段窗口空间坐标的x分量将在0到800之间，y分量在0到600之间。

通过利用片段着色器，我们可以根据片段的窗口坐标，计算出不同的颜色。`gl_FragCoord`的一个常见用处是用于对比不同片段计算的视觉输出效果，这在技术演示中可以经常看到。比如说，我们能够将屏幕分成两部分，在窗口的左侧渲染一种输出，在窗口的右侧渲染另一种输出。下面这个例子片段着色器会根据窗口坐标输出不同的颜色：

```
void main()
{
    if(gl_FragCoord.x < 400)
        FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    else
        FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

因为窗口的宽度是800。当一个像素的x坐标小于400时，它一定在窗口的左侧，所以我们给它一个不同的颜色。



我们现在会计算出两个完全不同的片段着色器结果，并将它们显示在窗口的两侧。举例来说，你可以将它用于测试不同的光照技巧。

gl_FrontFacing

片段着色器另外一个很有意思的输入变量是`gl_FrontFacing`。在[面剔除](#)教程中，我们提到OpenGL能够根据顶点的环绕顺序来决定一个面是正向还是背向面。如果我们不（启用`GL_FACE_CULL`来）使用面剔除，那么`gl_FrontFacing`将会告诉我们当前片段是属于正向面的一部分还是背向面的一部分。举例来说，我们能够对正向面计算出不同的颜色。

`gl_FrontFacing`变量是一个`bool`，如果当前片段是正向面的一部分那么就是`true`，否则就是`false`。比如说，我们可以这样子创建一个立方体，在内部和外部使用不同的纹理：

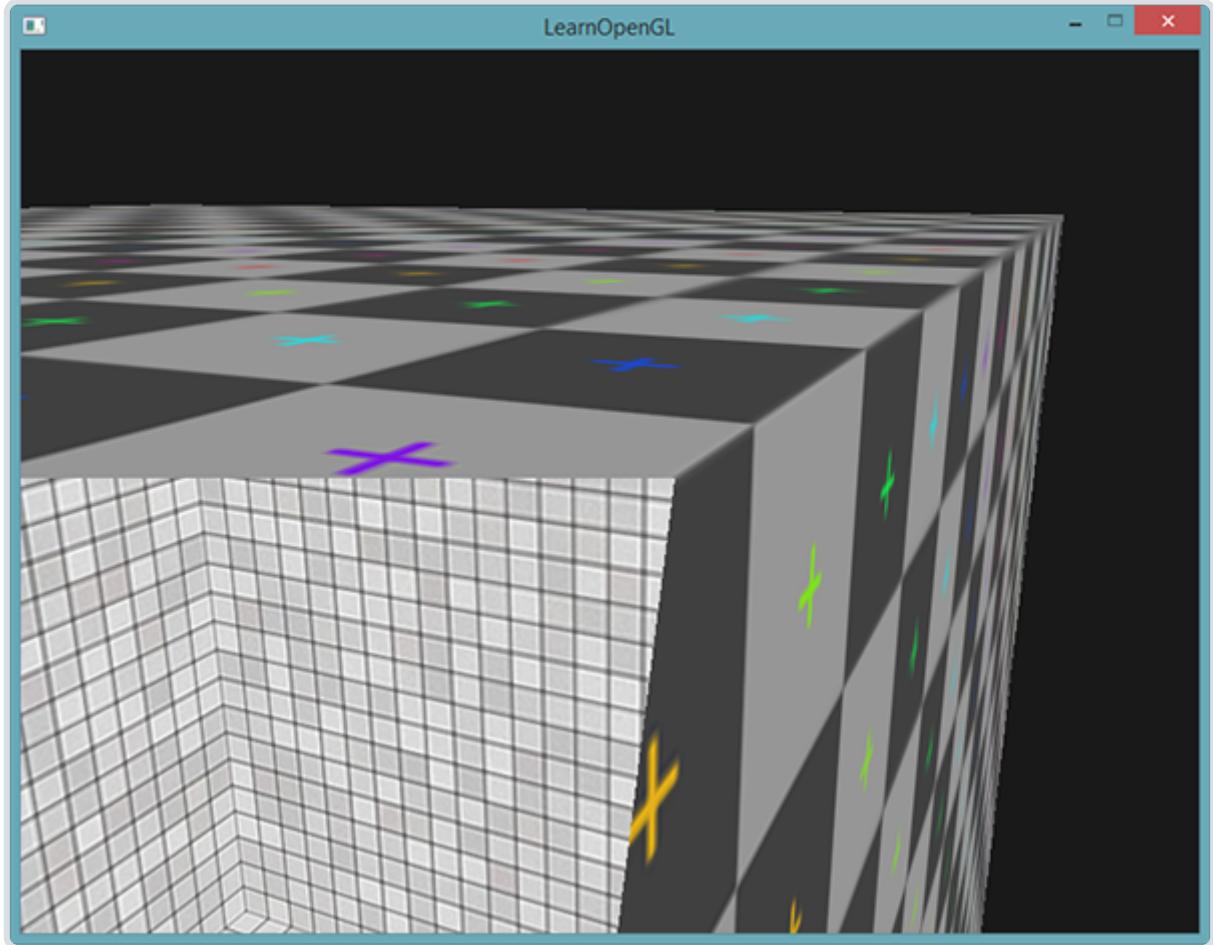
```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D frontTexture;
uniform sampler2D backTexture;

void main()
{
    if(gl_FrontFacing)
        FragColor = texture(frontTexture, TexCoords);
    else
        FragColor = texture(backTexture, TexCoords);
}
```

如果我们往箱子里面看，就能看到使用的是不同的纹理。



注意，如果你开启了面剔除，你就看不到箱子内部的面了，所以现在再使用`gl_FrontFacing`就没有意义了。

gl_FragDepth

输入变量`gl_FragCoord`能让我们读取当前片段的窗口空间坐标，并获取它的深度值，但是它是一个只读(Read-only)变量。我们不能修改片段的窗口空间坐标，但实际上修改片段的深度值还是可能的。GLSL提供给我们一个叫做`gl_FragDepth`的输出变量，我们可以使用它来在着色器内设置片段的深度值。

要想设置深度值，我们直接写入一个0.0到1.0之间的`float`值到输出变量就可以了：

```
gl_FragDepth = 0.0; // 这个片段现在的深度值为 0.0
```

如果着色器没有写入值到`gl_FragDepth`，它会自动取用`gl_FragCoord.z` 的值。

然而，由我们自己设置深度值有一个很大的缺点，只要我们在片段着色器中对`gl_FragDepth`进行写入，OpenGL就会（像[深度测试](#)小节中讨论的那样）禁用所有的提前深度测试(Early Depth Testing)。它被禁用的原因是，OpenGL无法在片段着色器运行之前得知片段将拥有的深度值，因为片段着色器可能会完全修改这个深度值。

在写入`gl_FragDepth`时，你就需要考虑到它所带来的性能影响。然而，从OpenGL 4.2起，我们仍可以对两者进行一定的调和，在片段着色器的顶部使用深度条件(Depth Condition)重新声明`gl_FragDepth`变量：

```
layout (depth_<condition>) out float gl_FragDepth;
```

`condition` 可以为下面的值：

条件	描述
<code>any</code>	默认值。提前深度测试是禁用的，你会损失很多性能
<code>greater</code>	你只能让深度值比 <code>gl_FragCoord.z</code> 更大
<code>less</code>	你只能让深度值比 <code>gl_FragCoord.z</code> 更小
<code>unchanged</code>	如果你要写入 <code>gl_FragDepth</code> ，你将只能写入 <code>gl_FragCoord.z</code> 的值

通过将深度条件设置为`greater` 或者 `less`，OpenGL就能假设你只会写入比当前片段深度值更大或者更小的值了。这样子的话，当深度值比片段的深度值要小的时候，OpenGL仍是能够进行提前深度测试的。

下面这个例子中，我们对片段的深度值进行了递增，但仍然也保留了一些提前深度测试：

```
#version 420 core // 注意GLSL的版本!
out vec4 FragColor;
layout (depth_greater) out float gl_FragDepth;

void main()
{
    FragColor = vec4(1.0);
    gl_FragDepth = gl_FragCoord.z + 0.1;
}
```

注意这个特性只在OpenGL 4.2版本或以上才提供。

2.5.14 接口块

到目前为止，每当我们希望从顶点着色器向片段着色器发送数据时，我们都声明了几个对应的输入/输出变量。将它们一个一个声明是着色器间发送数据最简单的方式，但当程序变得更大时，你希望发送的可能就不只是几个变量了，它还可能包括数组和结构体。

为了帮助我们管理这些变量，GLSL为我们提供了一个叫做**接口块**(Interface Block)的东西，来方便我们组合这些变量。接口块的声明和**struct**的声明有点相像，不同的是，现在根据它是一个输入还是输出块(Block)，使用**in**或**out**关键字来定义的。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out VS_OUT
{
    vec2 TexCoords;
} vs_out;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    vs_out.TexCoords = aTexCoords;
}
```

这次我们声明了一个叫做**vs_out**的接口块，它打包了我们希望发送到下一个着色器中的所有输出变量。这只是一个很简单的例子，但你可以想象一下，它能够帮助你管理着色器的输入和输出。当我们希望将着色器的输入或输出打包为数组时，它也会非常有用，我们将在[下一节](#)讨论几何着色器(Geometry Shader)时见到。

之后，我们还需要在下一个着色器，即片段着色器，中定义一个输入接口块。**块名**(Block Name)应该是和着色器中一样的(**VS_OUT**)，但**实例名**(Instance Name)(顶点着色器中用的是**vs_out**)可以是随意的，但要避免使用误导性的名称，比如对实际上包含输入变量的接口块命名为**vs_out**。

```
#version 330 core
out vec4 FragColor;

in VS_OUT
{
    vec2 TexCoords;
} fs_in;

uniform sampler2D texture;

void main()
{
    FragColor = texture(texture, fs_in.TexCoords);
}
```

只要两个接口块的名字一样，它们对应的输入和输出将会匹配起来。这是帮助你管理代码的又一个有用特性，它在几何着色器这样穿插特定着色器阶段的场景下会很有用。

2.5.15 Uniform缓冲对象

我们已经使用OpenGL很长时间了，学会了一些很酷的技巧，但也遇到了一些很麻烦的地方。比如说，当使用多于一个的着色器时，尽管大部分的uniform变量都是相同的，我们还是需要不断地设置它们，所以为什么要这么麻烦地重复设置它们呢？

OpenGL为我们提供了一个叫做Uniform缓冲对象(Uniform Buffer Object)的工具，它允许我们定义一系列在多个着色器中相同的全局Uniform变量。当使用Uniform缓冲对象的时候，我们只需要设置相关的uniform一次。当然，我们仍需要手动设置每个着色器中不同的uniform。并且创建和配置Uniform缓冲对象会有一点繁琐。

因为Uniform缓冲对象仍是一个缓冲，我们可以使用`glGenBuffers`来创建它，将它绑定到`GL_UNIFORM_BUFFER`缓冲目标，并将所有相关的uniform数据存入缓冲。在Uniform缓冲对象中储存数据是有一些规则的，我们将会在之后讨论它。首先，我们将使用一个简单的顶点着色器，将`projection`和`view`矩阵存储到所谓的Uniform块(Uniform Block)中：

```
#version 330 core
layout (location = 0) in vec3 aPos;

layout (std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};

uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

在我们大多数的例子中，我们都会在每个渲染迭代中，对每个着色器设置`projection`和`view`Uniform矩阵。这是利用Uniform缓冲对象的一个非常完美的例子，因为现在我们只需要存储这些矩阵一次就可以了。

这里，我们声明了一个叫做`Matrices`的Uniform块，它储存了两个4x4矩阵。Uniform块中的变量可以直接访问，不需要加块名作为前缀。接下来，我们在OpenGL代码中将这些矩阵值存入缓冲中，每个声明了这个Uniform块的着色器都能够访问这些矩阵。

你现在可能会想`layout (std140)`这个语句是什么意思。它的意思是说，当前定义的Uniform块对它的内容使用一个特定的内存布局。这个语句设置了Uniform块布局(Uniform Block Layout)。

Uniform块布局

Uniform块的内容是储存在一个缓冲对象中的，它实际上只是一块预留内存。因为这块内存并不会保存它具体保存的是什么类型的数据，我们还需要告诉OpenGL内存的哪一部分对应着着色器中的哪一个uniform变量。

假设着色器中有以下的这个Uniform块：

```
layout (std140) uniform ExampleBlock
{
    float value;
    vec3 vector;
    mat4 matrix;
    float values[3];
    bool boolean;
    int integer;
};
```

我们需要知道的是每个变量的大小（字节）和（从块起始位置的）偏移量，来让我们能够按顺序将它们放进缓冲中。每个元素的大小都是在OpenGL中有清楚地声明的，而且直接对应C++数据类型，其中向量和矩阵都是大的float数组。OpenGL没有声明的是这些变量间的间距(Spacing)。这允许硬件能够在它认为合适的位置放置变量。比如说，一些硬件可能会将一个`vec3`放置在`float`边上。不是所有的硬件都能这样处理，可能会在附加这个`float`之前，先将`vec3`填充(Pad)为一个4个float的数组。这个特性本身很棒，但是会对我们造成麻烦。

默认情况下，GLSL会使用一个叫做共享(Shared)布局的Uniform内存布局，共享是因为一旦硬件定义了偏移量，它们在多个程序中是共享并一致的。使用共享布局时，GLSL是可以为了优化而对uniform变量的位置进行变动的，只要变量的顺序保持不变。因为我们无法知道每个uniform变量的偏移量，我们也就无法知道如何准确地填充我们的Uniform缓冲了。我们能够使用像是`glGetUniformIndices`这样的函数来查询这个信息，但这超出本节的范围了。

虽然共享布局给了我们很多节省空间的优化，但是我们需要查询每个uniform变量的偏移量，这会产生非常多的工作量。通常的做法是，不使用共享布局，而是使用std140布局。std140布局声明了每个变量的偏移量都是由一系列规则所决定的，这显式地声明了每个变量类型的内存布局。由于这是显式提及的，我们可以手动计算出每个变量的偏移量。

每个变量都有一个基准对齐量(Base Alignment)，它等于一个变量在Uniform块中所占据的空间（包括填充量(Padding)），这个基准对齐量是使用std140布局的规则计算出来的。接下来，对每个变量，我们再计算它的对齐偏移量(Aligned Offset)，它是一个变量从块起始位置的字节偏移量。一个变量的对齐字节偏移量必须等于基准对齐量的倍数。

布局规则的原文可以在OpenGL的Uniform缓冲规范[这里](#)找到，但我们将会在下面列出最常见的规则。GLSL中的每个变量，比如说`int`、`float`和`bool`，都被定义为4字节量。每4个字节将会用一个`N`来表示。

类型	布局规则
标量，比如 <code>int</code> 和 <code>bool</code>	每个标量的基准对齐量为N。
向量	2N或者4N。这意味着 <code>vec3</code> 的基准对齐量为4N。
标量或向量的数组	每个元素的基准对齐量与 <code>vec4</code> 的相同。
矩阵	储存为列向量的数组，每个向量的基准对齐量与 <code>vec4</code> 的相同。
结构体	等于所有元素根据规则计算后的大小，但会填充到 <code>vec4</code> 大小的倍数。

和OpenGL大多数的规范一样，使用例子就能更容易地理解。我们会使用之前引入的那个叫做`ExampleBlock`的Uniform块，并使用std140布局计算出每个成员的对齐偏移量：

```
layout (std140) uniform ExampleBlock
{
    // 基准对齐量          // 对齐偏移量
    float value;        // 4           // 0
    vec3 vector;        // 16          // 16 (必须是16的倍数，所以 4->16)
    mat4 matrix;        // 16          // 32 (列 0)
                    // 16          // 48 (列 1)
                    // 16          // 64 (列 2)
                    // 16          // 80 (列 3)
    float values[3];   // 16          // 96 (values[0])
                    // 16          // 112 (values[1])
                    // 16          // 128 (values[2])
    bool boolean;       // 4           // 144
    int integer;        // 4           // 148
};
```

作为练习，尝试去自己计算一下偏移量，并和表格进行对比。使用计算后的偏移量值，根据std140布局的规则，我们就能使用像是`glBufferSubData`的函数将变量数据按照偏移量填充进缓冲中了。虽然std140布局不是最高效的布局，但它保证了内存布局在每个声明了这个Uniform块的程序中是一致的。

通过在Uniform块定义之前添加 `layout (std140)` 语句，我们告诉OpenGL这个Uniform块使用的是std140布局。除此之外还可以选择两个布局，但它们都需要我们在填充缓冲之前先查询每个偏移量。我们已经见过 `shared` 布局了，剩下的一个布局是 `packed`。当使用紧凑(Packed)布局时，是不能保证这个布局在每个程序中保持不变的（即非共享），因为它允许编译器去将uniform变量从Uniform块中优化掉，这在每个着色器中都可能是不同的。

使用Uniform缓冲

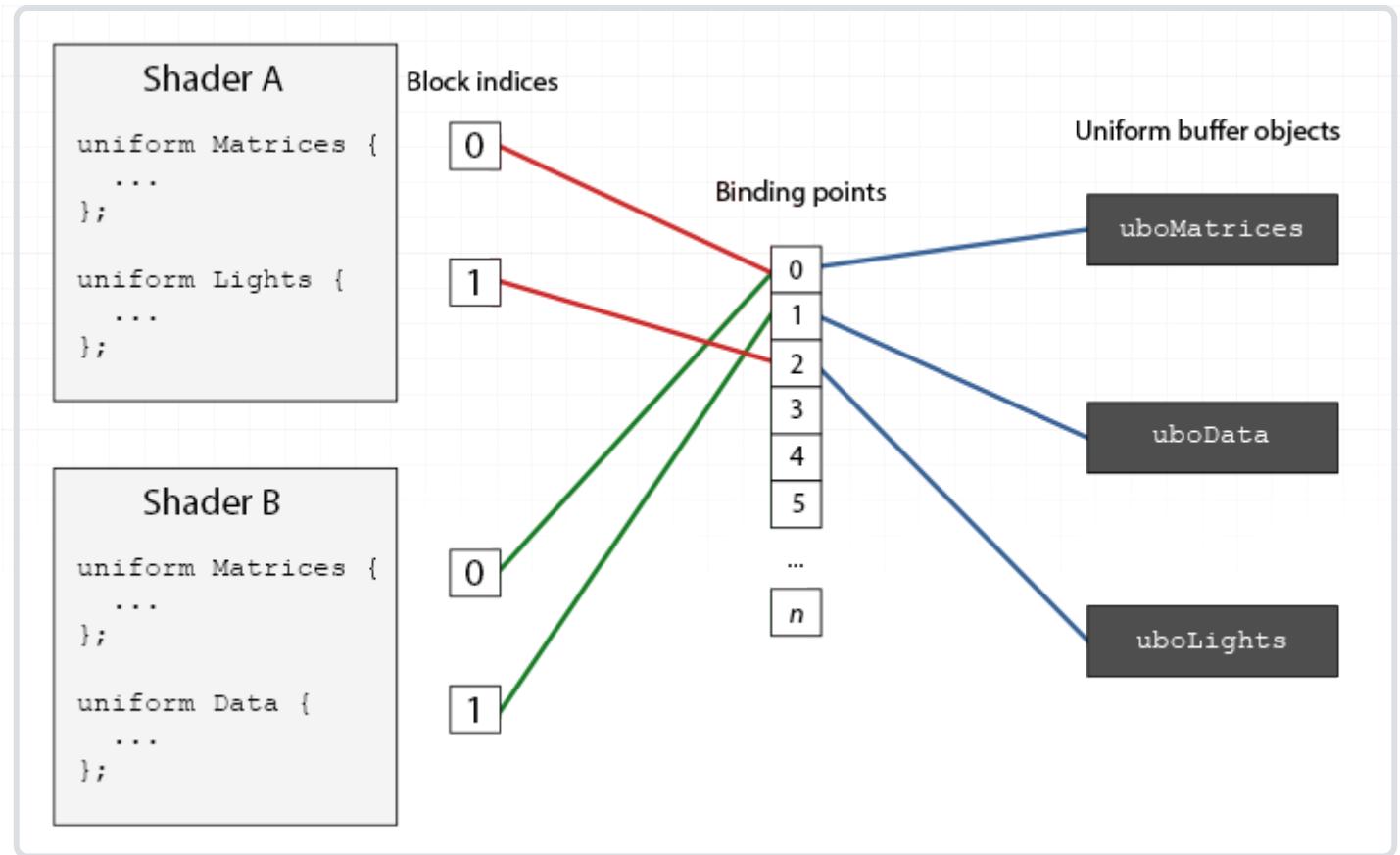
我们已经讨论了如何在着色器中定义Uniform块，并设定它们的内存布局了，但我们还没有讨论该如何使用它们。

首先，我们需要调用 `glGenBuffers`，创建一个Uniform缓冲对象。一旦我们有了一个缓冲对象，我们需要将它绑定到 `GL_UNIFORM_BUFFER` 目标，并调用 `glBufferData`，分配足够的内存。

```
unsigned int uboExampleBlock;
glGenBuffers(1, &uboExampleBlock);
 glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
 glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW); // 分配152字节的内存
 glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

现在，每当我们需要对缓冲更新或者插入数据，我们都会绑定到 `uboExampleBlock`，并使用 `glBufferSubData` 来更新它的内存。我们只需要更新这个Uniform缓冲一次，所有使用这个缓冲的着色器就都使用的是更新后的数据了。但是，如何才能让OpenGL知道哪个Uniform缓冲对应的是哪个Uniform块呢？

在OpenGL上下文中，定义了一些 **绑定点**(Binding Point)，我们可以将一个Uniform缓冲链接至它。在创建Uniform缓冲之后，我们将它绑定到其中一个绑定点上，并将着色器中的Uniform块绑定到相同的绑定点，把它们连接到一起。下面的这个图示展示了这个：



你可以看到，我们可以绑定多个Uniform缓冲到不同的绑定点上。因为着色器A和着色器B都有一个链接到绑定点0的Uniform块，它们的Uniform块将会共享相同的uniform数据，`uboMatrices`，前提条件是两个着色器都定义了相同的`Matrices` Uniform块。

为了将Uniform块绑定到一个特定的绑定点中，我们需要调用`glUniformBlockBinding`函数，它的第一个参数是一个程序对象，之后是一个Uniform块索引和链接到的绑定点。**Uniform块索引**(Uniform Block Index)是着色器中已定义Uniform块的位置值索引。这可以通过调用`glGetUniformBlockIndex`来获取，它接受一个程序对象和Uniform块的名称。我们可以用以下方式将图示中的`Lights` Uniform块链接到绑定点2：

```
unsigned int lights_index = glGetUniformLocation(shaderA.ID, "Lights");
glUniformBlockBinding(shaderA.ID, lights_index, 2);
```

注意我们需要对每个着色器重复这一步骤。

从OpenGL 4.2版本起，你也可以添加一个布局标识符，显式地将Uniform块的绑定点储存在着色器中，这样就不用再调用`glGetUniformBlockIndex`和`glUniformBlockBinding`了。下面的代码显式地设置了`Lights` Uniform块的绑定点。

```
layout(std140, binding = 2) uniform Lights { ... };
```

接下来，我们还需要绑定Uniform缓冲对象到相同的绑定点上，这可以使用`glBindBufferBase`或`glBindBufferRange`来完成。

```
glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboExampleBlock);
// 或
glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBlock, 0, 152);
```

`glBindBufferBase`需要一个目标，一个绑定点索引和一个Uniform缓冲对象作为它的参数。这个函数将`uboExampleBlock`链接到绑定点2上，自此，绑定点的两端都链接上了。你也可以使用`glBindBufferRange`函数，它需要一个附加的偏移量和大小参数，这样子你可以绑定Uniform缓冲的特定一部分到绑定点中。通过使用`glBindBufferRange`函数，你可以让多个不同的Uniform块绑定到同一个Uniform缓冲对象上。

现在，所有的东西都配置完毕了，我们可以开始向Uniform缓冲中添加数据了。只要我们需要，就可以使用`glBufferSubData`函数，用一个字节数组添加所有的数据，或者更新缓冲的一部分。要想更新uniform变量`boolean`，我们可以用以下方式更新Uniform缓冲对象：

```
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
int b = true; // GLSL中的bool是4字节的，所以我们将它存为一个integer
glBufferSubData(GL_UNIFORM_BUFFER, 144, 4, &b);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

同样的步骤也能应用到Uniform块中其它的uniform变量上，但需要使用不同的范围参数。

一个简单的例子

所以，我们来展示一个真正使用Uniform缓冲对象的例子。如果我们回头看看之前所有的代码例子，我们不断地在使用3个矩阵：投影、观察和模型矩阵。在所有的这些矩阵中，只有模型矩阵会频繁变动。如果我们有多个着色器使用了这一组矩阵，那么使用Uniform缓冲对象可能会更好。

我们会将投影和模型矩阵存储到一个叫做`Matrices`的Uniform块中。我们不会将模型矩阵存在这里，因为模型矩阵在不同的着色器中会不断改变，所以使用Uniform缓冲对象并不会带来什么好处。

```
#version 330 core
layout (location = 0) in vec3 aPos;

layout (std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

这里没什么特别的，除了我们现在使用的是一个std140布局的Uniform块。我们将在例子程序中，显示4个立方体，每个立方体都是使用不同的着色器程序渲染的。这4个着色器程序将使用相同的顶点着色器，但使用的是不同的片段着色器，每个着色器会输出不同的颜色。

首先，我们将顶点着色器的Uniform块设置为绑定点0。注意我们需要对每个着色器都设置一遍。

```
unsigned int uniformBlockIndexRed    = glGetUniformBlockIndex(shaderRed.ID, "Matrices");
unsigned int uniformBlockIndexGreen = glGetUniformBlockIndex(shaderGreen.ID, "Matrices");
unsigned int uniformBlockIndexBlue   = glGetUniformBlockIndex(shaderBlue.ID, "Matrices");
unsigned int uniformBlockIndexYellow = glGetUniformBlockIndex(shaderYellow.ID, "Matrices");

glUniformBlockBinding(shaderRed.ID, uniformBlockIndexRed, 0);
glUniformBlockBinding(shaderGreen.ID, uniformBlockIndexGreen, 0);
glUniformBlockBinding(shaderBlue.ID, uniformBlockIndexBlue, 0);
glUniformBlockBinding(shaderYellow.ID, uniformBlockIndexYellow, 0);
```

接下来，我们创建Uniform缓冲对象本身，并将其绑定到绑定点0：

```
unsigned int uboMatrices
 glGenBuffers(1, &uboMatrices);

 glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
 glBufferData(GL_UNIFORM_BUFFER, 2 * sizeof(glm::mat4), NULL, GL_STATIC_DRAW);
 glBindBuffer(GL_UNIFORM_BUFFER, 0);

 glBindBufferRange(GL_UNIFORM_BUFFER, 0, uboMatrices, 0, 2 * sizeof(glm::mat4));
```

首先我们为缓冲分配了足够的内存，它等于`glm::mat4`大小的两倍。GLM矩阵类型的大小直接对应于GLSL中的`mat4`。接下来，我们将缓冲中的特定范围（在这里是整个缓冲）链接到绑定点0。

剩余的就是填充这个缓冲了。如果我们将投影矩阵的视野(Field of View)值保持不变（所以摄像机就没有缩放了），我们只需要将其在程序中定义一次——这也意味着我们只需要将它插入到缓冲中一次。因为我们已经为缓冲对象分配了足够的内存，我们可以使用`glBufferSubData`在进入渲染循环之前存储投影矩阵：

```
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4), glm::value_ptr(projection));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

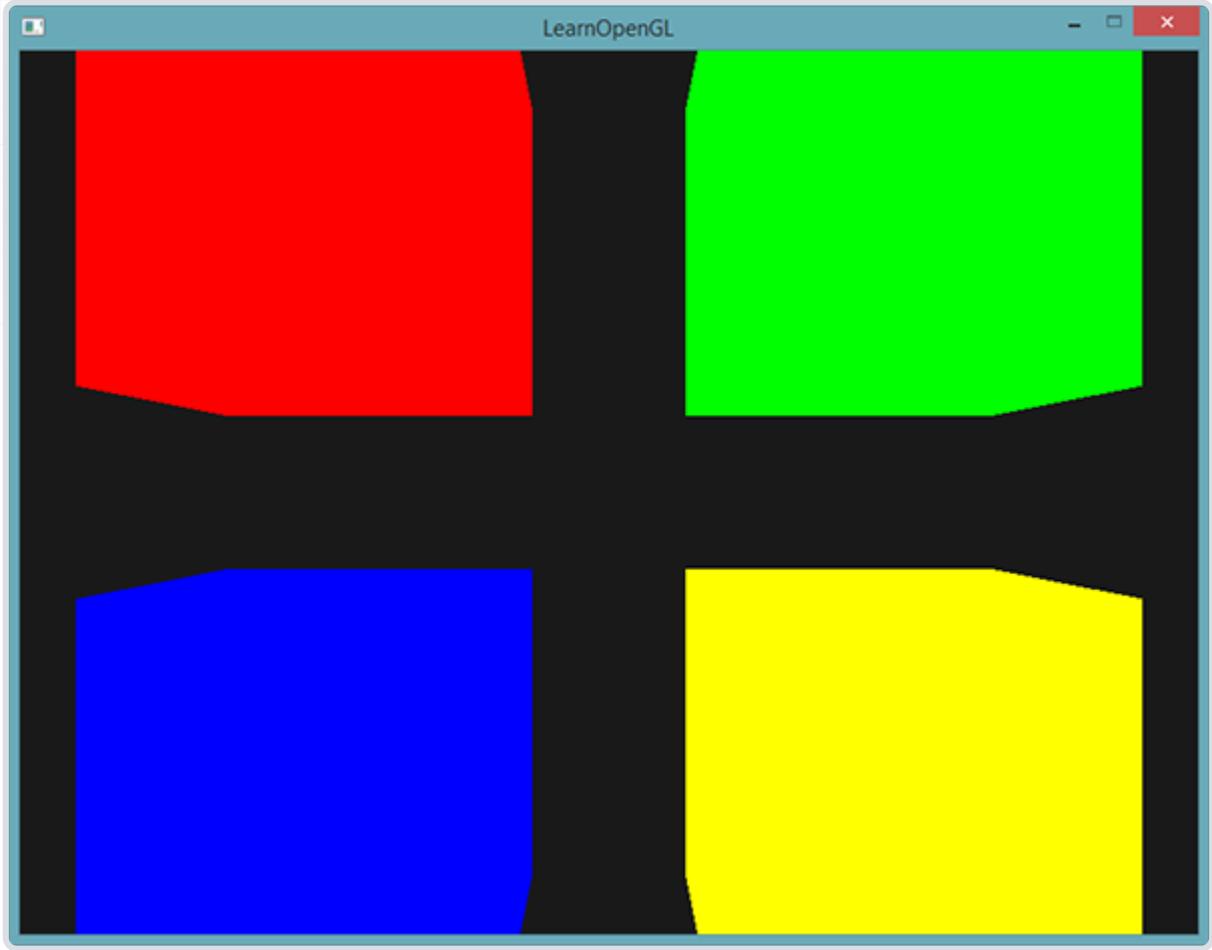
这里我们将投影矩阵储存在Uniform缓冲的前半部分。在每次渲染迭代中绘制物体之前，我们会将观察矩阵更新到缓冲的后半部分：

```
glm::mat4 view = camera.GetViewMatrix();
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(glm::mat4), sizeof(glm::mat4), glm::value_ptr(view));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Uniform缓冲对象的部分就结束了。每个包含了`Matrices`这个Uniform块的顶点着色器将会包含储存在`uboMatrices`中的数据。所以，如果我们现在要用4个不同的着色器绘制4个立方体，它们的投影和观察矩阵都会是一样的。

```
glBindVertexArray(cubeVAO);
shaderRed.use();
glm::mat4 model;
model = glm::translate(model, glm::vec3(-0.75f, 0.75f, 0.0f)); // 移动到左上角
shaderRed.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
// ... 绘制绿色立方体
// ... 绘制蓝色立方体
// ... 绘制黄色立方体
```

唯一需要设置的uniform只剩`model` uniform了。在像这样的场景中使用Uniform缓冲对象会让我们在每个着色器中都剩下一些uniform调用。最终的结果会是这样的：



因为修改了模型矩阵，每个立方体都移动到了窗口的一边，并且由于使用了不同的片段着色器，它们的颜色也不同。这只是一个很简单的情景，我们可能会需要使用Uniform缓冲对象，但任何大型的渲染程序都可能同时激活有上百个着色器程序，这时候Uniform缓冲对象的优势就会很大地体现出来了。

你可以在[这里](#)找到uniform例子程序的完整源代码。

Uniform缓冲对象比起独立的uniform有很多好处。第一，一次设置很多uniform会比一个一个设置多个uniform要快很多。第二，比起在多个着色器中修改同样的uniform，在Uniform缓冲中修改一次会更容易一些。最后一个好处可能不会立即显现，如果使用Uniform缓冲对象的话，你可以在着色器中使用更多的uniform。OpenGL限制了它能够处理的uniform数量，这可以通过`GL_MAX_VERTEX_UNIFORM_COMPONENTS`来查询。当使用Uniform缓冲对象时，最大的数量会更高。所以，当你达到了uniform的最大数量时（比如再做骨骼动画(Skeletal Animation)的时候），你总是可以选择使用Uniform缓冲对象。

2.5.16 几何着色器

原文 [Geometry Shader](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

在顶点和片段着色器之间有一个可选的 **几何着色器**(Geometry Shader)，几何着色器的输入是一个图元（如点或三角形）的一组顶点。几何着色器可以在顶点发送到下一着色器阶段之前对它们随意变换。然而，几何着色器最有趣的地方在于，它能够将（这一组）顶点变换为完全不同的图元，并且还能生成比原来更多的顶点。

废话不多说，我们直接先看一个几何着色器的例子：

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

在几何着色器的顶部，我们需要声明从顶点着色器输入的图元类型。这需要在 `in` 关键字前声明一个布局修饰符(Layout Qualifier)。这个输入布局修饰符可以从顶点着色器接收下列任何一个图元值：

- `points`：绘制`GL_POINTS`图元时 (1)。
- `lines`：绘制`GL_LINES`或`GL_LINE_STRIP`时 (2)
- `lines_adjacency`：`GL_LINES_ADJACENCY`或`GL_LINE_STRIP_ADJACENCY` (4)
- `triangles`：`GL_TRIANGLES`、`GL_TRIANGLE_STRIP`或`GL_TRIANGLE_FAN` (3)
- `triangles_adjacency`：`GL_TRIANGLES_ADJACENCY`或`GL_TRIANGLE_STRIP_ADJACENCY` (6)

以上是能提供给`glDrawArrays`渲染函数的几乎所有图元了。如果我们想要将顶点绘制为`GL_TRIANGLES`，我们就要将输入修饰符设置为`triangles`。括号内的数字表示的是一个图元所包含的最小顶点数。

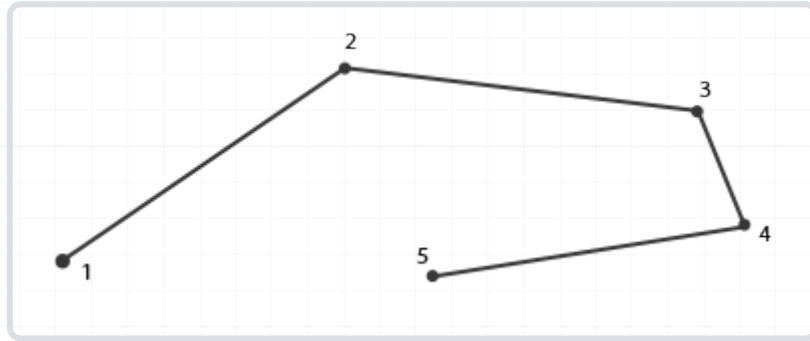
接下来，我们还需要指定几何着色器输出的图元类型，这需要在 `out` 关键字前面加一个布局修饰符。和输入布局修饰符一样，输出布局修饰符也可以接受几个图元值：

- `points`
- `line_strip`
- `triangle_strip`

有了这3个输出修饰符，我们就可以使用输入图元创建几乎任意的形状了。要生成一个三角形的话，我们将输出定义为`triangle_strip`，并输出3个顶点。

几何着色器同时希望我们设置一个它最大能够输出的顶点数量（如果你超过了这个值，OpenGL将不会绘制多出的顶点），这个也可以在 `out` 关键字的布局修饰符中设置。在这个例子中，我们将输出一个 `line_strip`，并将最大顶点数设置为2个。

如果你不知道什么是线条(Line Strip): 线条连接了一组点, 形成一条连续的线, 它最少要由两个点来组成。在渲染函数中每多加一个点, 就会在这个点与前一个点之间形成一条新的线。在下面这张图中, 我们有5个顶点:



如果使用的是上面定义的着色器, 那么这将只能输出一条线段, 因为最大顶点数等于2。

为了生成更有意义的结果, 我们需要某种方式来获取前一着色器阶段的输出。GLSL提供给我们一个**内建(Built-in)**变量, 在内部看起来(可能)是这样的:

```
in gl_Vertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

这里, 它被声明为一个**接口块** (Interface Block, 我们在[上一节](#)已经讨论过), 它包含了几个很有意思的变量, 其中最有趣的一个是**gl_Position**, 它是和顶点着色器输出非常相似的一个向量。

要注意的是, 它被声明为一个数组, 因为大多数的渲染图元包含多于1个的顶点, 而几何着色器的输入是一个图元的所有顶点。

有了之前顶点着色器阶段的顶点数据, 我们就可以使用2个几何着色器函数, **EmitVertex**和**EndPrimitive**, 来生成新的数据了。几何着色器希望你能够生成并输出至少一个定义为输出的图元。在我们的例子中, 我们需要至少生成一个线条图元。

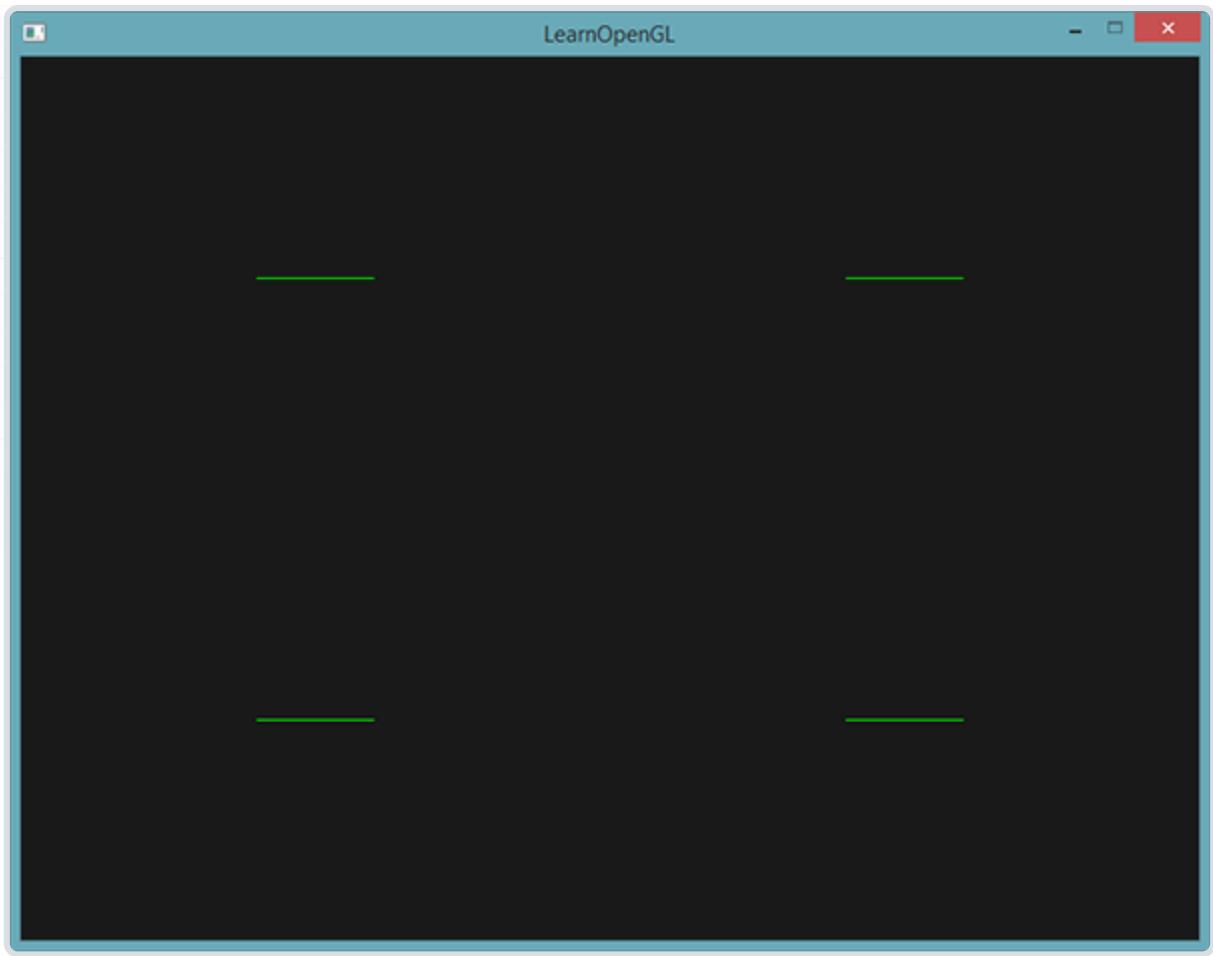
```
void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

每次我们调用**EmitVertex**时, **gl_Position**中的向量会被添加到图元中来。当**EndPrimitive**被调用时, 所有发射出的(Emitted)顶点都会合成为指定的输出渲染图元。在一个或多个**EmitVertex**调用之后重复调用**EndPrimitive**能够生成多个图元。在这个例子中, 我们发射了两个顶点, 它们从原始顶点位置平移了一段距离, 之后调用了**EndPrimitive**, 将这两个顶点合成为一个包含两个顶点的线条。

现在你(大概)了解了几何着色器的工作方式, 你可能已经猜出这个几何着色器是做什么的了。它接受一个点图元作为输入, 以这个点为中心, 创建一条水平的线图元。如果我们渲染它, 看起来会是这样的:



目前还并没有什么令人惊叹的效果，但考虑到这个输出是通过调用下面的渲染函数来生成的，它还是很有意思的：

```
glDrawArrays(GL_POINTS, 0, 4);
```

虽然这是一个比较简单的例子，它的确向你展示了如何能够使用几何着色器来（动态地）生成新的形状。在之后我们会利用几何着色器创建出更有意思的效果，但现在我们仍将从创建一个简单的几何着色器开始。

使用几何着色器

为了展示几何着色器的用法，我们将会渲染一个非常简单的场景，我们只会在标准化设备坐标的z平面上绘制四个点。这些点的坐标是：

```
float points[] = {
    -0.5f,  0.5f, // 左上
    0.5f,  0.5f, // 右上
    0.5f, -0.5f, // 右下
   -0.5f, -0.5f // 左下
};
```

顶点着色器只需要在z平面绘制点就可以了，所以我们将使用一个最基本顶点着色器：

```
#version 330 core
layout (location = 0) in vec2 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

直接在片段着色器中硬编码，将所有的点都输出为绿色：

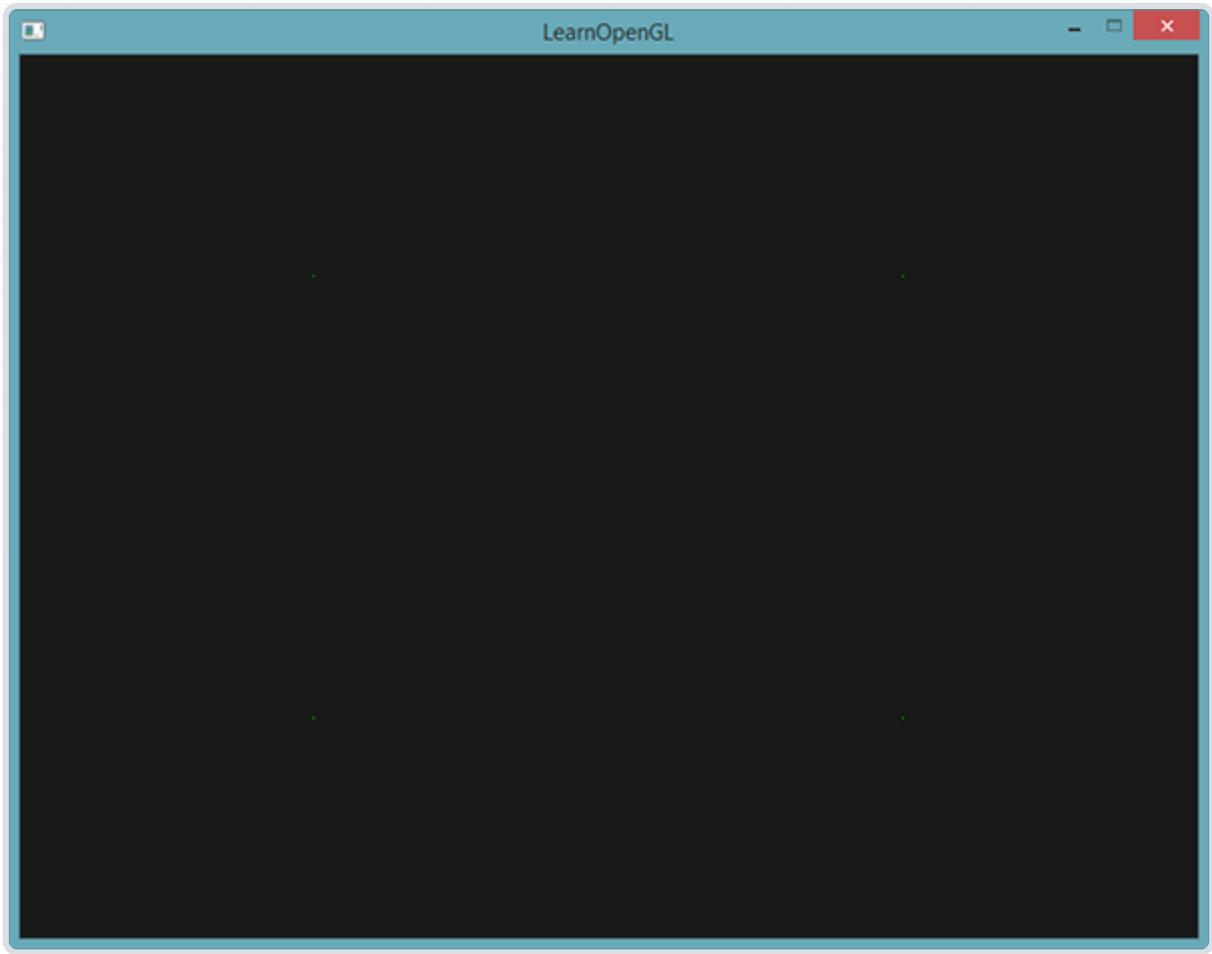
```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

为点的顶点数据生成一个VAO和一个VBO，然后使用`glDrawArrays`进行绘制：

```
shader.use();
 glBindVertexArray(VAO);
 glDrawArrays(GL_POINTS, 0, 4);
```

结果是在黑暗的场景中有四个（很难看见的）绿点：



但我们之前不是学过这些吗？是的，但是现在我们将会添加一个几何着色器，为场景添加活力。

出于学习目的，我们将会创建一个传递(Pass-through)几何着色器，它会接收一个点图元，并直接将它传递(Pass)到下一个着色器：

```
#version 330 core
layout (points) in;
layout (points, max_vertices = 1) out;

void main() {
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();
    EndPrimitive();
}
```

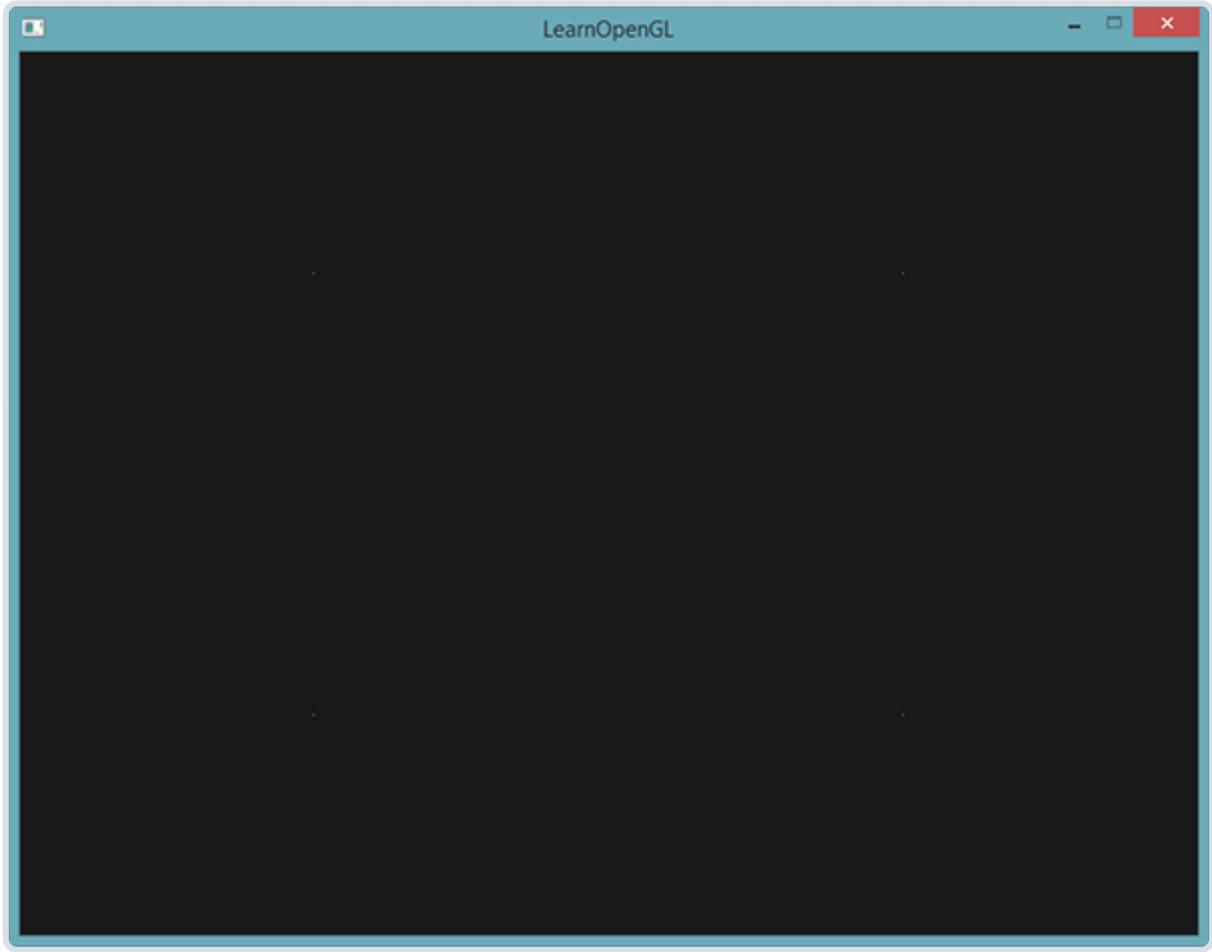
现在这个几何着色器应该很容易理解了，它只是将它接收到的顶点位置不作修改直接发射出去，并生成一个点图元。

和顶点与片段着色器一样，几何着色器也需要编译和链接，但这次在创建着色器时我们将会使用`GL_GEOMETRY_SHADER`作为着色器类型：

```
geometryShader = glCreateShader(GL_GEOMETRY_SHADER);
glShaderSource(geometryShader, 1, &gShaderCode, NULL);
glCompileShader(geometryShader);
...
glAttachShader(program, geometryShader);
glLinkProgram(program);
```

着色器编译的代码和顶点与片段着色器代码都是一样的。记得要检查编译和链接错误！

如果你现在编译并运行程序，会看到和下面类似的结果：

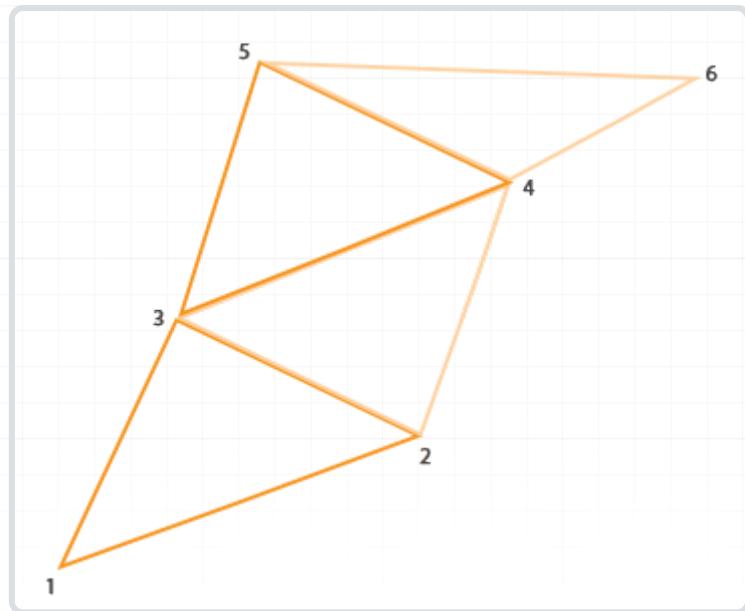


这和没使用几何着色器时是完全一样的！我承认这是有点无聊，但既然我们仍然能够绘制这些点，所以几何着色器是正常工作的，现在是时候做点更有趣的东西了！

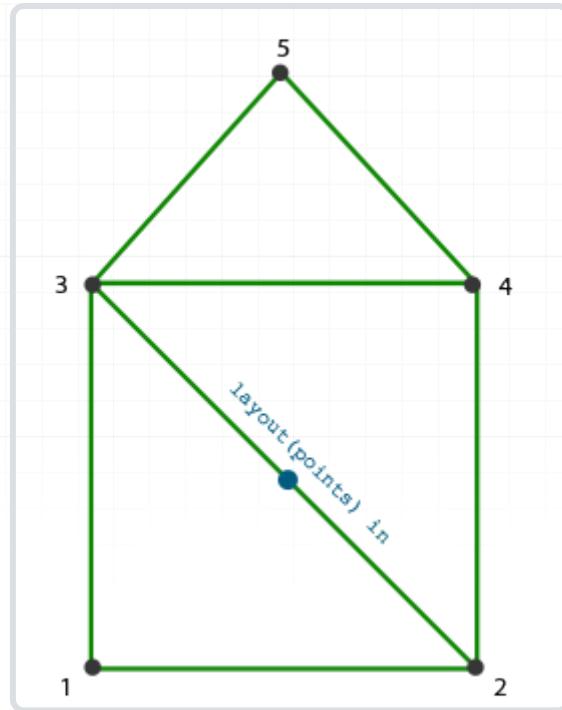
造几个房子

绘制点和线并没有那么有趣，所以我们会使用一点创造力，利用几何着色器在每个点的位置上绘制一个房子。要实现这个，我们可以将几何着色器的输出设置为`triangle_strip`，并绘制三个三角形：其中两个组成一个正方形，另一个用作房顶。

OpenGL中，三角形带(Triangle Strip)是绘制三角形更高效的方式，它使用顶点更少。在第一个三角形绘制完之后，每个后续顶点将会在上一个三角形边上生成另一个三角形：每3个临近的顶点将会形成一个三角形。如果我们一共有6个构成三角形带的顶点，那么我们会得到这些三角形：(1, 2, 3)、(2, 3, 4)、(3, 4, 5)和(4, 5, 6)，共形成4个三角形。一个三角形带至少需要3个顶点，并会生成N-2个三角形。使用6个顶点，我们创建了 $6-2 = 4$ 个三角形。下面这幅图展示了这点：



通过使用三角形带作为几何着色器的输出，我们可以很容易创建出需要的房子形状，只需要以正确的顺序生成3个相连的三角形就行了。下面这幅图展示了顶点绘制的顺序，蓝点代表的是输入点：



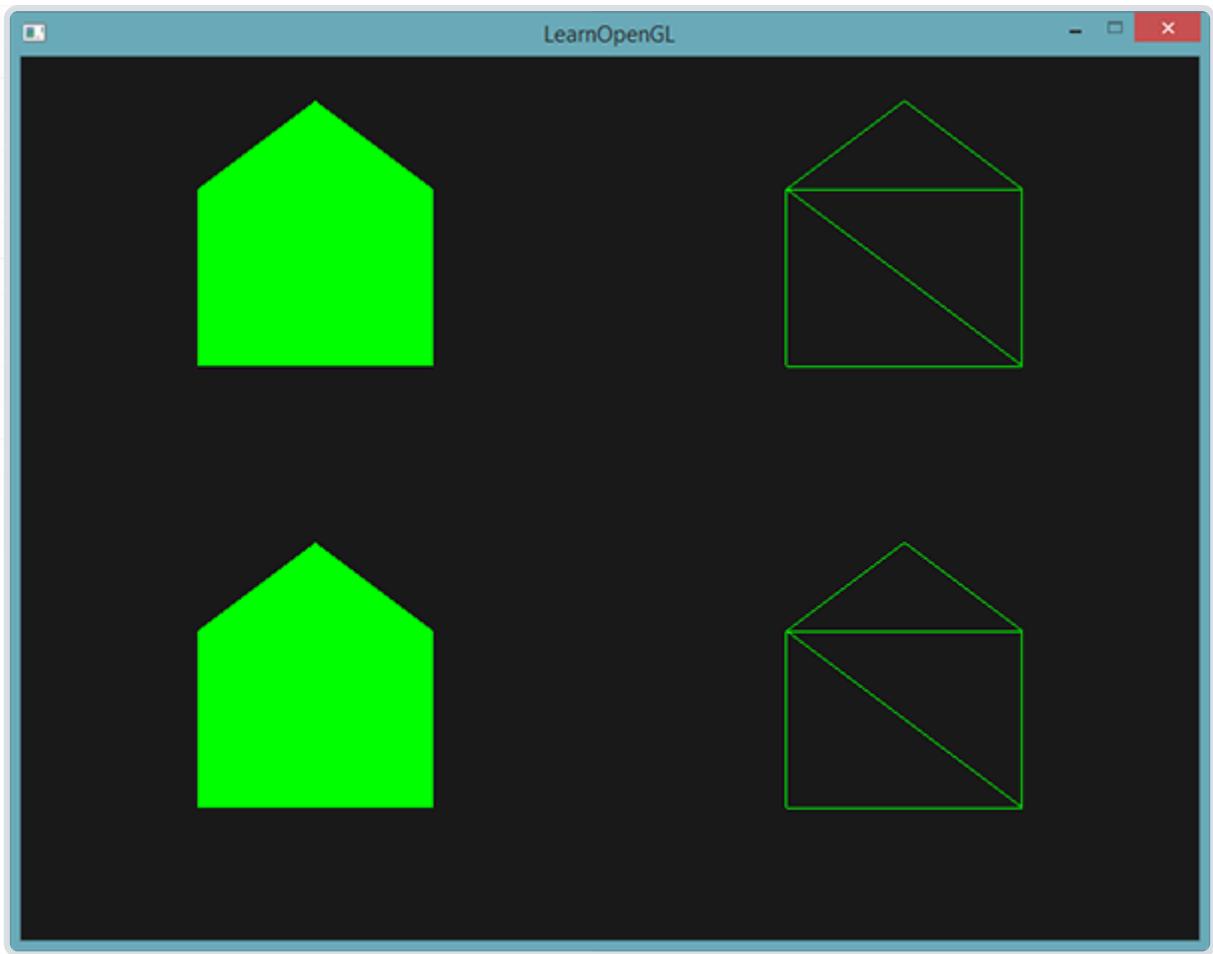
变为几何着色器是这样的：

```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position)
{
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:左下
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:右下
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:左上
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:右上
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:顶部
    EmitVertex();
    EndPrimitive();
}

void main() {
    build_house(gl_in[0].gl_Position);
}
```

这个几何着色器生成了5个顶点，每个顶点都是原始点的位置加上一个偏移量，来组成一个大的三角形带。最终的图元会被光栅化，然后片段着色器会处理整个三角形带，最终在每个绘制的点处生成一个绿色房子：



你可以看到，每个房子实际上是由3个三角形组成的——全部都是使用空间中一点来绘制的。这些绿房子看起来是有点无聊，所以我们会再给每个房子分配一个不同的颜色。为了实现这个，我们需要在顶点着色器中添加一个额外的顶点属性，表示颜色信息，将它传递至几何着色器，并再次发送到片段着色器中。

下面是更新后的顶点数据：

```
float points[] = {
    -0.5f,  0.5f, 1.0f, 0.0f, 0.0f, // 左上
     0.5f,  0.5f, 0.0f, 1.0f, 0.0f, // 右上
     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, // 右下
    -0.5f, -0.5f, 1.0f, 1.0f, 0.0f // 左下
};
```

然后我们更新顶点着色器，使用一个接口块将颜色属性发送到几何着色器中：

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out VS_OUT {
    vec3 color;
} vs_out;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    vs_out.color = aColor;
}
```

接下来我们还需要在几何着色器中声明相同的接口块（使用一个不同的接口名）：

```
in VS_OUT {
    vec3 color;
} gs_in[];
```

因为几何着色器是作用于输入的一组顶点的，从顶点着色器发来输入数据总是会以数组的形式表示出来，即便我们现在只有一个顶点。

我们并不是必须要用接口块来向几何着色器传递数据。如果顶点着色器发送的颜色向量是 `out vec3 vColor`，我们也可以这样写：

```
in vec3 vColor[];
```

然而，接口块在几何着色器这样的着色器中会更容易处理一点。实际上，几何着色器的输入能够变得非常大，将它们合并为一个大的接口块数组会更符合逻辑一点。

接下来我们还需要为下个片段着色器阶段声明一个输出颜色向量：

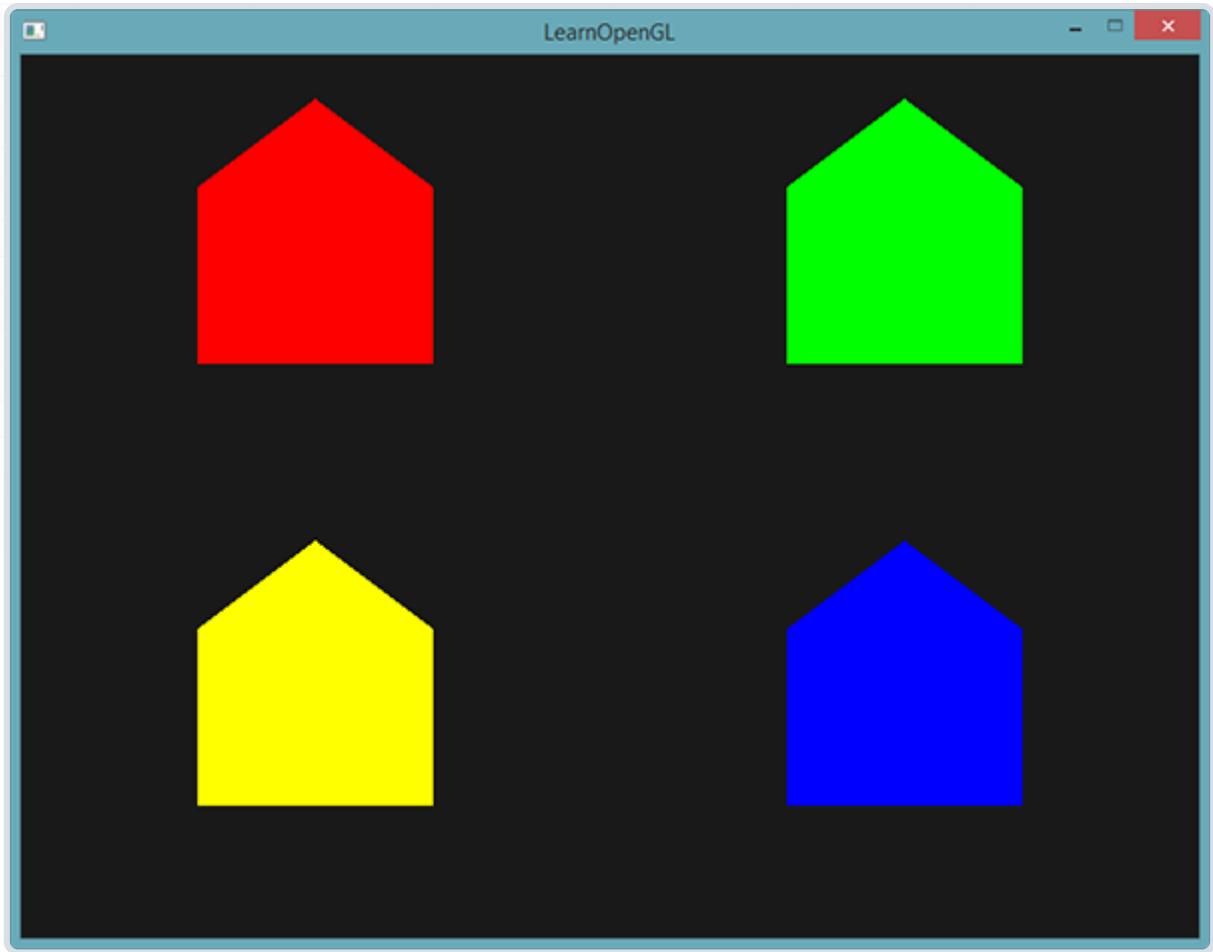
```
out vec3 fColor;
```

因为片段着色器只需要一个（插值的）颜色，发送多个颜色并没有什么意义。所以，`fColor`向量就不是一个数组，而是一个单独的向量。当发射一个顶点的时候，每个顶点将会使用最后在`fColor`中储存的值，来用于片段着色器的运行。对我们的房子来说，我们只需要在第一个顶点发射之前，使用顶点着色器中的颜色填充`fColor`一次就可以了。

因为片段着色器只需要一个（已进行了插值的）颜色，传送多个颜色没有意义。fColor向量这样就不是一个数组，而是一个单一的向量。当发射一个顶点时，为了它的片段着色器运行，每个顶点都会储存最后在fColor中储存的值。对于这些房子来说，我们可以在第一个顶点被发射，对整个房子上色前，只使用来自顶点着色器的颜色填充fColor一次：

```
fColor = gs_in[0].color; // gs_in[0] 因为只有一个输入顶点
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);      // 1:左下
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);      // 2:右下
EmitVertex();
gl_Position = position + vec4(-0.2, 0.2, 0.0, 0.0);      // 3:左上
EmitVertex();
gl_Position = position + vec4( 0.2, 0.2, 0.0, 0.0);      // 4:右上
EmitVertex();
gl_Position = position + vec4( 0.0, 0.4, 0.0, 0.0);      // 5:顶部
EmitVertex();
EndPrimitive();
```

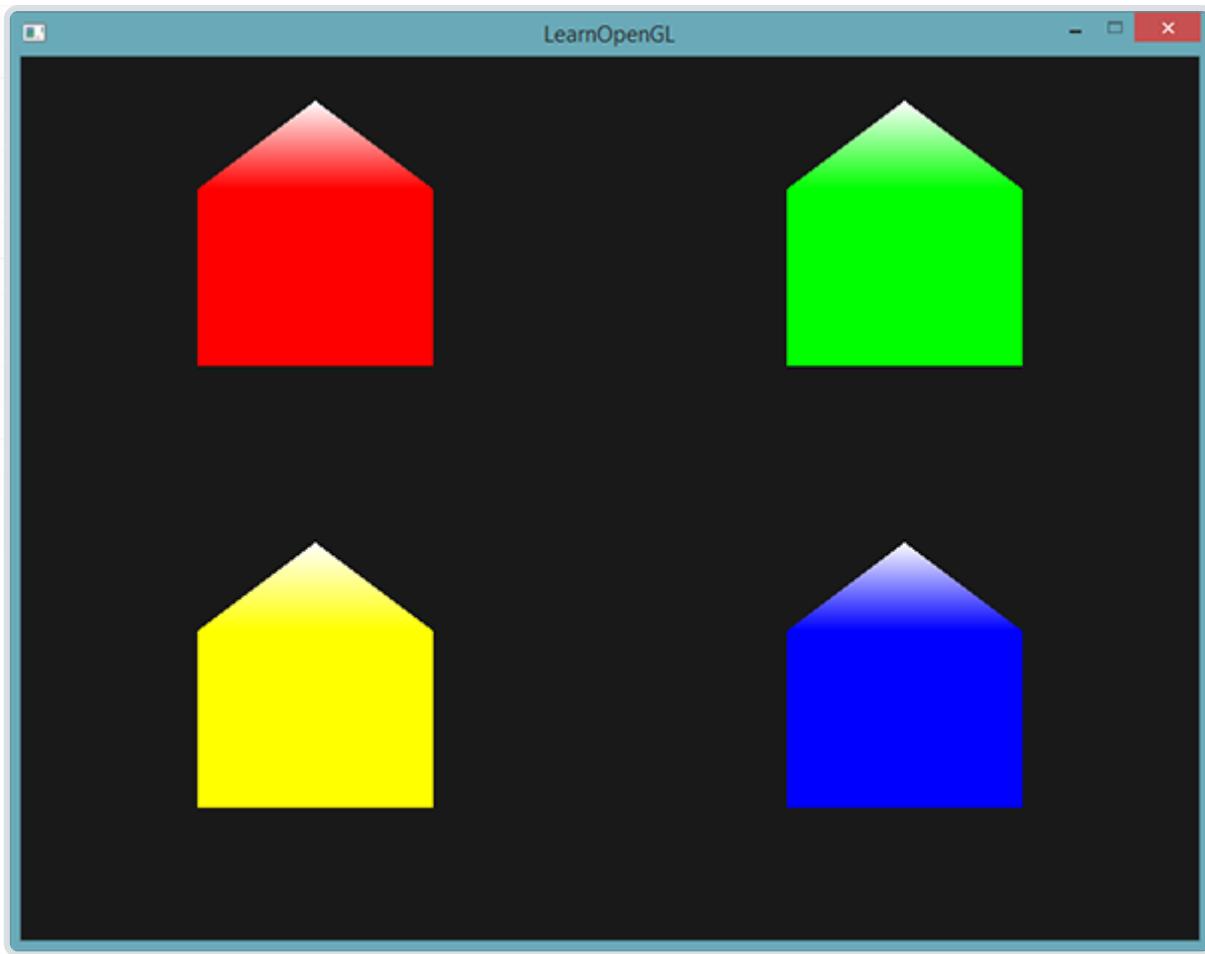
所有发射出的顶点都将嵌有最后储存在fColor中的值，即顶点的颜色属性值。所有的房子都会有它们自己的颜色了：



仅仅是为了有趣，我们也可以假装这是冬天，将最后一个顶点的颜色设置为白色，给屋顶落上一些雪。

```
fColor = gs_in[0].color;
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);      // 1:左下
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);      // 2:右下
EmitVertex();
gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);      // 3:左上
EmitVertex();
gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);      // 4:右上
EmitVertex();
gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);      // 5:顶部
fColor = vec3(1.0, 1.0, 1.0);
EmitVertex();
EndPrimitive();
```

最终结果看起来是这样的：



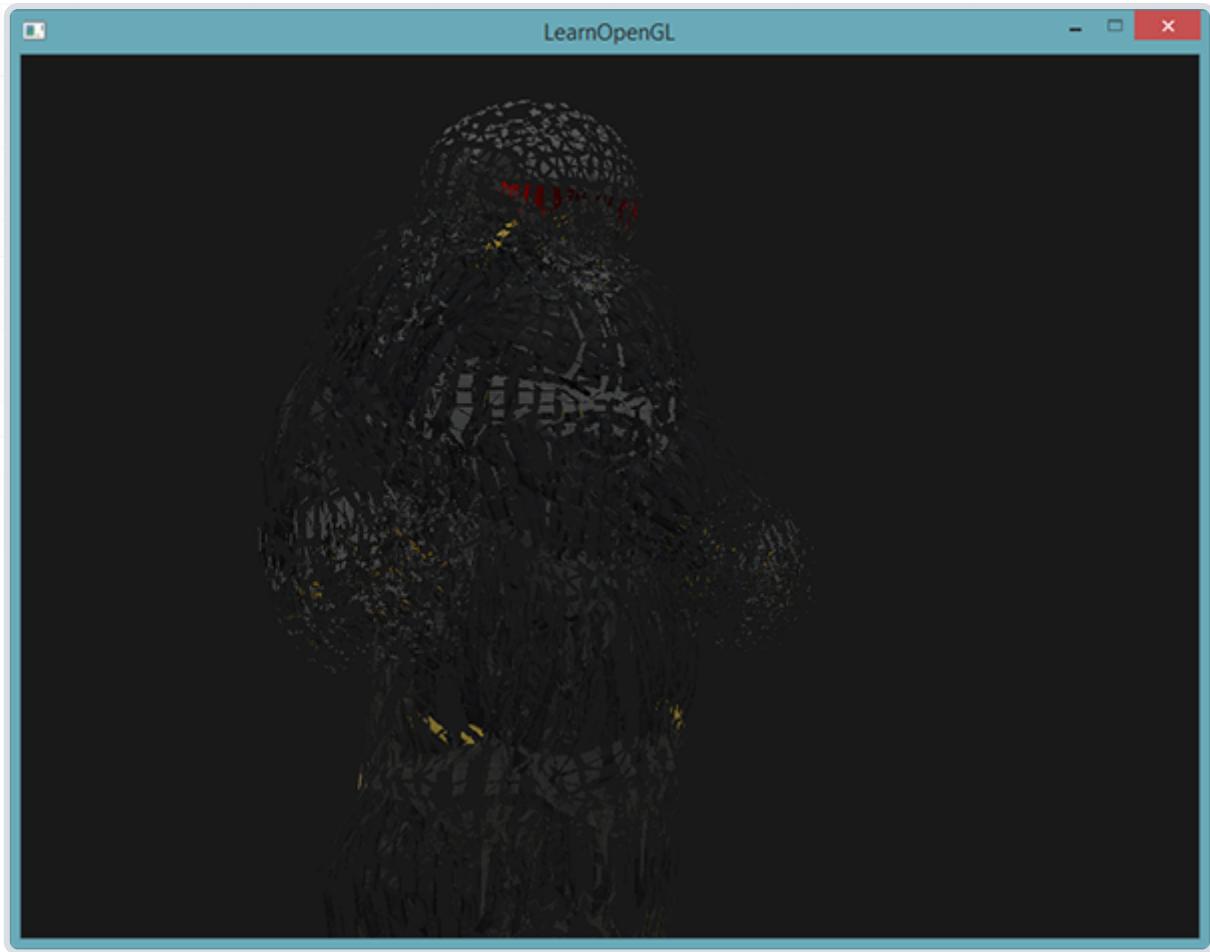
你可以将你的代码与[这里的](#)OpenGL代码进行比对。

你可以看到，有了几何着色器，你甚至可以将最简单的图元变得十分有创意。因为这些形状是在GPU的超快硬件中动态生成的，这会比在顶点缓冲中手动定义图形要高效很多。因此，几何缓冲对简单而且经常重复的形状来说是一个很好的优化工具，比如体素(Voxel)世界中的方块和室外草地的每一根草。

2.5.17 爆破物体

尽管绘制房子非常有趣，但我们不会经常这么做。这也是为什么我们接下来要继续深入，来爆破(Explode)物体！虽然这也是一个不怎么常用的东西，但是它能向你展示几何着色器的强大之处。

当我们说爆破一个物体时，我们并不是指要将宝贵的顶点集给炸掉，我们是要将每个三角形沿着法向量的方向移动一小段时间。效果就是，整个物体看起来像是沿着每个三角形的法线向量爆炸一样。爆炸三角形的效果在纳米级模型上看起来像是这样的：



这样的几何着色器效果的一个好处就是，无论物体有多复杂，它都能够应用上去。

因为我们想要沿着三角形的法向量位移每个顶点，我们首先需要计算这个法向量。我们所要做的是计算垂直于三角形表面的向量，仅使用我们能够访问的3个顶点。你可能还记得在[变换](#)小节中，我们使用[叉乘](#)来获取垂直于其它两个向量的一个向量。如果我们能够获取两个平行于三角形表面的向量`a`和`b`，我们就能够对这两个向量进行叉乘来获取法向量了。下面这个几何着色器函数做的正是这个，来使用3个输入顶点坐标来获取法向量：

```
vec3 GetNormal()
{
    vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);
    vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);
    return normalize(cross(a, b));
}
```

这里我们使用减法获取了两个平行于三角形表面的向量`a`和`b`。因为两个向量相减能够得到这两个向量之间的差值，并且三个点都位于三角平面上，对任意两个向量相减都能够得到一个平行于平面的向量。注意，如果我们交换了`cross`函数中`a`和`b`的位置，我们会得到一个指向相反方向的法向量——这里的顺序很重要！

既然知道了如何计算法向量了，我们就能够创建一个`explode`函数了，它使用法向量和顶点位置向量作为参数。这个函数会返回一个新的向量，它是位置向量沿着法线向量进行位移之后的结果：

```
vec4 explode(vec4 position, vec3 normal)
{
    float magnitude = 2.0;
    vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * magnitude;
    return position + vec4(direction, 0.0);
}
```

函数本身应该不是非常复杂。`sin`函数接收一个`time`参数，它根据时间返回一个-1.0到1.0之间的值。因为我们不想让物体向内爆炸(Implode)，我们将`sin`值变换到了[0, 1]的范围内。最终的结果会乘以`normal`向量，并且最终的`direction`向量会被加到位置向量上。

当使用我们的模型加载器绘制一个模型时，爆破(Explode)效果的完整几何着色器是这样的：

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in VS_OUT {
    vec2 texCoords;
} gs_in[];

out vec2 TexCoords;

uniform float time;

vec4 explode(vec4 position, vec3 normal) { ... }

vec3 GetNormal() { ... }

void main() {
    vec3 normal = GetNormal();

    gl_Position = explode(gl_in[0].gl_Position, normal);
    TexCoords = gs_in[0].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[1].gl_Position, normal);
    TexCoords = gs_in[1].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[2].gl_Position, normal);
    TexCoords = gs_in[2].texCoords;
    EmitVertex();
    EndPrimitive();
}
```

注意我们在发射顶点之前输出了对应的纹理坐标。

而且别忘了在OpenGL代码中设置`time`变量：

```
shader.SetFloat("time", glfwGetTime());
```

最终的效果是，3D模型看起来随着时间不断在爆破它的顶点，在这之后又回到正常状态。虽然这并不是非常有用，它的确向你展示了几何着色器更高级的用法。你可以将你的代码和[这里](#)完整的源码进行比较。

2.5.18 法向量可视化

在这一部分中，我们将使用几何着色器来实现一个真正有用的例子：显示任意物体的法向量。当编写光照着色器时，你可能会最终会得到一些奇怪的视觉输出，但又很难确定导致问题的原因。光照错误很常见的原因就是法向量错误，这可能是由于不正确加载顶点数据、错误地将它们定义为顶点属性或在着色器中不正确地管理所导致的。我们想要的是使用某种方式来检测提供的法向量是正确的。检测法向量是否正确的一个很好的方式就是对它们进行可视化，几何着色器正是实现这一目的非常有用的工具。

思路是这样的：我们首先不使用几何着色器正常绘制场景。然后再次绘制场景，但这次只显示通过几何着色器生成法向量。几何着色器接收一个三角形图元，并沿着法向量生成三条线——每个顶点一个法向量。伪代码看起来会像是这样：

```
shader.use();
DrawScene();
normalDisplayShader.use();
DrawScene();
```

这次在几何着色器中，我们会使用模型提供的顶点法线，而不是自己生成，为了适配（观察和模型矩阵的）缩放和旋转，我们在将法线变换到裁剪空间坐标之前，先使用法线矩阵变换一次（几何着色器接受的位置向量是剪裁空间坐标，所以我们应该将法向量变换到相同的空间中）。这可以在顶点着色器中完成：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out VS_OUT {
    vec3 normal;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = normalize(vec3(projection * vec4(normalMatrix * aNormal, 0.0)));
}
```

变换后的裁剪空间法向量会以接口块的形式传递到下个着色器阶段。接下来，几何着色器会接收每一个顶点（包括一个位置向量和一个法向量），并在每个位置向量处绘制一个法线向量：

```
#version 330 core
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
    vec3 normal;
} gs_in[];

const float MAGNITUDE = 0.4;

void GenerateLine(int index)
{
    gl_Position = gl_in[index].gl_Position;
    EmitVertex();
    gl_Position = gl_in[index].gl_Position + vec4(gs_in[index].normal, 0.0) * MAGNITUDE;
    EmitVertex();
    EndPrimitive();
}

void main()
{
    GenerateLine(0); // 第一个顶点法线
    GenerateLine(1); // 第二个顶点法线
    GenerateLine(2); // 第三个顶点法线
}
```

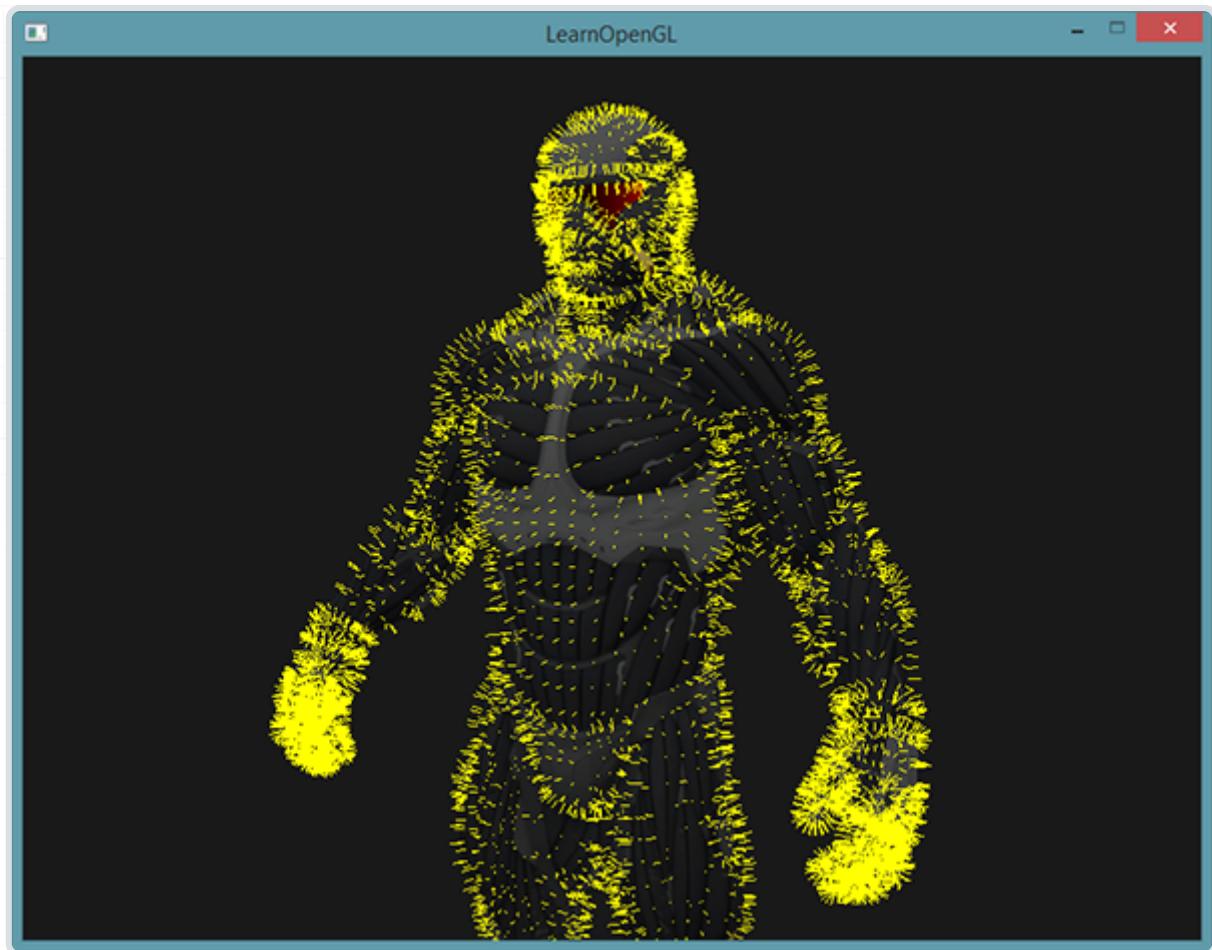
像这样的几何着色器应该很容易理解了。注意我们将法向量乘以了一个MAGNITUDE向量，来限制显示出的法向量大小（否则它们就有点大了）。

因为法线的可视化通常都是用于调试目的，我们可以使用片段着色器，将它们显示为单色的线（如果你愿意也可以是非常好看的线）：

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

现在，首先使用普通着色器渲染模型，再使用特别的法线可视化着色器渲染，你将看到这样的效果：



尽管我们的纳米装现在看起来像是一个体毛很多而且带着隔热手套的人，它能够很有效地帮助我们判断模型的法线是否正确。你可以想象到，这样的几何着色器也经常用于给物体添加毛发(Fur)。

你可以在[这里](#)找到源码。

2.5.19 实例化

原文 [Instancing](#)

作者 JoeyDeVries

翻译 Krasjet

校对 暂未校对

假设你有一个绘制了很多模型的场景，而大部分的模型包含的是同一组顶点数据，只不过进行的是不同的世界空间变换。想象一个充满草的场景：每根草都是一个包含几个三角形的小模型。你可能会需要绘制很多根草，最终在每帧中你可能会需要渲染上千或者上万根草。因为每一根草仅仅是由几个三角形构成，渲染几乎是瞬间完成的，但上千个渲染函数调用却会极大地影响性能。

如果我们需要渲染大量物体时，代码看起来会像这样：

```
for(unsigned int i = 0; i < amount_of_models_to_draw; i++)
{
    DoSomePreparations(); // 绑定VAO, 绑定纹理, 设置uniform等
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);
}
```

如果像这样绘制模型的大量[实例](#)(Instance)，你很快就会因为绘制调用过多而达到性能瓶颈。与绘制顶点本身相比，使用[glDrawArrays](#)或[glDrawElements](#)函数告诉GPU去绘制你的顶点数据会消耗更多的性能，因为OpenGL在绘制顶点数据之前需要做很多准备工作（比如告诉GPU该从哪个缓冲读取数据，从哪寻找顶点属性，而且这些都是在相对缓慢的CPU到GPU总线(CPU to GPU Bus)上进行的）。所以，即便渲染顶点非常快，命令GPU去渲染却未必。

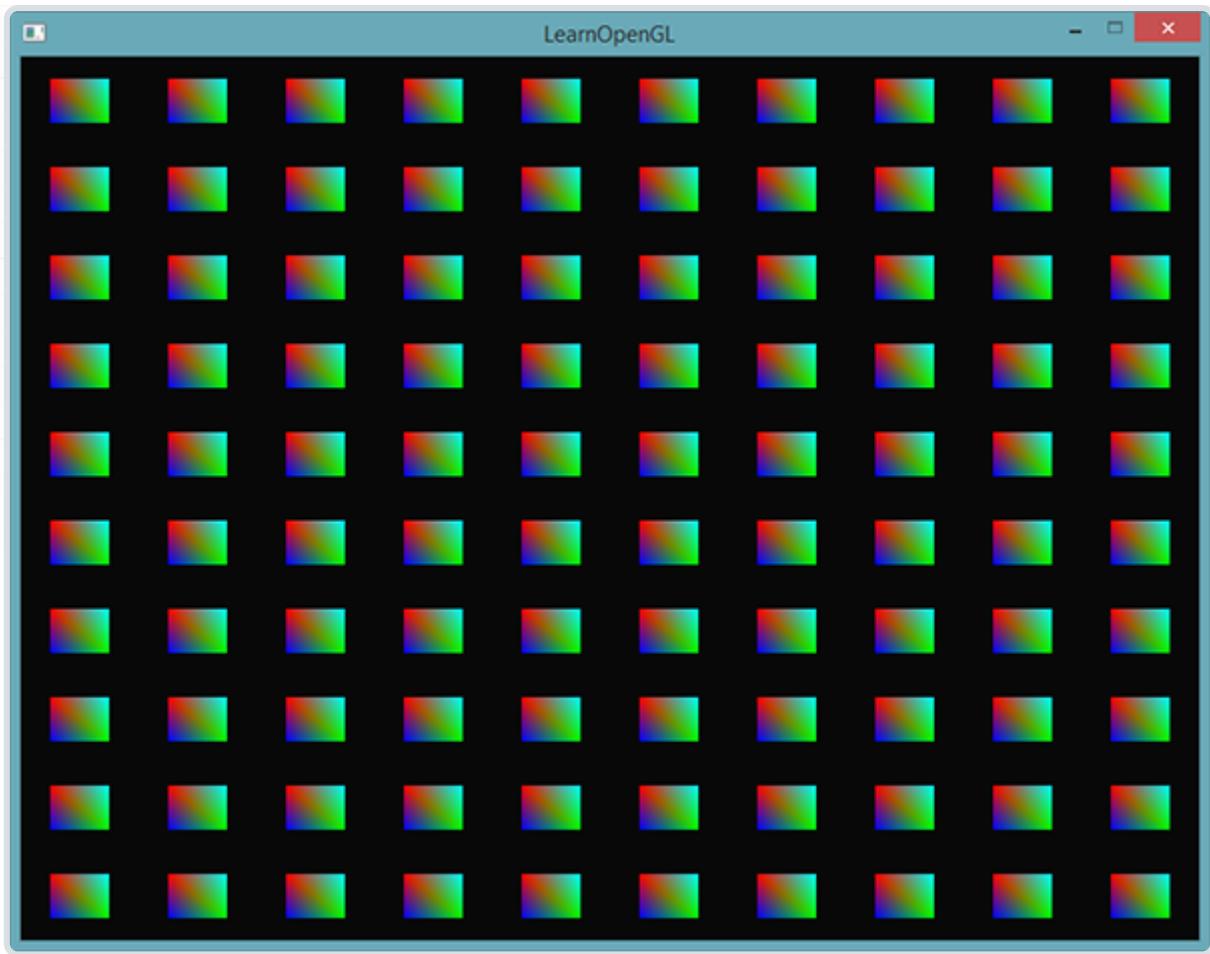
如果我们能够将数据一次性发送给GPU，然后使用一个绘制函数让OpenGL利用这些数据绘制多个物体，就会更方便了。这就是[实例化](#)(Instancing)。

实例化这项技术能够让我们使用一个渲染调用来绘制多个物体，来节省每次绘制物体时CPU -> GPU的通信，它只需要一次即可。如果想使用实例化渲染，我们只需要将[glDrawArrays](#)和[glDrawElements](#)的渲染调用分别改为[glDrawArraysInstanced](#)和[glDrawElementsInstanced](#)就可以了。这些渲染函数的实例化版本需要一个额外的参数，叫做[实例数量](#)(Instance Count)，它能够设置我们需要渲染的实例个数。这样我们只需要将必须的数据发送到GPU一次，然后使用一次函数调用告诉GPU它应该如何绘制这些实例。GPU将会直接渲染这些实例，而不用不断地与CPU进行通信。

这个函数本身并没有什么用。渲染同一个物体一千次对我们并没有什么用处，每个物体都是完全相同的，而且还在同一个位置。我们只能看见一个物体！出于这个原因，GLSL在顶点着色器中嵌入了另一个内建变量，[gl_InstanceID](#)。

在使用实例化渲染调用时，[gl_InstanceID](#)会从0开始，在每个实例被渲染时递增1。比如说，我们正在渲染第43个实例，那么顶点着色器中它的[gl_InstanceID](#)将会是42。因为每个实例都有唯一的ID，我们可以建立一个数组，将ID与位置值对应起来，将每个实例放置在世界的不同位置。

为了体验一下实例化绘制，我们将会在标准化设备坐标系中使用一个渲染调用，绘制100个2D四边形。我们会索引一个包含100个偏移向量的uniform数组，将偏移值加到每个实例化的四边形上。最终的结果是一个排列整齐的四边形网格：



每个四边形由2个三角形所组成，一共有6个顶点。每个顶点包含一个2D的标准化设备坐标位置向量和一个颜色向量。下面就是这个例子使用的顶点数据，为了大量填充屏幕，每个三角形都很小：

```
float quadVertices[] = {
    // 位置          // 颜色
    -0.05f,  0.05f,  1.0f, 0.0f, 0.0f,
    0.05f, -0.05f,  0.0f, 1.0f, 0.0f,
    -0.05f, -0.05f,  0.0f, 0.0f, 1.0f,

    -0.05f,  0.05f,  1.0f, 0.0f, 0.0f,
    0.05f, -0.05f,  0.0f, 1.0f, 0.0f,
    0.05f,  0.05f,  0.0f, 1.0f, 1.0f
};
```

片段着色器会从顶点着色器接受颜色向量，并将其设置为它的颜色输出，来实现四边形的颜色：

```
#version 330 core
out vec4 FragColor;

in vec3 fColor;

void main()
{
    FragColor = vec4(fColor, 1.0);
}
```

到现在都没有什么新内容，但从顶点着色器开始就变得很有趣了：

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fColor;

uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

这里我们定义了一个叫做`offsets`的数组，它包含100个偏移向量。在顶点着色器中，我们会使用`gl_InstanceID`来索引`offsets`数组，获取每个实例的偏移向量。如果我们要实例化绘制100个四边形，仅使用这个顶点着色器我们就能得到100个位于不同位置的四边形。

当前，我们仍要设置这些偏移位置，我们会在进入渲染循环之前使用一个嵌套for循环计算：

```
glm::vec2 translations[100];
int index = 0;
float offset = 0.1f;
for(int y = -10; y < 10; y += 2)
{
    for(int x = -10; x < 10; x += 2)
    {
        glm::vec2 translation;
        translation.x = (float)x / 10.0f + offset;
        translation.y = (float)y / 10.0f + offset;
        translations[index++] = translation;
    }
}
```

这里，我们创建100个位移向量，表示10x10网格上的所有位置。除了生成`translations`数组之外，我们还需要将数据转移到顶点着色器的uniform数组中：

```
shader.use();
for(unsigned int i = 0; i < 100; i++)
{
    stringstream ss;
    string index;
    ss << i;
    index = ss.str();
    shader.setVec2(("offsets[" + index + "]").c_str(), translations[i]);
}
```

在这一段代码中，我们将for循环的计数器`i`转换为一个`string`，我们可以用它来动态创建位置值的字符串，用于uniform位置值的索引。接下来，我们会对`offsets` uniform数组中的每一项设置对应的位移向量。

现在所有的准备工作都做完了，我们可以开始渲染四边形了。对于实例化渲染，我们使用`glDrawArraysInstanced`或`glDrawElementsInstanced`。因为我们使用的不是索引缓冲，我们会调用`glDrawArrays`版本的函数：

```
glBindVertexArray(quadVA0);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
```

`glDrawArraysInstanced`的参数和`glDrawArrays`完全一样，除了最后多了个参数用来设置需要绘制的实例数量。因为我们想要在10x10网格中显示100个四边形，我们将它设置为100。运行代码之后，你应该能得到熟悉的100个五彩的四边形。

实例化数组

虽然之前的实现在目前的情况下能够正常工作，但是如果我们要渲染远超过100个实例的时候（这其实非常普遍），我们最终会超过最大能够发送至着色器的uniform数据大小上限。它的一个代替方案是**实例化数组**(Instanced Array)，它被定义为一个顶点属性（能够让我们储存更多的数据），仅在顶点着色器渲染一个新的实例时才会更新。

使用顶点属性时，顶点着色器的每次运行都会让GLSL获取新一组适用于当前顶点的属性。而当我们把顶点属性定义为一个实例化数组时，顶点着色器就只需要对每个实例，而不是每个顶点，更新顶点属性的内容了。这允许我们对逐顶点的数据使用普通的顶点属性，而对逐实例的数据使用实例化数组。

为了给你一个实例化数组的例子，我们将使用之前的例子，并将偏移量uniform数组设置为一个实例化数组。我们需要在顶点着色器中再添加一个顶点属性：

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main()
{
    gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

我们不再使用`gl_InstanceID`，现在不需要索引一个uniform数组就能够直接使用`offset`属性了。

因为实例化数组和`position`与`color`变量一样，都是顶点属性，我们还需要将它的内容存在顶点缓冲对象中，并且配置它的属性指针。我们首先将（上一部分的）`translations`数组存到一个新的缓冲对象中：

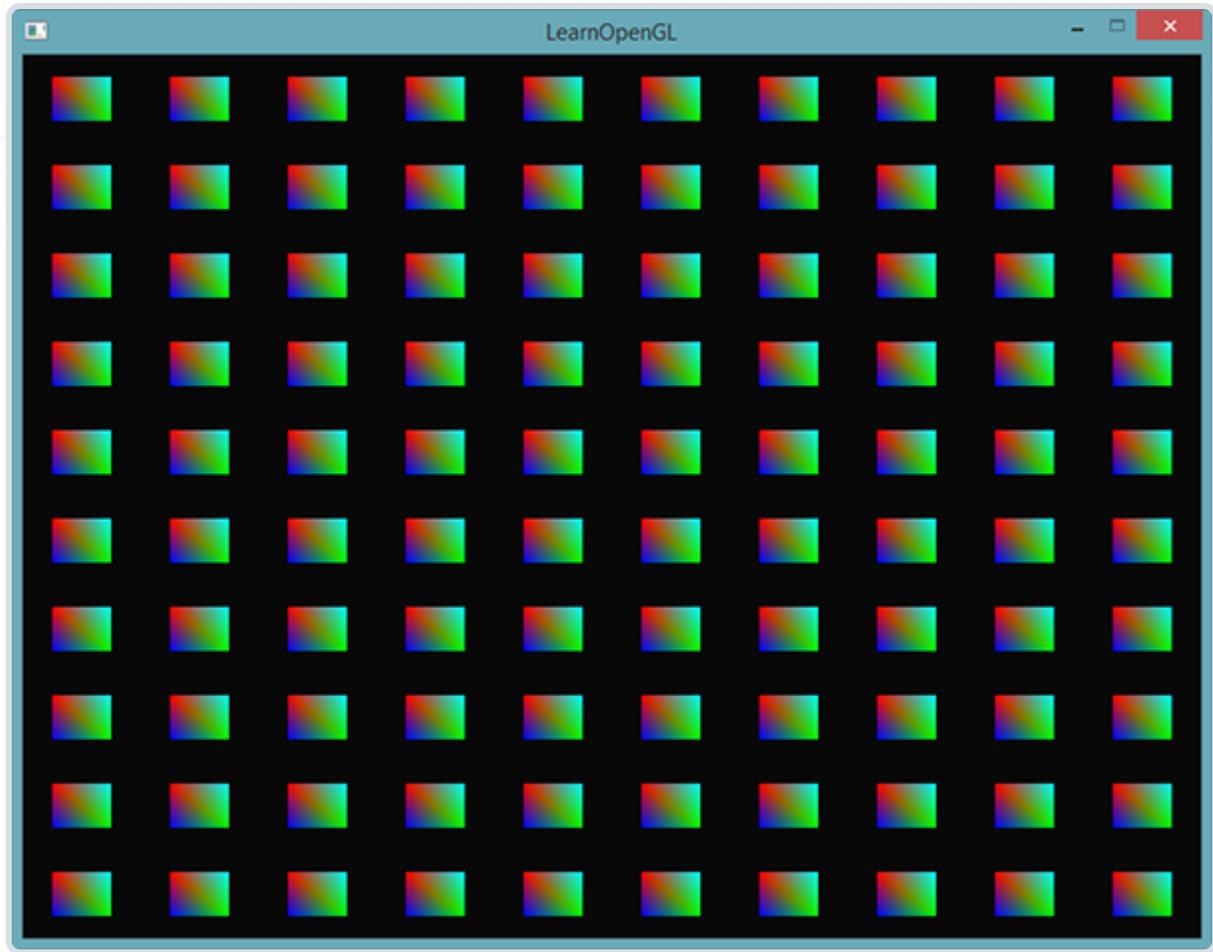
```
unsigned int instanceVBO;
 glGenBuffers(1, &instanceVBO);
 glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
 glBindBuffer(GL_ARRAY_BUFFER, sizeof(glm::vec2) * 100, &translations[0], GL_STATIC_DRAW);
 glBindBuffer(GL_ARRAY_BUFFER, 0);
```

之后我们还需要设置它的顶点属性指针，并启用顶点属性：

```
	glEnableVertexAttribArray(2);
	glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
	glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
	glBindBuffer(GL_ARRAY_BUFFER, 0);
	glVertexAttribDivisor(2, 1);
```

这段代码很有意思的地方在于最后一行，我们调用了`glVertexAttribDivisor`。这个函数告诉了OpenGL该什么时候更新顶点属性的内容至新一组数据。它的第一个参数是需要的顶点属性，第二个参数是属性除数(Attribute Divisor)。默认情况下，属性除数是0，告诉OpenGL我们需要在顶点着色器的每次迭代时更新顶点属性。将它设置为1时，我们告诉OpenGL我们希望在渲染一个新实例的时候更新顶点属性。而设置为2时，我们希望每2个实例更新一次属性，以此类推。我们将属性除数设置为1，是在告诉OpenGL，处于位置值2的顶点属性是一个实例化数组。

如果我们现在使用`glDrawArraysInstanced`，再次渲染四边形，会得到以下输出：

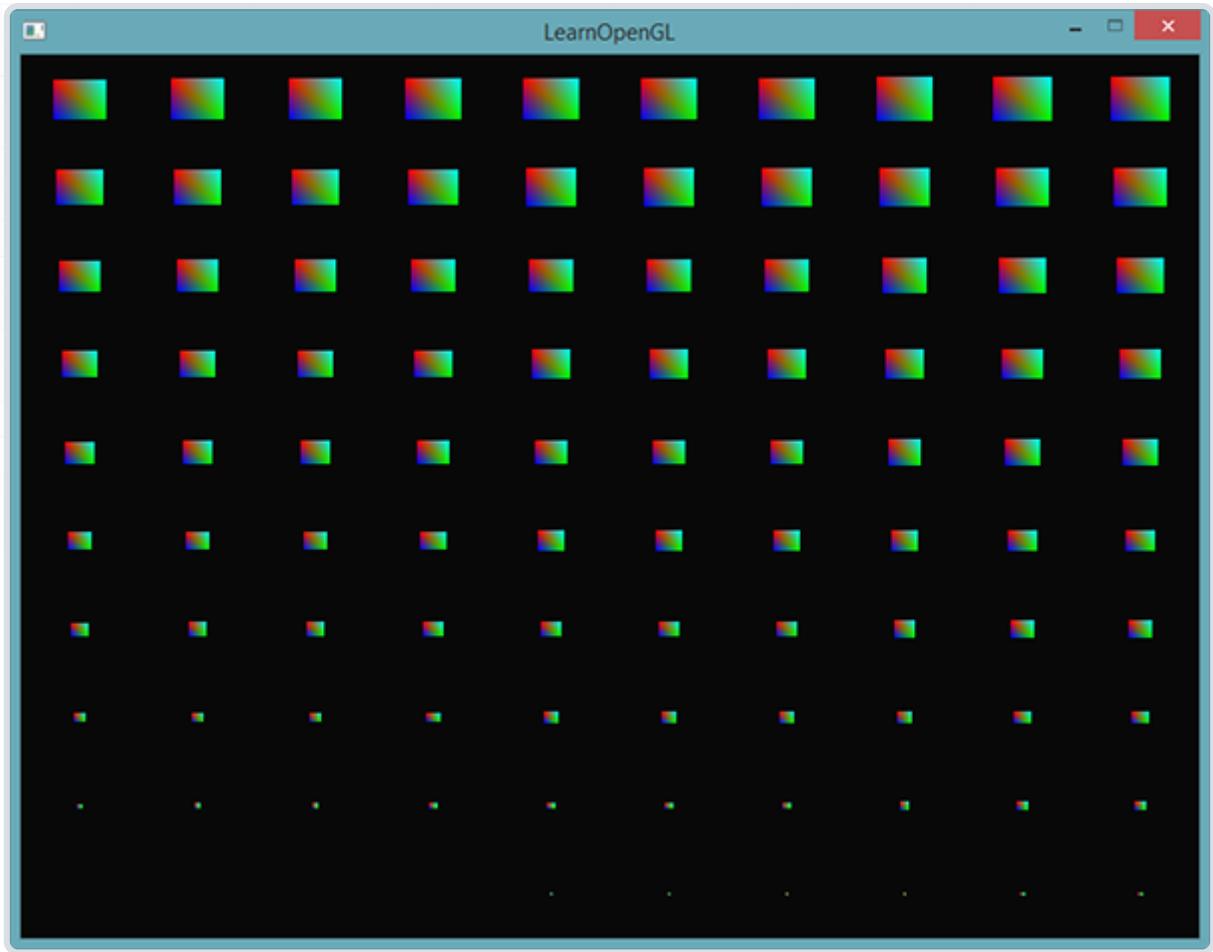


这和之前的例子是完全一样的，但这次是使用实例化数组实现的，这让我们能够传递更多的数据到顶点着色器（只要内存允许）来用于实例化绘制。

为了更有趣一点，我们也可以使用`gl_InstanceID`，从右上到左下逐渐缩小四边形：

```
void main()
{
    vec2 pos = aPos * (gl_InstanceID / 100.0);
    gl_Position = vec4(pos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

结果就是，第一个四边形的实例会非常小，随着绘制实例的增加，`gl_InstanceID`会越来越接近100，四边形也就越来越接近原始大小。像这样将实例化数组与`gl_InstanceID`结合使用是完全可行的。



如果你还是不确定实例化渲染是如何工作的，或者想看看所有代码是如何组合起来的，你可以在[这里](#)找到程序的源代码。

虽然很有趣，但是这些例子并不是实例化的好例子。是的，它们的确让你知道实例化是怎么工作的，但是我们还没接触到它最有用的一点：绘制巨大数量的相似物体。出于这个原因，我们将会在下一部分进入太空探险，见识实例化渲染真正的威力。

2.5.20 小行星带

想象这样一个场景，在宇宙中有一个大的行星，它位于小行星带的中央。这样的小行星带可能包含成千上万的岩块，在很不错的显卡上也很难完成这样的渲染。实例化渲染正是适用于这样的场景，因为所有的小行星都可以使用一个模型来表示。每个小行星可以再使用不同的变换矩阵来进行少许的变化。

为了展示实例化渲染的作用，我们首先会不使用实例化渲染，来渲染小行星绕着行星飞行的场景。这个场景将会包含一个大的行星模型，它可以在[这里](#)下载，以及很多环绕着行星的小行星。小行星的岩石模型可以在[这里](#)下载。

在代码例子中，我们将使用在[模型加载](#)小节中定义的模型加载器来加载模型。

为了得到想要的效果，我们将会为每个小行星生成一个变换矩阵，用作它们的模型矩阵。变换矩阵首先将小行星位移到小行星带中的某处，我们还会加一个随机偏移值到这个偏移量上，让这个圆环看起来更自然一点。接下来，我们应用一个随机的缩放，并且以一个旋转向量为轴进行一个随机的旋转。最终的变换矩阵不仅能将小行星变换到行星的周围，而且会让它看起来更自然，与其它小行星不同。最终的结果是一个布满小行星的圆环，其中每一个小行星都与众不同。

```
unsigned int amount = 1000;
glm::mat4 *modelMatrices;
modelMatrices = new glm::mat4[amount];
srand(glfwGetTime()); // 初始化随机种子
float radius = 50.0;
float offset = 2.5f;
for(unsigned int i = 0; i < amount; i++)
{
    glm::mat4 model;
    // 1. 位移：分布在半径为 'radius' 的圆形上，偏移的范围是 [-offset, offset]
    float angle = (float)i / (float)amount * 360.0f;
    float displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float x = sin(angle) * radius + displacement;
    displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float y = displacement * 0.4f; // 让行星带的高度比x和z的宽度要小
    displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float z = cos(angle) * radius + displacement;
    model = glm::translate(model, glm::vec3(x, y, z));

    // 2. 缩放：在 0.05 和 0.25f 之间缩放
    float scale = (rand() % 20) / 100.0f + 0.05;
    model = glm::scale(model, glm::vec3(scale));

    // 3. 旋转：绕着一个（半）随机选择的旋转轴向量进行随机的旋转
    float rotAngle = (rand() % 360);
    model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

    // 4. 添加到矩阵的数组中
    modelMatrices[i] = model;
}
```

这段代码看起来可能有点吓人，但我们只是将小行星的 `x` 和 `z` 位置变换到了一个半径为 `radius` 的圆形上，并且在半径的基础上偏移了 `-offset` 到 `offset`。我们让 `y` 偏移的影响更小一点，让小行星带更扁平一点。接下来，我们应用了缩放和旋转变换，并将最终的变换矩阵储存在 `modelMatrices` 中，这个数组的大小是 `amount`。这里，我们一共生成 1000 个模型矩阵，每个小行星一个。

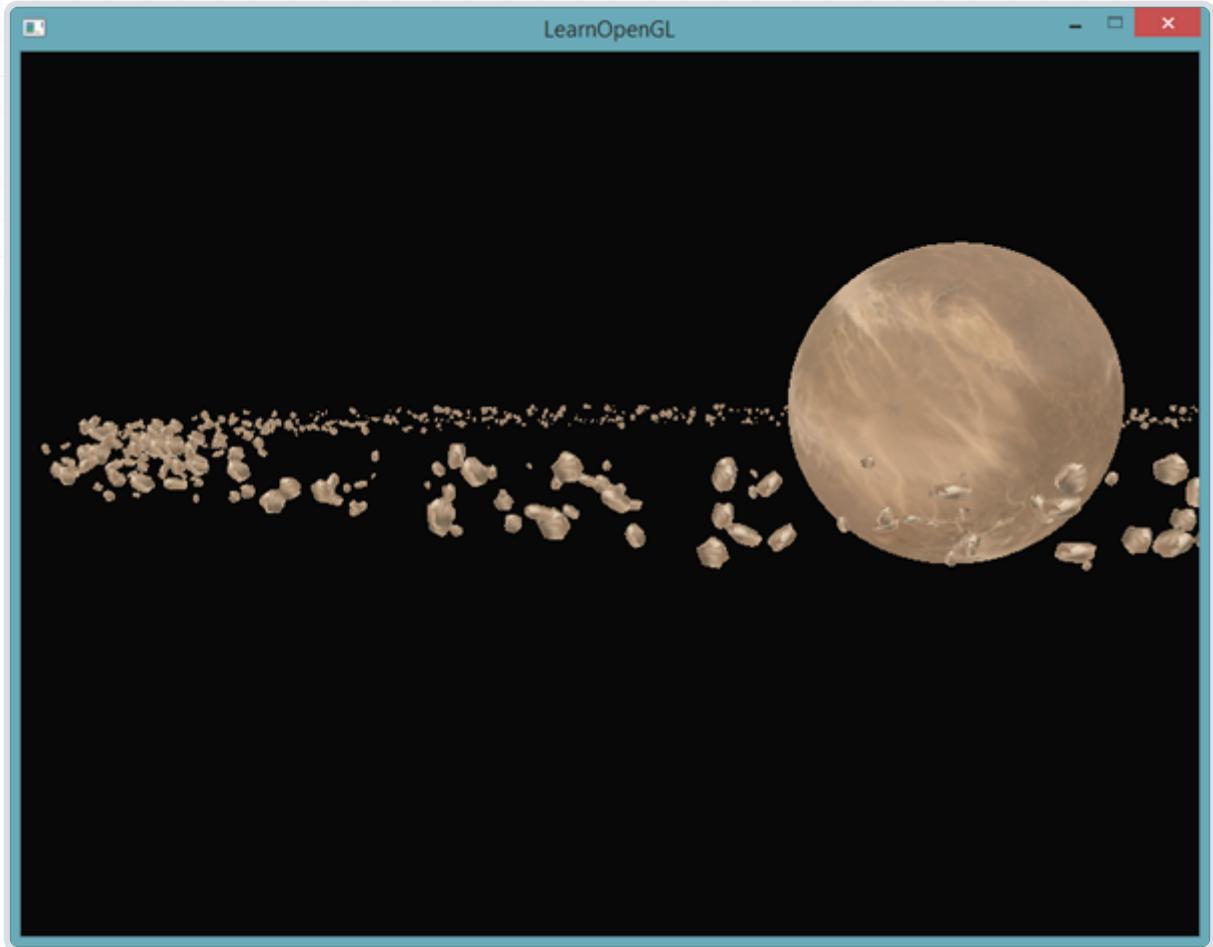
在加载完行星和岩石模型，并编译完着色器之后，渲染的代码看起来是这样的：

```
// 绘制行星
shader.use();
glm::mat4 model;
model = glm::translate(model, glm::vec3(0.0f, -3.0f, 0.0f));
model = glm::scale(model, glm::vec3(4.0f, 4.0f, 4.0f));
shader.setMat4("model", model);
planet.Draw(shader);

// 绘制小行星
for(unsigned int i = 0; i < amount; i++)
{
    shader.setMat4("model", modelMatrices[i]);
    rock.Draw(shader);
}
```

我们首先绘制了行星的模型，并对它进行位移和缩放，以适应场景，接下来，我们绘制`amount`数量的岩石模型。在绘制每个岩石之前，我们首先需要在着色器内设置对应的模型变换矩阵。

最终的结果是一个看起来像是太空的场景，环绕着行星的是看起来很自然的小行星带：



这个场景每帧包含1001次渲染调用，其中1000个是岩石模型。你可以在[这里](#)找到源代码。

当我们开始增加这个数字的时候，你很快就会发现场景不再能够流畅运行了，帧数也下降很厉害。当我们把`amount`设置为2000的时候，场景就已经慢到移动都很困难的程度了。

现在，我们来尝试使用实例化渲染来渲染相同的场景。我们首先对顶点着色器进行一点修改：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in mat4 instanceMatrix;

out vec2 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    gl_Position = projection * view * instanceMatrix * vec4(aPos, 1.0);
    TexCoords = aTexCoords;
}
```

我们不再使用模型uniform变量，改为一个mat4的顶点属性，让我们能够存储一个实例化数组的变换矩阵。然而，当我们顶点属性的类型大于vec4时，就要多进行一步处理了。顶点属性最大允许的数据大小等于一个vec4。因为一个mat4本质上是4个vec4，我们需要为这个矩阵预留4个顶点属性。因为我们将它的位置值设置为3，矩阵每一列的顶点属性位置值就是3、4、5和6。

接下来，我们需要为这4个顶点属性设置属性指针，并将它们设置为实例化数组：

```
// 顶点缓冲对象
unsigned int buffer;
 glGenBuffers(1, &buffer);
 glBindBuffer(GL_ARRAY_BUFFER, buffer);
 glBufferData(GL_ARRAY_BUFFER, amount * sizeof(glm::mat4), &modelMatrices[0], GL_STATIC_DRAW);

for(unsigned int i = 0; i < rock.meshes.size(); i++)
{
    unsigned int VAO = rock.meshes[i].VAO;
    glBindVertexArray(VAO);
    // 顶点属性
    GLsizei vec4Size = sizeof(glm::vec4);
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)0);
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(vec4Size));
    glEnableVertexAttribArray(5);
    glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(2 * vec4Size));
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(3 * vec4Size));

    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);

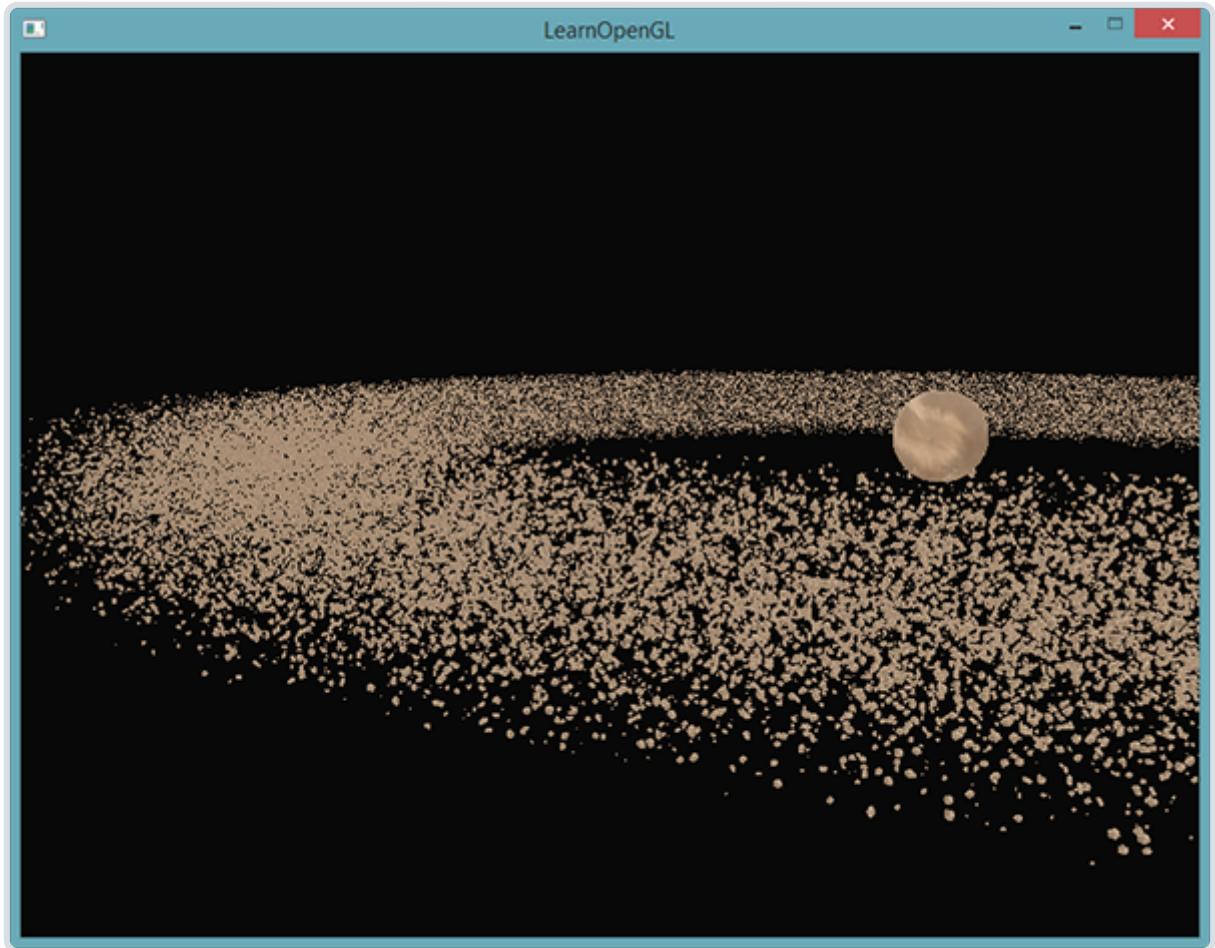
    glBindVertexArray(0);
}
```

注意这里我们将Mesh的VAO从私有变量改为了公有变量，让我们能够访问它的顶点数组对象。这并不是最好的解决方案，只是为了配合本小节的一个简单的改动。除此之外代码就应该很清楚了。我们告诉了OpenGL应该如何解释每个缓冲顶点属性的缓冲，并且告诉它这些顶点属性是实例化数组。

接下来，我们再次使用网格的VAO，这一次使用`glDrawElementsInstanced`进行绘制：

```
// 绘制小行星
instanceShader.use();
for(unsigned int i = 0; i < rock.meshes.size(); i++)
{
    glBindVertexArray(rock.meshes[i].VAO);
    glDrawElementsInstanced(
        GL_TRIANGLES, rock.meshes[i].indices.size(), GL_UNSIGNED_INT, 0, amount
    );
}
```

这里，我们绘制与之前相同数量`amount`的小行星，但是使用的是实例渲染。结果应该是非常相似的，但如果你开始增加`amount`变量，你就能看见实例化渲染的效果了。没有实例化渲染的时候，我们只能流畅渲染1000到1500个小行星。而使用了实例化渲染之后，我们可以将这个值设置为100000，每个岩石模型有576个顶点，每帧加起来大概要绘制5700万个顶点，但性能却没有受到任何影响！



上面这幅图渲染了10万个小行星，半径为`150.0f`，偏移量等于`25.0f`。你可以在[这里](#)找到实例化渲染的代码。

在某些机器上，10万个小行星可能会太多了，所以尝试修改这个值，直到达到一个你能接受的帧率。

可以看到，在合适的环境下，实例化渲染能够大大增加显卡的渲染能力。正是出于这个原因，实例化渲染通常会用于渲染草、植被、粒子，以及上面这样的场景，基本上只要场景中有很多重复的形状，都能够使用实例化渲染来提高性能。

2.5.21 抗锯齿

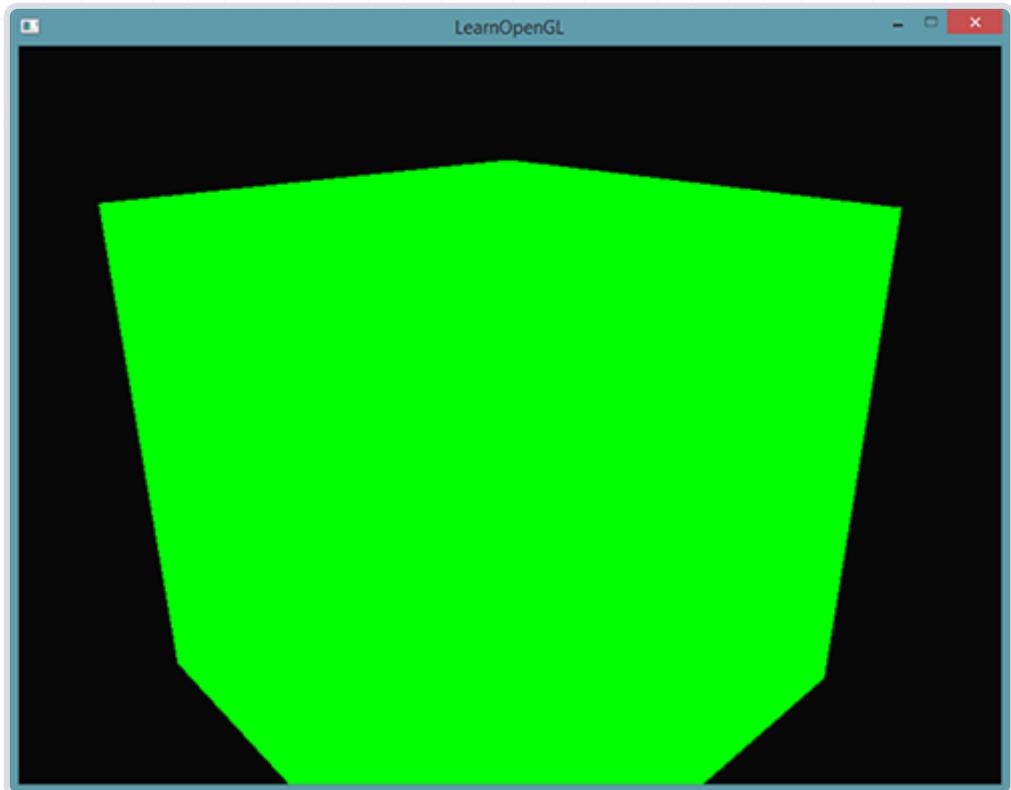
原文 [Anti Aliasing](#)

作者 JoeyDeVries

翻译 Krasjet, [Django](#)

校对 暂未校对

在学习渲染的旅途中，你可能会时不时遇到模型边缘有锯齿的情况。这些**锯齿边缘**(Jagged Edges)的产生和光栅器将顶点数据转化为片段的方式有关。在下面的例子中，你可以看到，我们只是绘制了一个简单的立方体，你就能注意到它存在锯齿边缘了：



可能不是非常明显，但如果你离近仔细观察立方体的边缘，你就应该能够看到锯齿状的图案。如果放大的话，你会看到下面的图案：



这很明显不是我们想要在最终程序中所实现的效果。你能够清楚看见形成边缘的像素。这种现象被称之为**走样**(Aliasing)。有很多种抗锯齿 (Anti-aliasing，也被称为反走样) 的技术能够帮助我们缓解这种现象，从而产生更平滑的边缘。

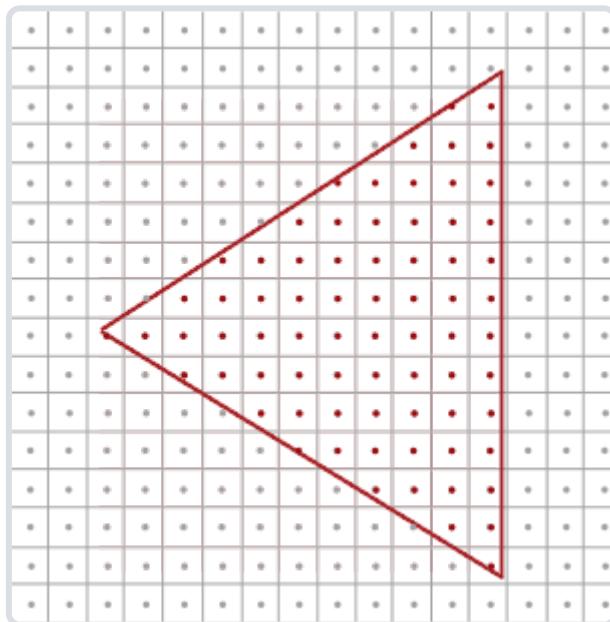
最开始我们有一种叫做**超采样抗锯齿**(Super Sample Anti-aliasing, SSAA)的技术，它会使用比正常分辨率更高的分辨率（即超采样）来渲染场景，当图像输出在帧缓冲中更新时，分辨率会被下采样(Downsample)至正常的分辨率。这些额外的分辨率会被用来防止锯齿边缘的产生。虽然它确实能够解决走样的问题，但是由于这样比平时要绘制更多的片段，它也会带来很大的性能开销。所以这项技术只拥有了短暂的辉煌。

然而，在这项技术的基础上也诞生了更为现代的技术，叫做**多重采样抗锯齿**(Multisample Anti-aliasing, MSAA)。它借鉴了SSAA背后的理念，但却以更加高效的方式实现了抗锯齿。我们在这一节中会深度讨论OpenGL中内建的MSAA技术。

多重采样

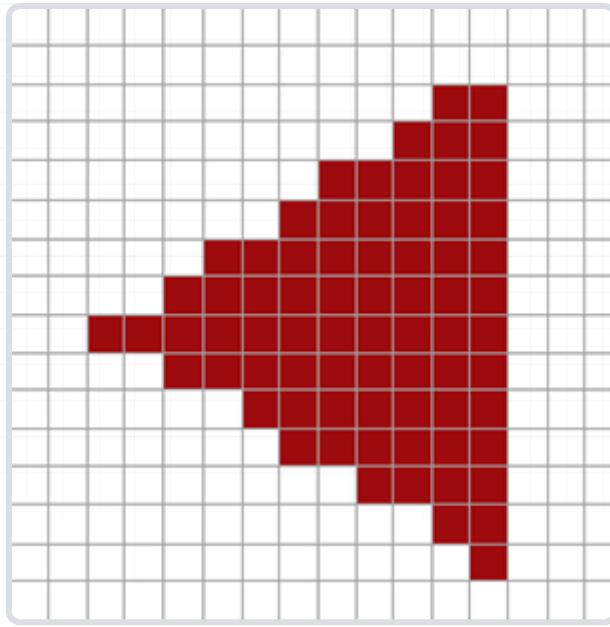
为了理解什么是多重采样(Multisampling)，以及它是如何解决锯齿问题的，我们有必要更加深入地了解OpenGL光栅器的工作方式。

光栅器是位于最终处理过的顶点之后到片段着色器之前所经过的所有算法与过程的总和。光栅器会将一个图元的所有顶点作为输入，并将它转换为一系列的片段。顶点坐标理论上可以取任意值，但片段不行，因为它们受限于你窗口的分辨率。顶点坐标与片段之间几乎永远也不会有一对一的映射，所以光栅器必须以某种方式来决定每个顶点最终所在的片段/屏幕坐标。



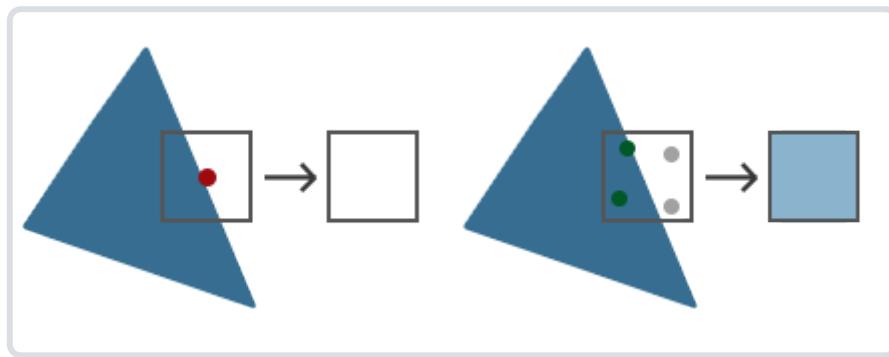
这里我们可以看到一个屏幕像素的网格，每个像素的中心包含有一个**采样点**(Sample Point)，它会被用来决定这个三角形是否遮盖了某个像素。图中红色的采样点被三角形所遮盖，在每一个遮住的像素处都会生成一个片段。虽然三角形边缘的一些部分也遮住了某些屏幕像素，但是这些像素的采样点并没有被三角形内部所遮盖，所以它们不会受到片段着色器的影响。

你现在可能已经清楚走样的原因了。完整渲染后的三角形在屏幕上会是这样的：



由于屏幕像素总量的限制，有些边缘的像素能够被渲染出来，而有些则不会。结果就是我们使用了不光滑的边缘来渲染图元，导致之前讨论到的锯齿边缘。

多重采样所做的正是将单一的采样点变为多个采样点（这也是它名称的由来）。我们不再使用像素中心的单一采样点，取而代之的是以特定图案排列的4个子采样点(Subsample)。我们将用这些子采样点来决定像素的遮盖度。当然，这也意味着颜色缓冲的大小会随着子采样点的增加而增加。



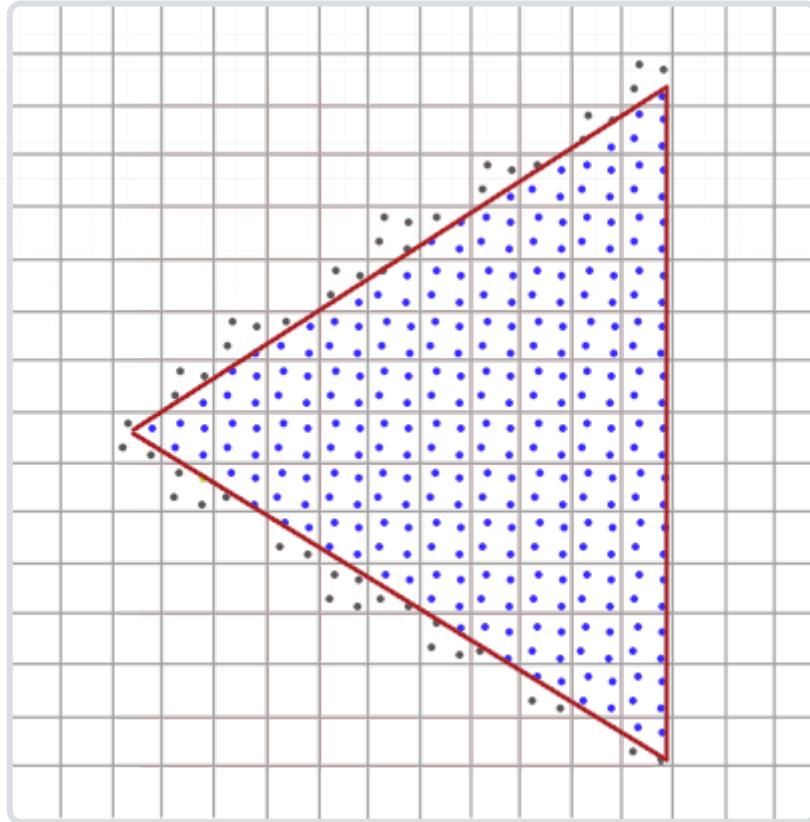
上图的左侧展示了正常情况下判定三角形是否遮盖的方式。在例子中的这个像素上不会运行片段着色器（所以它会保持空白）。因为它的采样点并未被三角形所覆盖。上图的右侧展示的是实施多重采样之后的版本，每个像素包含有4个采样点。这里，只有两个采样点遮盖住了三角形。

采样点的数量可以是任意的，更多的采样点能带来更精确的遮盖率。

从这里开始多重采样就变得有趣起来了。我们知道三角形只遮盖了2个子采样点，所以下一步是决定这个像素的颜色。你的猜想可能是，我们对每个被遮盖住的子采样点运行一次片段着色器，最后将每个像素所有子采样点的颜色平均一下。在这个例子中，我们需要在两个采样点上对被插值的顶点数据运行两次片段着色器，并将结果的颜色储存在这些采样点中。（幸运的是）这并不是它工作的方式，因为这本质上说还是需要运行更多次的片段着色器，会显著地降低性能。

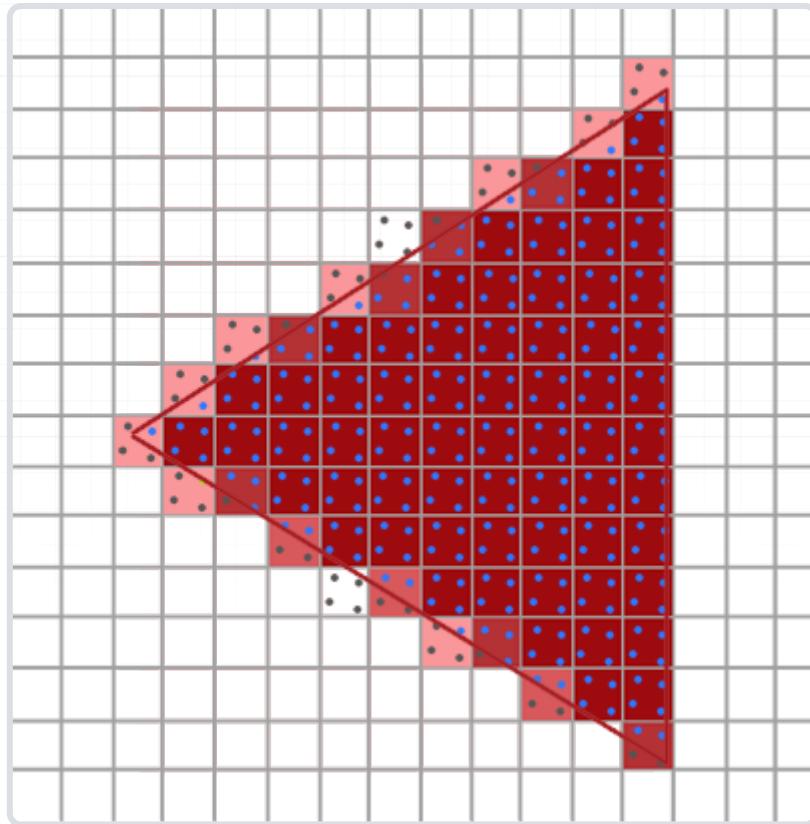
MSAA真正的工作方式是，无论三角形遮盖了多少个子采样点，（每个图元中）每个像素只运行一次片段着色器。片段着色器所使用的顶点数据会插值到每个像素的中心，所得到的结果颜色会被储存在每个被遮盖住的子采样点中。当颜色缓冲的子样本被图元的所有颜色填满时，所有的这些颜色将会在每个像素内部平均化。因为上图的4个采样点中只有2个被遮盖住了，这个像素的颜色将会是三角形颜色与其他两个采样点的颜色（在这里是无色）的平均值，最终形成一种淡蓝色。

这样子做之后，颜色缓冲中所有的图元边缘将会产生一种更平滑的图形。让我们来看看前面三角形的多重采样会是什么样子：



这里，每个像素包含4个子采样点（不相关的采样点都没有标注），蓝色的采样点被三角形所遮盖，而灰色的则没有。对于三角形的内部的像素，片段着色器只会运行一次，颜色输出会被存储到全部的4个子样本中。而在三角形的边缘，并不是所有的子采样点都被遮盖，所以片段着色器的结果将只会储存到部分的子样本中。根据被遮盖的子样本的数量，最终的像素颜色将由三角形的颜色与其它子样本中所储存的颜色来决定。

简单来说，一个像素中如果有更多的采样点被三角形遮盖，那么这个像素的颜色就会更接近于三角形的颜色。如果我们给上面的三角形填充颜色，就能得到以下的效果：



对于每个像素来说，越少的子采样点被三角形所覆盖，那么它受到三角形的影响就越小。三角形的不平滑边缘被稍浅的颜色所包围后，从远处观察时就会显得更加平滑了。

不仅仅是颜色值会受到多重采样的影响，深度和模板测试也能够使用多个采样点。对深度测试来说，每个顶点的深度值会在运行深度测试之前被插值到各个子样本中。对模板测试来说，我们对每个子样本，而不是每个像素，存储一个模板值。当然，这也意味着深度和模板缓冲的大小会乘以子采样点的个数。

我们到目前为止讨论的都是多重采样抗锯齿的背后原理，光栅器背后的实际逻辑比目前讨论的要复杂，但你现在应该已经可以理解多重采样抗锯齿的大体概念和逻辑了。

(译者注：如果看到这里还是对原理似懂非懂，可以简单看看知乎上[@文刀秋二](#)对抗锯齿技术的精彩介绍)

OpenGL中的MSAA

如果我们想要在OpenGL中使用MSAA，我们必须要使用一个能在每个像素中存储大于1个颜色值的颜色缓冲（因为多重采样需要我们为每个采样点都储存一个颜色）。所以，我们需要一个新的缓冲类型，来存储特定数量的多重采样样本，它叫做**多重采样缓冲**（Multisample Buffer）。

大多数的窗口系统都应该提供了一个多重采样缓冲，用以代替默认的颜色缓冲。GLFW同样给了我们这个功能，我们所要做的只是提示（Hint）GLFW，我们希望使用一个包含N个样本的多重采样缓冲。这可以在创建窗口之前调用`glfwWindowHint`来完成。

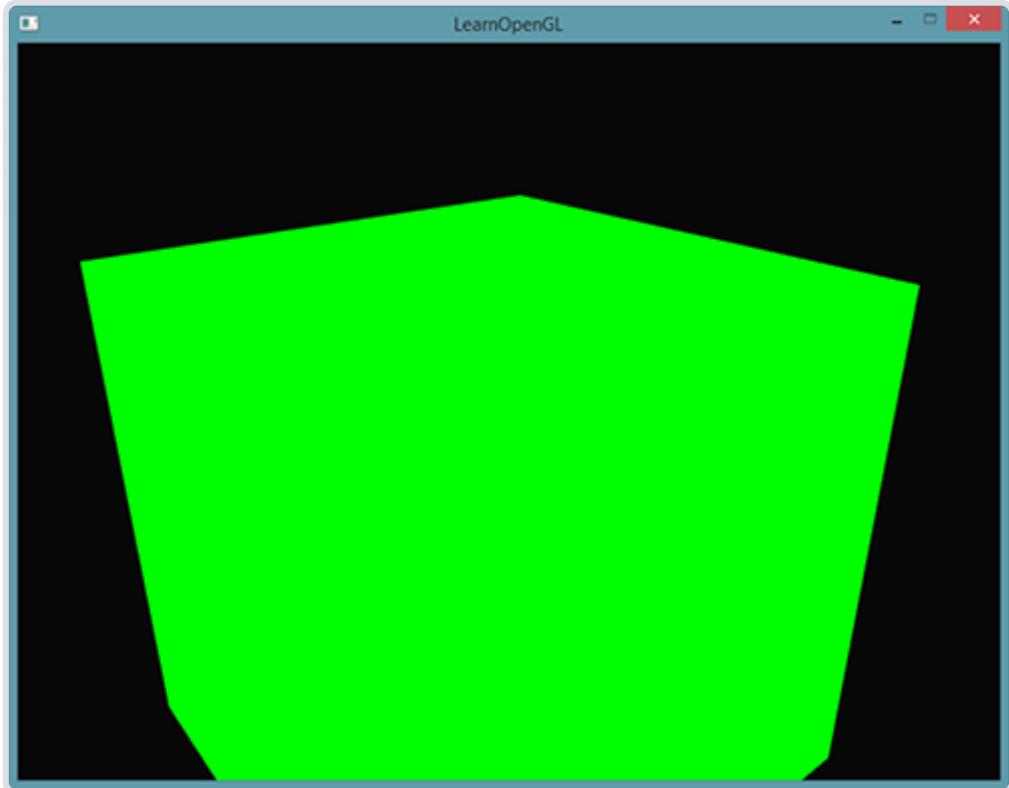
```
glfwWindowHint(GLFW_SAMPLES, 4);
```

现在再调用`glfwCreateWindow`创建渲染窗口时，每个屏幕坐标就会使用一个包含4个子采样点的颜色缓冲了。GLFW会自动创建一个每像素4个子采样点的深度和样本缓冲。这也意味着所有缓冲的大小都增长了4倍。

现在我们已经向GLFW请求了多重采样缓冲，我们还需要调用`glEnable`并启用`GL_MULTISAMPLE`，来启用多重采样。在大多数OpenGL的驱动上，多重采样都是默认启用的，所以这个调用可能会有点多余，但显式地调用一下会更保险一点。这样子不论是什么OpenGL的实现都能够正常启用多重采样了。

```
glEnable(GL_MULTISAMPLE);
```

只要默认的帧缓冲有了多重采样缓冲的附件，我们所要做的只是调用`glEnable`来启用多重采样。因为多重采样的算法都在OpenGL驱动的光栅器中实现了，我们不需要再多做什么。如果现在再来渲染本节一开始的那个绿色的立方体，我们应该能看到更平滑的边缘：



这个箱子看起来的确要平滑多了，如果在场景中有其它的物体，它们也会看起来平滑很多。你可以在[这里](#)找到这个简单例子的源代码。

离屏MSAA

由于GLFW负责了创建多重采样缓冲，启用MSAA非常简单。然而，如果我们想要使用我们自己的帧缓冲来进行离屏渲染，那么我们就必须要自己动手生成多重采样缓冲了。

有两种方式可以创建多重采样缓冲，将其作为帧缓冲的附件：纹理附件和渲染缓冲附件，这和在[帧缓冲](#)教程中所讨论的普通附件很相似。

多重采样纹理附件

为了创建一个支持储存多个采样点的纹理，我们使用`glTexImage2DMultisample`来替代`glTexImage2D`，它的纹理目标是`GL_TEXTURE_2D_MULTISAMPLE`。

```
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, samples, GL_RGB, width, height, GL_TRUE);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
```

它的第二个参数设置的是纹理所拥有的样本个数。如果最后一个参数为`GL_TRUE`, 图像将会对每个纹素使用相同的样本位置以及相同数量的子采样点个数。

我们使用`glFramebufferTexture2D`将多重采样纹理附加到帧缓冲上, 但这里纹理类型使用的是`GL_TEXTURE_2D_MULTISAMPLE`。

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, tex, 0);
```

当前绑定的帧缓冲现在就有了一个纹理图像形式的多重采样颜色缓冲。

多重采样渲染缓冲对象

和纹理类似, 创建一个多重采样渲染缓冲对象并不难。我们所要做的只是在指定 (当前绑定的) 渲染缓冲的内存存储时, 将`glRenderbufferStorage`的调用改为`glRenderbufferStorageMultisample`就可以了。

```
glRenderbufferStorageMultisample(GL_RENDERBUFFER, 4, GL_DEPTH24_STENCIL8, width, height);
```

函数中, 渲染缓冲对象后的参数我们将设定为样本的数量, 在当前的例子中是4。

渲染到多重采样帧缓冲

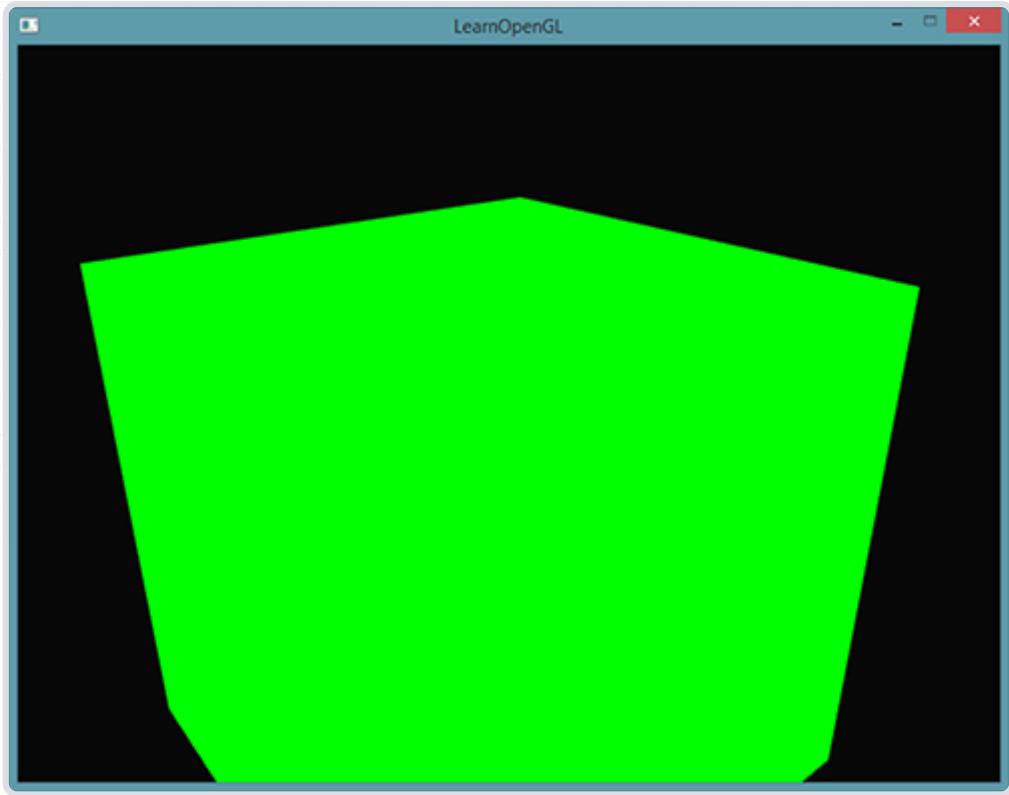
渲染到多重采样帧缓冲对象的过程都是自动的。只要我们在帧缓冲绑定时绘制任何东西, 光栅器就会负责所有的多重采样运算。我们最终会得到一个多重采样颜色缓冲以及/或深度和模板缓冲。因为多重采样缓冲有一点特别, 我们不能直接将它们的缓冲图像用于其他运算, 比如在着色器中对它们进行采样。

一个多重采样的图像包含比普通图像更多的信息, 我们所要做的是缩小或者还原(Resolve)图像。多重采样帧缓冲的还原通常是通过`glBlitFramebuffer`来完成, 它能够将一个帧缓冲中的某个区域复制到另一个帧缓冲中, 并且将多重采样缓冲还原。

`glBlitFramebuffer`会将一个用4个屏幕空间坐标所定义的源区域复制到一个同样用4个屏幕空间坐标所定义的目标区域中。你可能记得在帧缓冲教程中, 当我们绑定到`GL_FRAMEBUFFER`时, 我们是同时绑定了读取和绘制的帧缓冲目标。我们也可以将帧缓冲分开绑定至`GL_READ_FRAMEBUFFER`与`GL_DRAW_FRAMEBUFFER`。`glBlitFramebuffer`函数会根据这两个目标, 决定哪个是源帧缓冲, 哪个是目标帧缓冲。接下来, 我们可以将图像位块传送(Blit)到默认的帧缓冲中, 将多重采样的帧缓冲传送到屏幕上。

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, multisampledFB0);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

如果现在再来渲染这个程序, 我们会得到与之前完全一样的结果: 一个使用MSAA显示出来的橄榄绿色的立方体, 而且锯齿边缘明显减少了:



你可以在[这里](#)找到源代码。

但如果我们想要使用多重采样帧缓冲的纹理输出来做像是后期处理这样的事情呢？我们不能直接在片段着色器中使用多重采样的纹理。但我们能做的是将多重采样缓冲位块传送到一个没有使用多重采样纹理附件的FBO中。然后用这个普通的颜色附件来做后期处理，从而达到我们的目的。然而，这也意味着我们需要生成一个新的FBO，作为中介帧缓冲对象，将多重采样缓冲还原为一个能在着色器中使用的普通2D纹理。这个过程的伪代码是这样的：

```

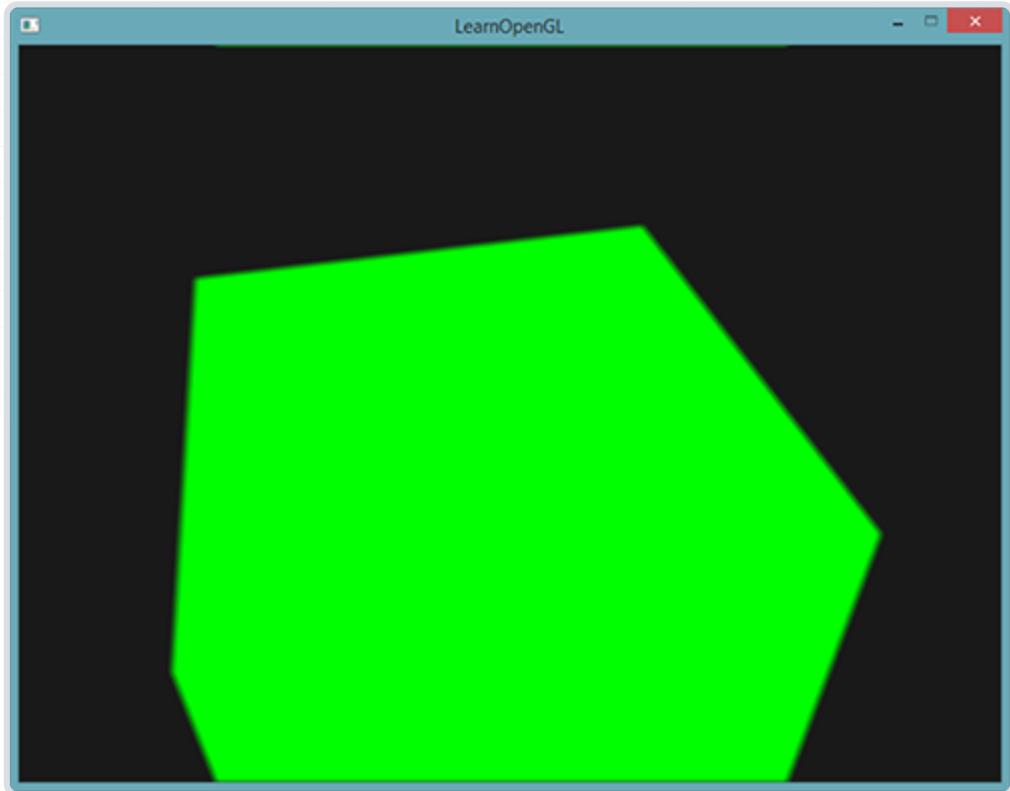
unsigned int msFBO = CreateFBOWithMultiSampledAttachments();
// 使用普通的纹理颜色附件创建一个新的FBO
...
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, screenTexture, 0);
...
while(!glfwWindowShouldClose(window))
{
    ...

    glBindFramebuffer(msFBO);
    ClearFrameBuffer();
    DrawScene();
    // 将多重采样缓冲还原到中介FBO上
    glBindFramebuffer(GL_READ_FRAMEBUFFER, msFBO);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, intermediateFBO);
    glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
    // 现在场景是一个2D纹理缓冲，可以将这个图像用来后期处理
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    ClearFrameBuffer();
    glBindTexture(GL_TEXTURE_2D, screenTexture);
    DrawPostProcessingQuad();

    ...
}

```

如果现在再实现[帧缓冲](#)教程中的后期处理效果，我们就能够在一个几乎没有锯齿的场景纹理上进行后期处理了。如果施加模糊的核滤镜，看起来将会是这样：



因为屏幕纹理又变回了一个只有单一采样点的普通纹理，像是边缘检测这样的后期处理滤镜会重新导致锯齿。为了补偿这一问题，你可以之后对纹理进行模糊处理，或者想出你自己的抗锯齿算法。

你可以看到，如果将多重采样与离屏渲染结合起来，我们需要自己负责一些额外的细节。但所有的这些细节都是值得额外的努力的，因为多重采样能够显著提升场景的视觉质量。当然，要注意，如果使用的采样点非常多，启用多重采样会显著降低程序的性能。在本节写作时，通常采用的是4采样点的MSAA。

自定义抗锯齿算法

将一个多重采样的纹理图像不进行还原直接传入着色器也是可行的。GLSL提供了这样的选项，让我们能够对纹理图像的每个子样本进行采样，所以我们可以创建我们自己的抗锯齿算法。在大型的图形应用中通常都会这么做。

要想获取每个子样本的颜色值，你需要将纹理uniform采样器设置为`sampler2DMS`，而不是平常使用的`sampler2D`：

```
uniform sampler2DMS screenTextureMS;
```

使用`texelFetch`函数就能够获取每个子样本的颜色值了：

```
vec4 colorSample = texelFetch(screenTextureMS, TexCoords, 3); // 第4个子样本
```

我们不会深入探究自定义抗锯齿技术的细节，这里仅仅是给你一点启发。

2.6 高级光照

2.6.1 高级光照

原文 [Advanced Lighting](#)

作者 JoeyDeVries

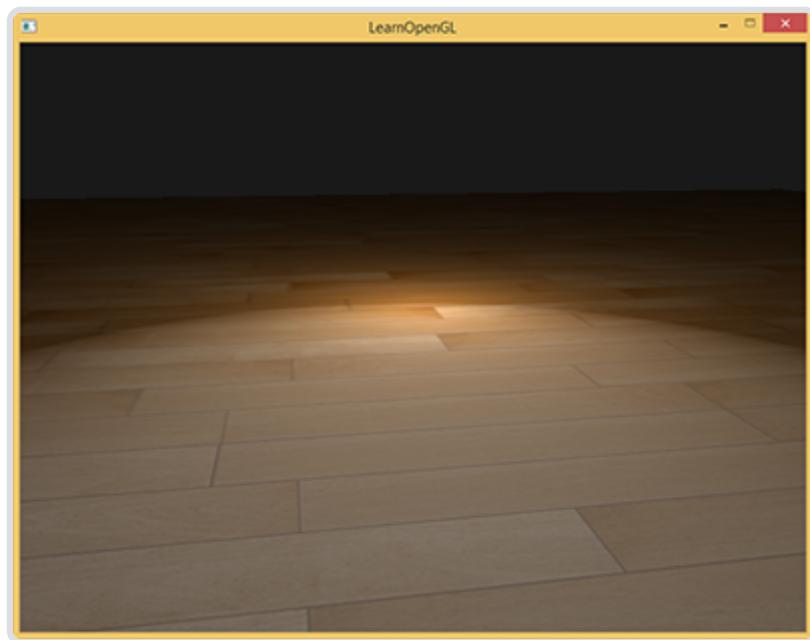
翻译 Krasjet

校对 暂未校对

在[光照](#)小节中，我们简单地介绍了冯氏光照模型，它让我们的场景有了一定的真实感。虽然冯氏模型看起来已经很不错了，但是使用它的时候仍然存在一些细节问题，我们将在这一节里讨论它们。

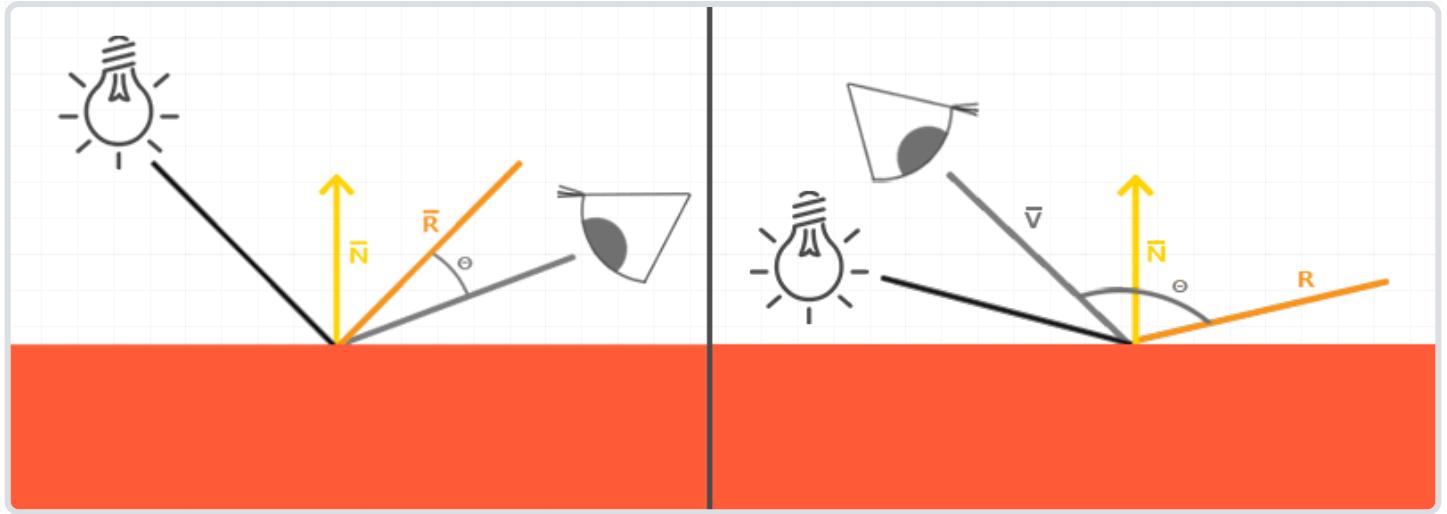
Blinn-Phong

冯氏光照不仅对真实光照有很好的近似，而且性能也很高。但是它的镜面反射会在一些情况下出现问题，特别是物体反光度很低时，会导致大片（粗糙的）高光区域。下面这张图展示了当反光度为1.0时地板会出现的效果：



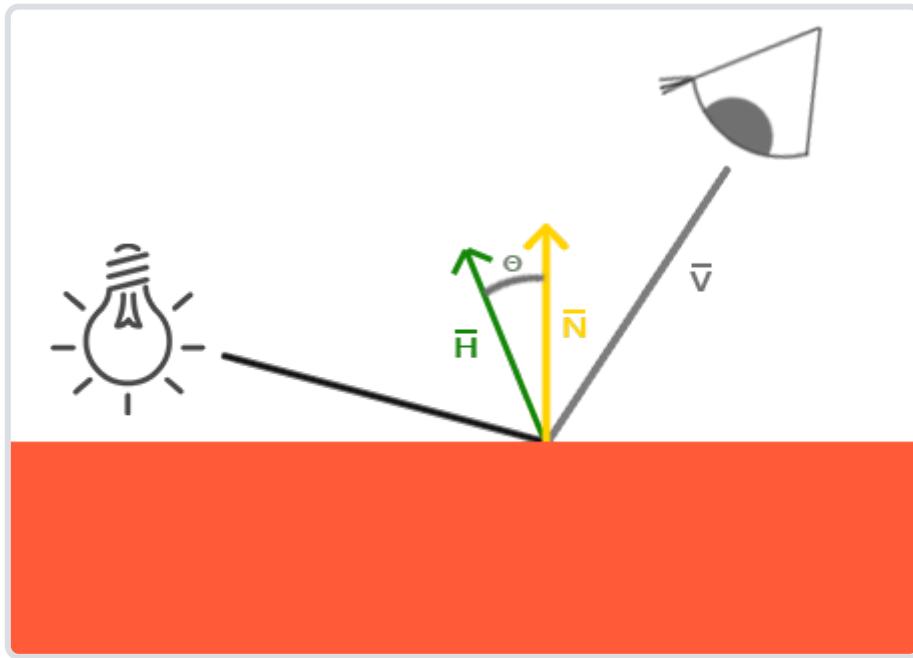
可以看到，在镜面高光区域的边缘出现了一道很明显的断层。出现这个问题的原因是观察向量和反射向量间的夹角不能大于90度。如果点积的结果为负数，镜面光分量会变为0.0。你可能会觉得，当光线与视线夹角大于90度时你应该不会接收到任何光才对，所以这不是什么问题。

然而，这种想法仅仅只适用于漫反射分量。当考虑漫反射光的时候，如果法线和光源夹角大于90度，光源会处于被照表面的下方，这个时候光照的漫反射分量的确是为0.0。但是，在考虑镜面高光时，我们测量的角度并不是光源与法线的夹角，而是视线与反射光线向量的夹角。看一下下面这两张图：



现在问题就应该很明显了。左图中是我们熟悉的冯氏光照中的反射向量，其中 θ 角小于90度。而右图中，视线与反射方向之间的夹角明显大于90度，这种情况下镜面光分量会变为0.0。这在大多数情况下都不是什么问题，因为观察方向离反射方向都非常远。然而，当物体的反光度非常小时，它产生的镜面高光半径足以让这些相反方向的光线对亮度产生足够大的影响。在这种情况下就不能忽略它们对镜面光分量的贡献了。

1977年，James F. Blinn在冯氏着色模型上加以拓展，引入了Blinn-Phong着色模型。Blinn-Phong模型与冯氏模型非常相似，但是它对镜面光模型的处理上有一些不同，让我们能够解决之前提到的问题。Blinn-Phong模型不再依赖于反射向量，而是采用了所谓的半程向量(Halfway Vector)，即光线与视线夹角一半方向上的一个单位向量。当半程向量与法线向量越接近时，镜面光分量就越大。



当视线正好与（现在不需要的）反射向量对齐时，半程向量就会与法线完美契合。所以当观察者视线越接近于原本反射光线的方向时，镜面高光就会越强。

现在，不论观察者向哪个方向看，半程向量与表面法线之间的夹角都不会超过90度（除非光源在表面以下）。它产生的效果会与冯氏光照有些许不同，但是大部分情况下看起来会更自然一点，特别是低高光的区域。Blinn-Phong着色模型正是早期固定渲染管线时代时OpenGL所采用的光照模型。

获取半程向量的方法很简单，只需要将光线的方向向量和观察向量加到一起，并将结果正规化(Normalize)就可以了：

翻译成GLSL代码如下：

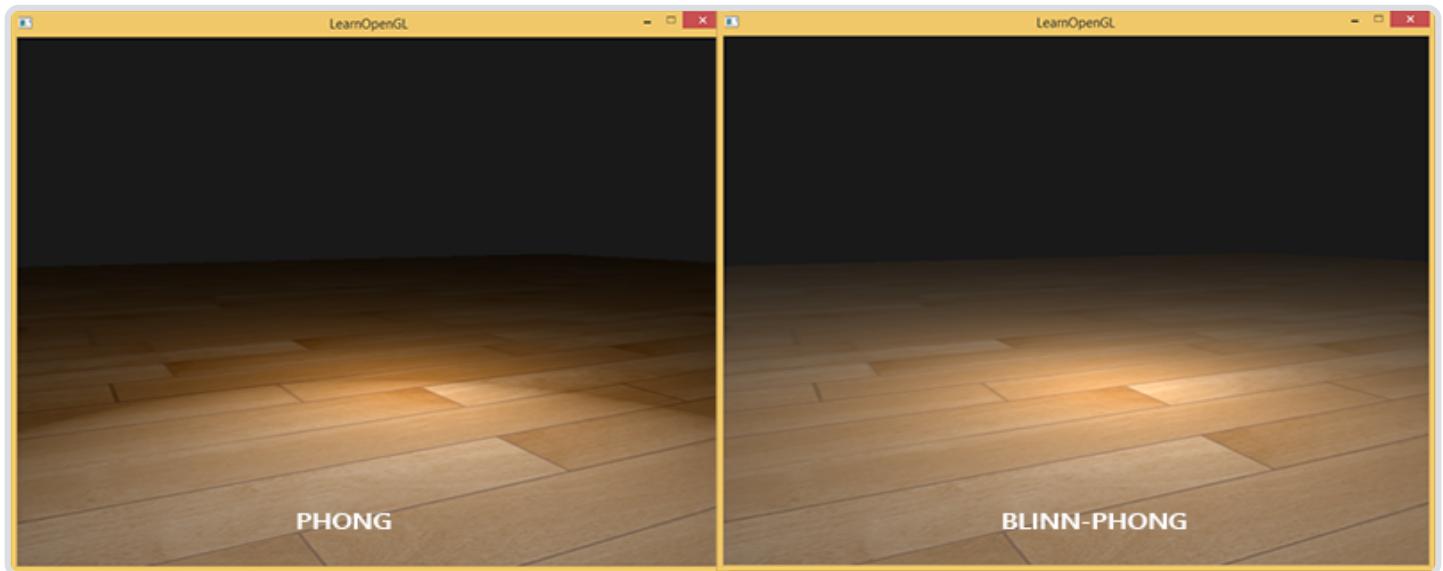
```
vec3 lightDir = normalize(lightPos - FragPos);
vec3 viewDir = normalize(viewPos - FragPos);
vec3 halfwayDir = normalize(lightDir + viewDir);
```

接下来，镜面光分量的实际计算只不过是对表面法线和半程向量进行一次约束点乘(Clamped Dot Product)，让点乘结果不为负，从而获取它们之间夹角的余弦值，之后我们对这个值取反光度次方：

```
float spec = pow(max(dot(normal, halfwayDir), 0.0), shininess);
vec3 specular = lightColor * spec;
```

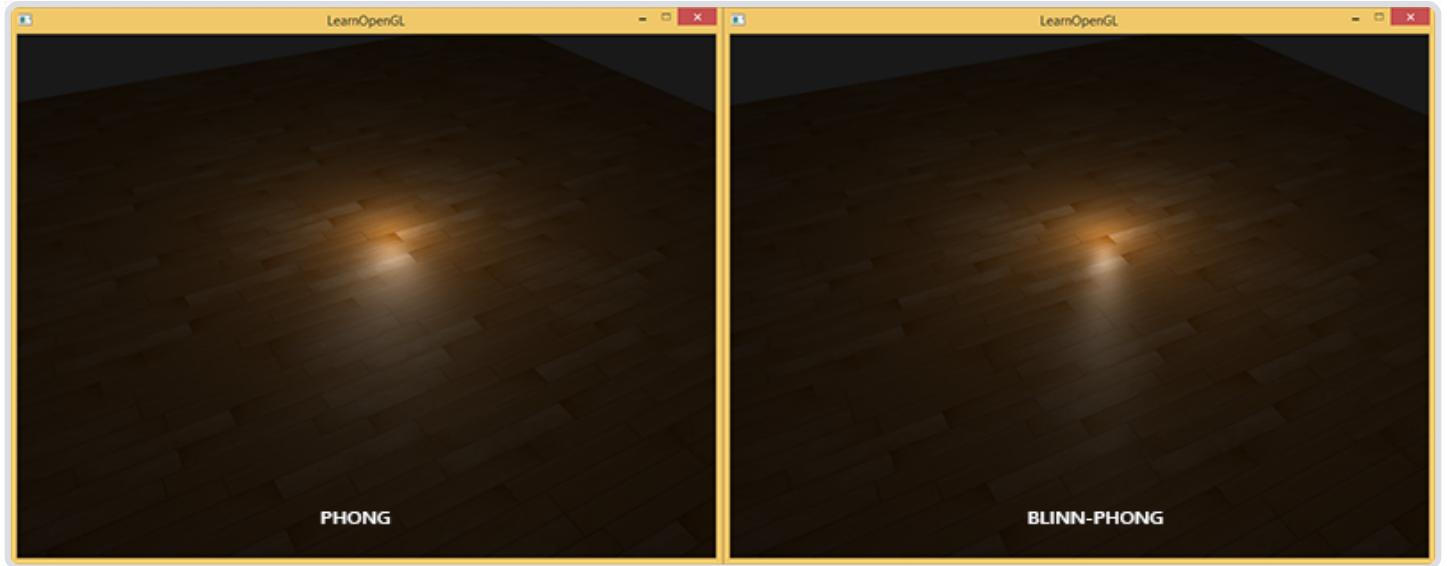
除此之外Blinn-Phong模型就没什么好说的了，Blinn-Phong与冯氏模型唯一的区别就是，Blinn-Phong测量的是法线与半程向量之间的夹角，而冯氏模型测量的是观察方向与反射向量间的夹角。

在引入半程向量之后，我们现在应该就不会再看到冯氏光照中高光断层的情况了。下面两个图片展示的是两种方法在镜面光分量为0.5时的对比：



除此之外，冯氏模型与Blinn-Phong模型也有一些细微的差别：半程向量与表面法线的夹角通常会小于观察与反射向量的夹角。所以，如果你想获得和冯氏着色类似的效果，就必须在使用Blinn-Phong模型时将镜面反光度设置更高一点。通常我们会选择冯氏着色时反光度分量的2到4倍。

下面是冯氏着色反光度为8.0，Blinn-Phong着色反光度为32.0时的一个对比：



你可以看到，Blinn-Phong的镜面光分量会比冯氏模型更锐利一些。为了得到与冯氏模型类似的结果，你可能会需要不断进行一些微调，但Blinn-Phong模型通常会产出更真实的结果。

这里，我们使用了一个简单的片段着色器，让我们能够在冯氏反射与Blinn-Phong反射间进行切换：

```
void main()
{
    [...]
    float spec = 0.0;
    if(blinn)
    {
        vec3 halfwayDir = normalize(lightDir + viewDir);
        spec = pow(max(dot(normal, halfwayDir), 0.0), 16.0);
    }
    else
    {
        vec3 reflectDir = reflect(-lightDir, normal);
        spec = pow(max(dot(viewDir, reflectDir), 0.0), 8.0);
    }
}
```

你可以在[这里](#)找到这个Demo的源代码。你可以按下 **B** 键来切换冯氏光照与Blinn-Phong光照。

2.6.2 Gamma校正

原文 [Gamma Correction](#)

作者 JoeyDeVries

翻译 [Django](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

当我们计算出场景中所有像素的最终颜色以后，我们就必须把它们显示在监视器上。过去，大多数监视器是阴极射线管显示器（CRT）。这些监视器有一个物理特性就是两倍的输入电压产生的不是两倍的亮度。输入电压产生约为输入电压的2.2次幂的亮度，这叫做监视器Gamma。

译注

Gamma也叫灰度系数，每种显示设备都有自己的Gamma值，都不相同，有一个公式：设备输出亮度 = 电压的Gamma次幂，任何设备Gamma基本上都不会等于1，等于1是一种理想的线性状态，这种理想状态是：如果电压和亮度都是在0到1的区间，那么多少电压就等于多少亮度。对于CRT，Gamma通常为2.2，因而，输出亮度 = 输入电压的2.2次幂，你可以从本节第二张图中看到Gamma2.2实际显示出来的总会比预期暗，相反Gamma0.45就会比理想预期亮，如果你讲Gamma0.45叠加到Gamma2.2的显示设备上，便会对偏暗的显示效果做到校正，这个简单的思路就是本节的核心

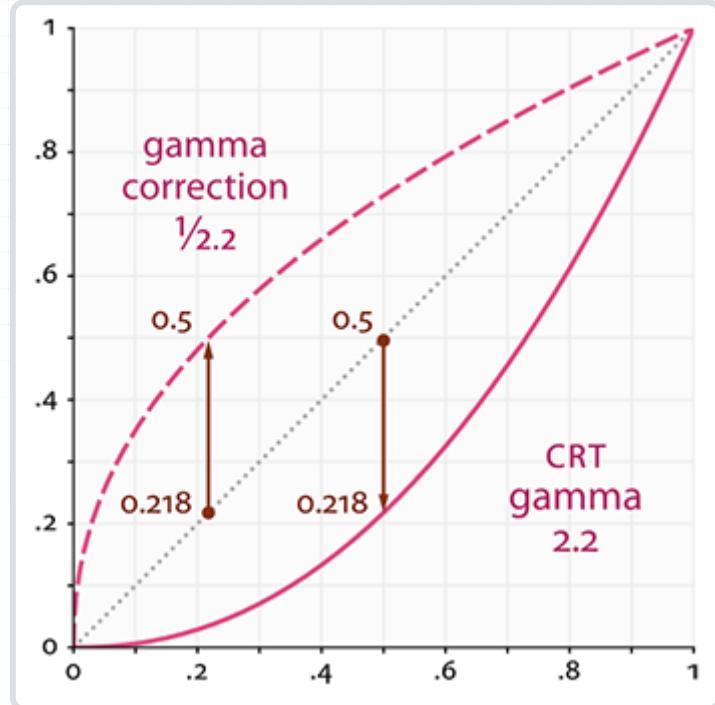
人类所感知的亮度恰好和CRT所显示出来相似的指数关系非常匹配。为了更好的理解所有含义，请看下面的图片：

Perceived (linear) brightness =	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Physical (linear) brightness =	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0

第一行是人眼所感知到的正常的灰阶，亮度要增加一倍（比如从0.1到0.2）你才会感觉比原来变亮了一倍（译注：我们在看颜色值从0到1（从黑到白）的过程中，亮度要增加一倍，我们才会感受到明显的颜色变化（变亮一倍）。打个比方：颜色值从0.1到0.2，我们会感受到一倍的颜色变化，而从0.4到0.8我们才能感受到相同程度（变亮一倍）的颜色变化。如果还是不理解，可以参考知乎的[答案](#)）。然而，当我们谈论光的物理亮度，比如光源发射光子的数量的时候，底部（第二行）的灰阶显示出的才是物理世界真实的亮度。如底部的灰阶显示，亮度加倍时返回的也是真实的物理亮度（译注：这里亮度是指光子数量和正相关的亮度，即物理亮度，前面讨论的是人的感知亮度；物理亮度和感知亮度的区别在于，物理亮度基于光子数量，感知亮度基于人的感觉，比如第二个灰阶里亮度0.1的光子数量是0.2的二分之一），但是由于这与我们的眼睛感知亮度不完全一致（对比较暗的颜色变化更敏感），所以它看起来有差异。

因为人眼看到颜色的亮度更倾向于顶部的灰阶，监视器使用的也是一种指数关系（电压的2.2次幂），所以物理亮度通过监视器能够被映射到顶部的非线性亮度；因此看起来效果不错（译注：CRT亮度是电压的2.2次幂而人眼相当于2次幂，因此CRT这个缺陷正好能满足人的需要）。

监视器的这个非线性映射的确可以让亮度在我们眼中看起来更好，但当渲染图像时，会产生一个问题：我们在应用中配置的亮度和颜色是基于监视器所看到的，这样所有的配置实际上是非线性的亮度/颜色配置。请看下图：



点线代表线性颜色/亮度值（译注：这表示的是理想状态，Gamma为1），实线代表监视器显示的颜色。如果我们把一个点线线性的颜色翻一倍，结果就是这个值的两倍。比如，光的颜色向量代表的是暗红色。如果我们在线性空间中把它翻倍，就会变成，就像你在图中看到的那样。然而，由于我们定义的颜色仍然需要输出的监视器上，监视器上显示的实际颜色就会是。在这儿问题就出现了：当我们把理想中直线上的那个暗红色翻一倍时，在监视器上实际上亮度翻了4.5倍以上！

直到现在，我们还一直假设我们所有的工作都是在线性空间中进行的（译注：Gamma为1），但最终还是要把所有的颜色输出到监视器上，所以我们配置的所有颜色和光照变量从物理角度来看都是不正确的，在我们的监视器上很少能够正确地显示。出于这个原因，我们（以及艺术家）通常将光照值设置得比本来更亮一些（由于监视器会将其亮度显示的更暗一些），如果不是这样，在线性空间里计算出来的光照就会不正确。同时，还要记住，监视器所显示出来的图像和线性图像的最小亮度是相同的，它们最大的亮度也是相同的；只是中间亮度部分会被压暗。

因为所有中间亮度都是线性空间计算出来的（译注：计算的时候假设Gamma为1）监视器显以后，实际上都会不正确。当使用更高级的光照算法时，这个问题会变得越来越明显，你可以看看下图：



Gamma校正

Gamma校正(Gamma Correction)的思路是在最终的颜色输出上应用监视器Gamma的倒数。回头看前面的Gamma曲线图，你会有一个短划线，它是监视器Gamma曲线的翻转曲线。我们在颜色显示到监视器的时候把每个颜色输出都加上这个翻转的Gamma曲线，这样应用了监视器Gamma以后最终的颜色将会变为线性的。我们所得到的中间色调就会更亮，所以虽然监视器使它们变暗，但是我们又将其平衡回来了。

我们来看另一个例子。还是那个暗红色。在将颜色显示到监视器之前，我们先对颜色应用Gamma校正曲线。线性的颜色显示在监视器上相当于降低了次幂的亮度，所以倒数就是次幂。Gamma校正后的暗红色就会成为。校正后的颜色接着被发送给监视器，最终显示出来的颜色是。你会发现使用了Gamma校正，监视器最终会显示出我们在应用中设置的那种线性的颜色。

2.2通常是大多数显示设备的大概平均gamma值。基于gamma2.2的颜色空间叫做sRGB颜色空间。每个监视器的gamma曲线都有所不同，但是gamma2.2在大多数监视器上表现都不错。出于这个原因，游戏经常都会为玩家提供改变游戏gamma设置的选项，以适应每个监视器（译注：现在Gamma2.2相当于一个标准，后文中你会看到。但现在你可能会问，前面不是说Gamma2.2看起来不是正好适合人眼么，为何还需要校正。这是因为你在程序中设置的颜色，比如光照都是基于线性Gamma，即Gamma1，所以你理想中的亮度和实际表达出的不一样，如果要表达出你理想中的亮度就要对这个光照进行校正）。

有两种在你的场景中应用gamma校正的方式：

使用OpenGL内建的sRGB帧缓冲。自己在像素着色器中进行gamma校正。第一个选项也许是最简单的方式，但是我们也会丧失一些控制权。开启GL_FRAMEBUFFER_SRGB，可以告诉OpenGL每个后续的绘制命令里，在颜色储存到颜色缓冲之前先校正sRGB颜色。sRGB这个颜色空间大致对应于gamma2.2，它也是家用设备的一个标准。开启GL_FRAMEBUFFER_SRGB以后，每次像素着色器运行后续帧缓冲，OpenGL将自动执行gamma校正，包括默认帧缓冲。

开启GL_FRAMEBUFFER_SRGB简单的调用glEnable就行：

```
glEnable(GL_FRAMEBUFFER_SRGB);
```

自此，你渲染的图像就被进行gamma校正处理，你不需要做任何事情硬件就帮你处理了。有时候，你应该记得这个建议：gamma校正将把线性颜色空间转变为非线性空间，所以在最后一步进行gamma校正是极其重要的。如果你在最后输出之前就进行gamma校正，所有的后续操作都是在操作不正确的颜色值。例如，如果你使用多个帧缓冲，你可能打算让两个帧缓冲之间传递的中间结果仍然保持线性空间颜色，只是给发送给监视器的那个帧缓冲应用gamma校正。

第二个方法稍微复杂点，但同时也是我们对gamma操作有完全的控制权。我们在每个相关像素着色器运行的最后应用gamma校正，所以在发送到帧缓冲前，颜色就被校正了。

```
void main()
{
    // do super fancy lighting
    [...]
    // apply gamma correction
    float gamma = 2.2;
    fragColor.rgb = pow(fragColor.rgb, vec3(1.0/gamma));
}
```

最后一行代码，将fragColor的每个颜色元素应用有一个 $1.0/\text{gamma}$ 的幂运算，校正像素着色器的颜色输出。

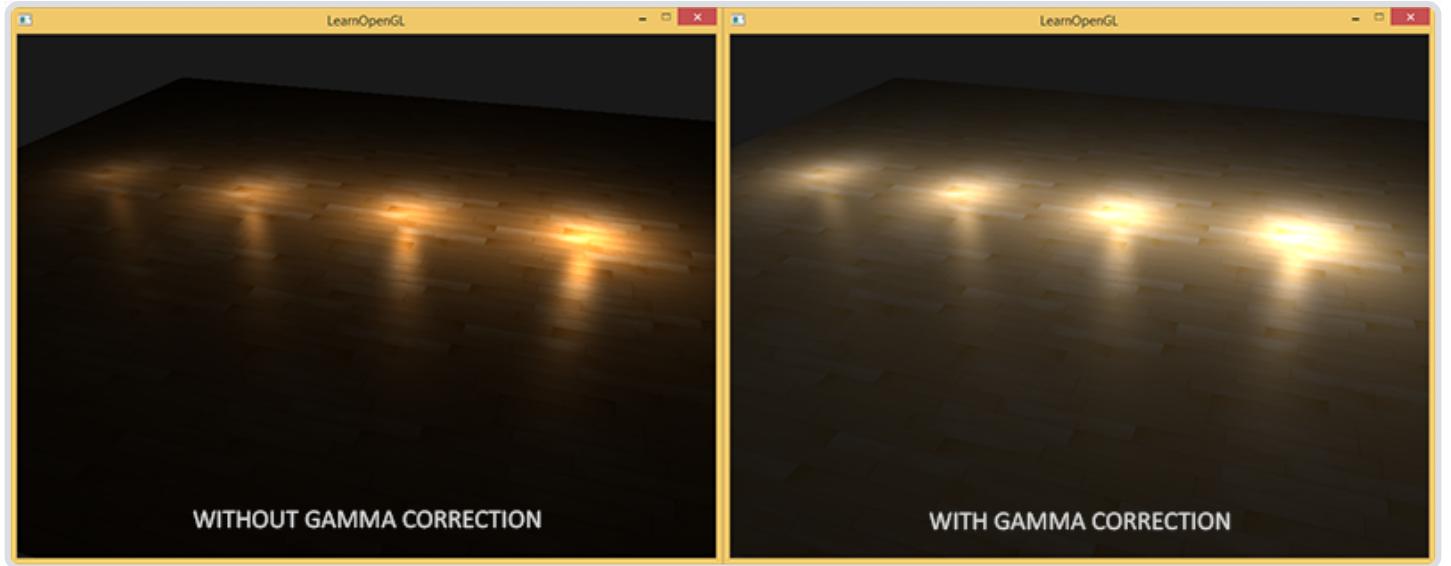
这个方法有个问题就是为了保持一致，你必须在像素着色器里加上这个gamma校正，所以如果你有很多像素着色器，它们可能分别用于不同物体，那么你就必须在每个着色器里都加上gamma校正了。一个更简单的方案是在你的渲染循环中引入后处理阶段，在后处理四边形上应用gamma校正，这样你只要做一次就好了。

这些单行代码代表了gamma校正的实现。不太令人印象深刻，但当你进行gamma校正的时候有一些额外的事情别忘了考虑。

sRGB纹理

因为监视器总是在sRGB空间中显示应用了gamma的颜色，无论什么时候当你在计算机上绘制、编辑或者画出一个图片的时候，你所选的颜色都是根据你在监视器上看到的那种。这实际意味着所有你创建或编辑的图片并不是在线性空间，而是在sRGB空间中（译注：sRGB空间定义的gamma接近于2.2），假如在你的屏幕上对暗红色翻一倍，便是根据你所感知到的亮度进行的，并不等于将红色元素加倍。

结果就是纹理编辑者，所创建的所有纹理都是在sRGB空间中的纹理，所以如果我们在渲染应用中使用这些纹理，我们必须考虑到这点。在我们应用gamma校正之前，这不是个问题，因为纹理在sRGB空间创建和展示，同样我们还是在sRGB空间中使用，从而不必gamma校正纹理显示也没问题。然而，现在我们是把所有东西都放在线性空间中展示的，纹理颜色就会变坏，如下图展示的那样：



纹理图像实在太亮了，发生这种情况是因为，它们实际上进行了两次gamma校正！想一想，当我们基于监视器上看到的情况创建一个图像，我们就已经对颜色值进行了gamma校正，所以再次显示在监视器上就没错。由于我们在渲染中又进行了一次gamma校正，图片就实在太亮了。

为了修复这个问题，我们得确保纹理制作者是在线性空间中进行创作的。但是，由于大多数纹理制作者并不知道什么是gamma校正，并且在sRGB空间中进行创作更简单，这也许不是一个好办法。

另一个解决方案是重校，或把这些sRGB纹理在进行任何颜色值的计算前变回线性空间。我们可以这样做：

```
float gamma = 2.2;
vec3 diffuseColor = pow(texture(diffuse, texCoords).rgb, vec3(gamma));
```

为每个sRGB空间的纹理做这件事非常烦人。幸好，OpenGL给我们提供了另一个方案来解决我们的麻烦，这就是GL_SRGB和GL_SRGB_ALPHA内部纹理格式。

如果我们在OpenGL中创建了一个纹理，把它指定为以上两种sRGB纹理格式其中之一，OpenGL将自动把颜色校正到线性空间中，这样我们所使用的所有颜色值都是在线性空间中的了。我们可以这样把一个纹理指定为一个sRGB纹理：

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
```

如果你还打算在你的纹理中引入alpha元素，必究必须将纹理的内部格式指定为GL_SRGB_ALPHA。

因为不是所有纹理都是在sRGB空间中的所以当你把纹理指定为sRGB纹理时要格外小心。比如diffuse纹理，这种为物体上色的纹理几乎都是在sRGB空间中的。而为了获取光照参数的纹理，像specular贴图和法线贴图几乎都在线性空间中，所以如果你把它们也配置为sRGB纹理的话，光照就坏掉了。指定sRGB纹理时要当心。

将diffuse纹理定义为sRGB纹理之后，你将获得你所期望的视觉输出，但这次每个物体都会只进行一次gamma校正。

衰减

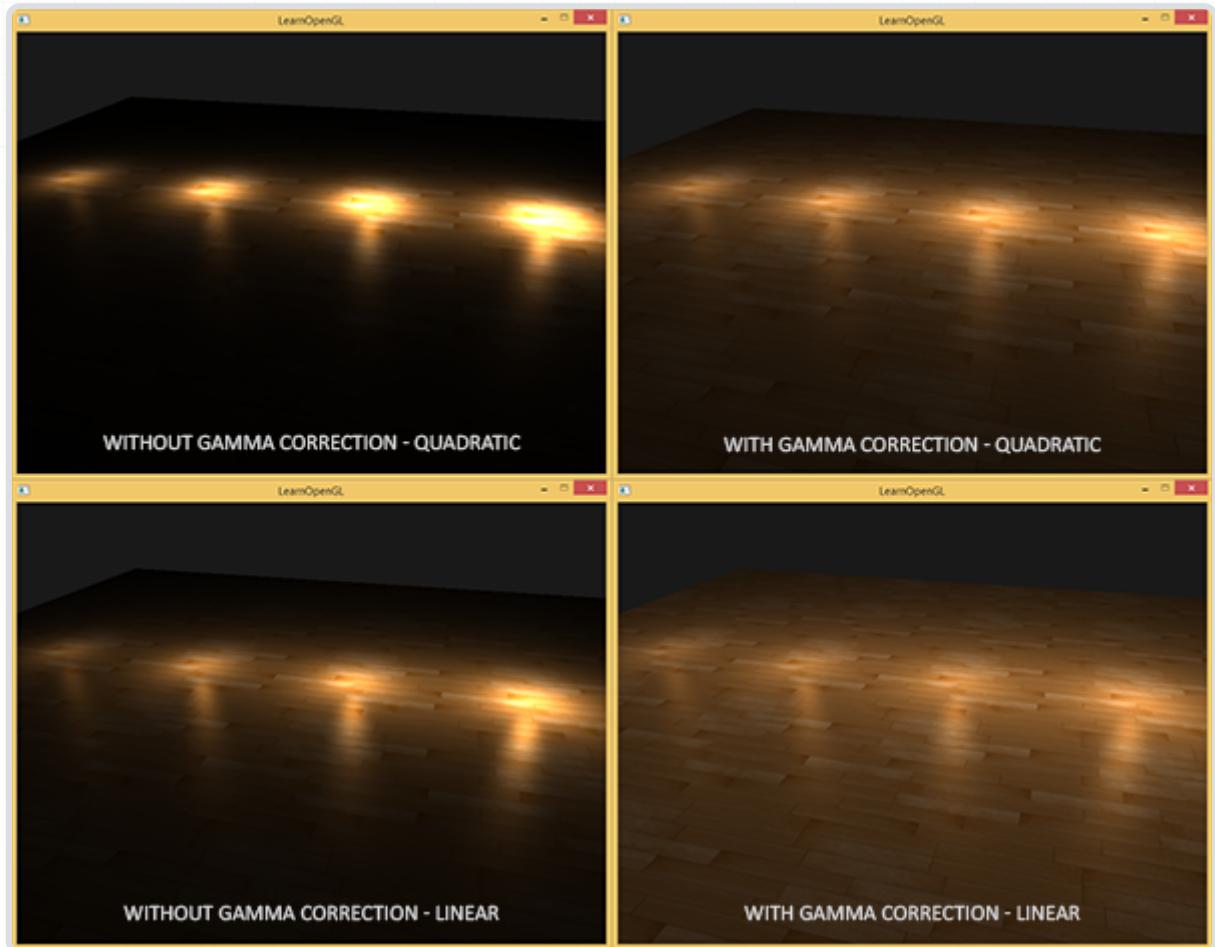
在使用了gamma校正之后，另一个不同之处是光照衰减(Attenuation)。真实的物理世界中，光照的衰减和光源的距离的平方成反比。

```
float attenuation = 1.0 / (distance * distance);
```

然而，当我们使用这个衰减公式的时候，衰减效果总是过于强烈，光只能照亮一小圈，看起来并不真实。出于这个原因，我们使用在基本光照教程中所讨论的那种衰减方程，它给了我们更大的控制权，此外我们还可以使用双曲线函数：

```
float attenuation = 1.0 / distance;
```

双曲线比使用二次函数变体在不用gamma校正的时候看起来更真实，不过但我们开启gamma校正以后线性衰减看起来太弱了，符合物理的二次函数突然出现了更好的效果。下图显示了其中的不同：



这种差异产生的原因是，光的衰减方程改变了亮度值，而且屏幕上显示出来的也不是线性空间，在监视器上效果最好的衰减方程，并不是符合物理的。想想平方衰减方程，如果我们使用这个方程，而且不进行gamma校正，显示在监视器上的衰减方程实际上将变成。若不进行gamma校正，将产生更强烈的衰减。这也解释了为什么双曲线不用gamma校正时看起来更真实，因为它实际变成了。这和物理公式是很相似的。

我们在基础光照教程中讨论的那个衰减方程在有gamma校正的场景中也仍然有用，因为它可以让我们对衰减拥有更多准确的控制权（不过，在进行gamma校正的场景中当然需要不同的参数）。

我创建的这个简单的demo场景，你可以在这里找到源码以及顶点和像素着色器。按下空格就能在有gamma校正和无gamma校正的场景进行切换，两个场景使用的是相同的纹理和衰减。这不是效果最好的demo，不过它能展示出如何应用所有这些技术。

总而言之，gamma校正使你可以在线性空间中进行操作。因为线性空间更符合物理世界，大多数物理公式现在都可以获得较好效果，比如真实的光的衰减。你的光照越真实，使用gamma校正获得漂亮的效果就越容易。这也正是为什么当引进gamma校正时，建议只去调整光照参数的原因。

附加资源

- cambridgeincolour.com: 更多关于gamma和gamma校正的内容。
- wolfire.com: David Rosen关于在渲染领域使用gamma校正的好处。
- renderwonk.com: 一些额外的实践上的思考。

2.6.3 阴影

阴影映射

原文 [Shadow Mapping](#)

作者 JoeyDeVries

翻译 [Django](#)

校对 ggy_1992

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

阴影是光线被阻挡的结果；当一个光源的光线由于其他物体的阻挡不能够达到一个物体的表面的时候，那么这个物体就在阴影中了。阴影能够使场景看起来真实得多，并且可以让观察者获得物体之间的空间位置关系。场景和物体的深度感因此能够得到极大提升，下图展示了有阴影和没有阴影的情况下的不同：



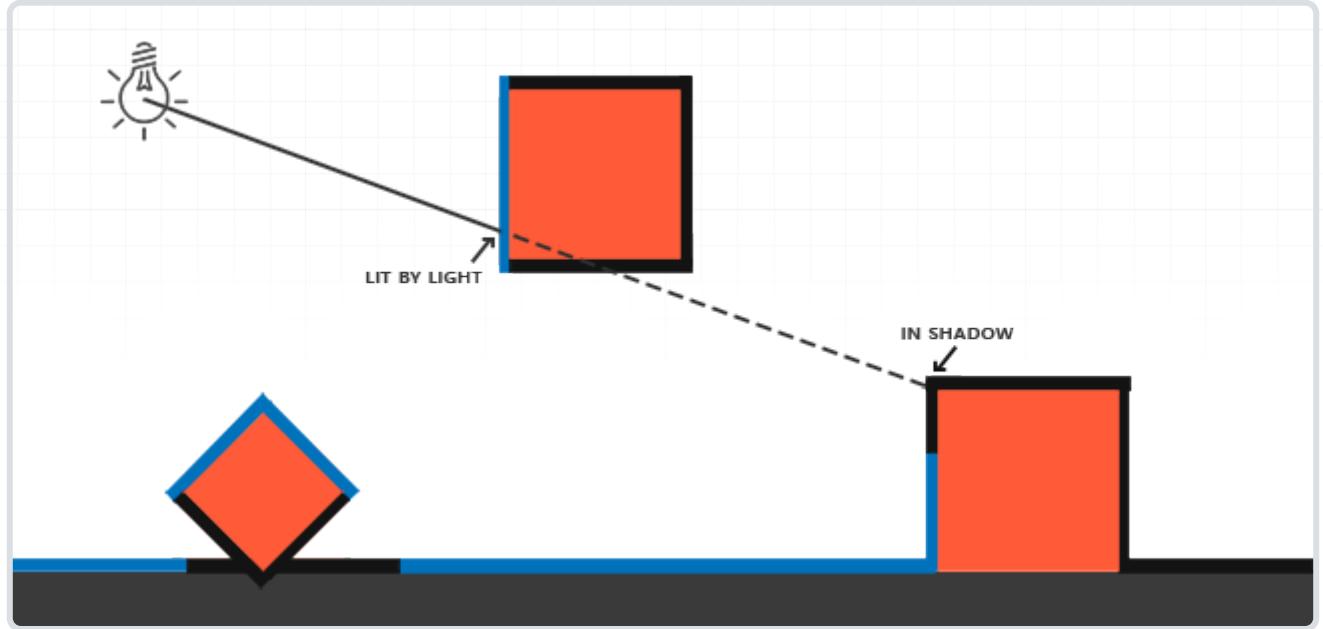
你可以看到，有阴影的时候你能更容易地区分出物体之间的位置关系，例如，当使用阴影的时候浮在地板上的立方体的事实更加清晰。

阴影还是比较不好实现的，因为当前实时渲染领域还没找到一种完美的阴影算法。目前有几种近似阴影技术，但它们都有自己的弱点和不足，这点我们必须要考虑到。

视频游戏中较多使用的一种技术是阴影贴图（shadow mapping），效果不错，而且相对容易实现。阴影贴图并不难以理解，性能也不会太低，而且非常容易扩展成更高级的算法（比如 [Omnidirectional Shadow Maps](#) 和 [Cascaded Shadow Maps](#)）。

阴影映射

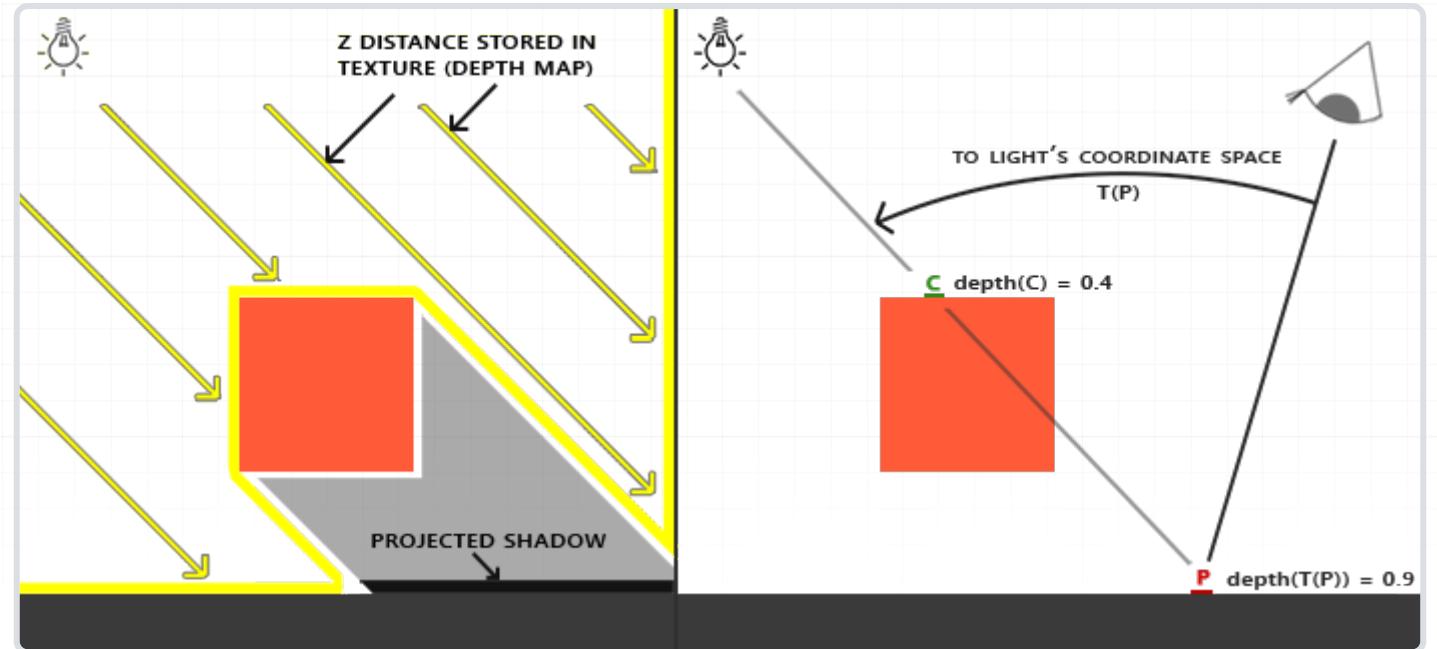
阴影映射(Shadow Mapping)背后的思路非常简单：我们以光的位置为视角进行渲染，我们能看到的东西都将被点亮，看不见的一定是在阴影之中了。假设有一个地板，在光源和它之间有一个大盒子。由于光源处向光线方向看去，可以看到这个盒子，但看不到地板的一部分，这部分就应该在阴影中了。



这里的所有蓝线代表光源可以看到的fragment。黑线代表被遮挡的fragment：它们应该渲染为带阴影的。如果我们绘制一条从光源出发，到达最右边盒子上的一个片段上的线段或射线，那么射线将先击中悬浮的盒子，随后才会到达最右侧的盒子。结果就是悬浮的盒子被照亮，而最右侧的盒子将处于阴影之中。

我们希望得到射线第一次击中的那个物体，然后用这个最近点和射线上其他点进行对比。然后我们将测试一下看看射线上的其他点是否比最近点更远，如果是的话，这个点就在阴影中。对从光源发出的射线上的成千上万个点进行遍历是个极端消耗性能的举措，实时渲染上基本不可取。我们可以采取相似举措，不用投射出光的射线。我们所使用的是非常熟悉的东西：深度缓冲。

你可能记得在[深度测试](#)教程中，在深度缓冲里的一个值是摄像机视角下，对应于一个片段的一个0到1之间的深度值。如果我们从光源的透视图来渲染场景，并把深度值的结果储存到纹理中会怎样？通过这种方式，我们就能对光源的透视图所见的最近的深度值进行采样。最终，深度值就会显示从光源的透视图下见到的第一个片段了。我们管储存在纹理中的所有这些深度值，叫做深度贴图（depth map）或阴影贴图。



左侧的图片展示了一个定向光源（所有光线都是平行的）在立方体下的表面投射的阴影。通过储存到深度贴图中的深度值，我们就能找到最近点，用以决定片段是否在阴影中。我们使用一个来自光源的视图和投影矩阵来渲染场景就能创建一个深度贴图。这个投影和视图矩阵结合在一起成为一个变换，它可以将任何三维位置转变到光源的可见坐标空间。

定向光并没有位置，因为它被规定为无穷远。然而，为了实现阴影贴图，我们得从一个光的透视图渲染场景，这样就得在光的方向的某一点上渲染场景。

在右边的图中我们显示出同样的平行光和观察者。我们渲染一个点处的片段，需要决定它是否在阴影中。我们先得使用把变换到光源的坐标空间里。既然点是从光的透视图中看到的，它的z坐标就对应于它的深度，例子中这个值是0.9。使用点在光源的坐标空间的坐标，我们可以索引深度贴图，来获得从光的视角中最近的可见深度，结果是点，最近的深度是0.4。因为索引深度贴图的结果是一个小于点的深度，我们可以断定被挡住了，它在阴影中了。

阴影映射由两个步骤组成：首先，我们渲染深度贴图，然后我们像往常一样渲染场景，使用生成的深度贴图来计算片段是否在阴影之中。听起来有点复杂，但随着我们一步一步地讲解这个技术，就能理解了。

深度贴图

第一步我们需要生成一张深度贴图(Depth Map)。深度贴图是从光的透视图里渲染的深度纹理，用它计算阴影。因为我们需要将场景的渲染结果储存到一个纹理中，我们将再次需要帧缓冲。

首先，我们要为渲染的深度贴图创建一个帧缓冲对象：

```
GLuint depthMapFBO;
 glGenFramebuffers(1, &depthMapFBO);
```

然后，创建一个2D纹理，提供给帧缓冲的深度缓冲使用：

```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;

GLuint depthMap;
 glGenTextures(1, &depthMap);
 glBindTexture(GL_TEXTURE_2D, depthMap);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
             SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

生成深度贴图不太复杂。因为我们只关心深度值，我们要把纹理格式指定为`GL_DEPTH_COMPONENT`。我们还要把纹理的高宽设置为1024：这是深度贴图的分辨率。

把我们生成的深度纹理作为帧缓冲的深度缓冲：

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

我们需要的只是在从光的透视图下渲染场景的时候深度信息，所以颜色缓冲没有用。然而，不包含颜色缓冲的帧缓冲对象是不完整的，所以我们需要显式告诉OpenGL我们不适用任何颜色数据进行渲染。我们通过将调用`glDrawBuffer`和`glReadBuffer`把读和绘制缓冲设置为`GL_NONE`来做这件事。

合理配置将深度值渲染到纹理的帧缓冲后，我们就可以开始第一步了：生成深度贴图。两个步骤的完整的渲染阶段，看起来有点像这样：

```
// 1. 首选渲染深度贴图
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. 像往常一样渲染场景，但这次使用深度贴图
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

这段代码隐去了一些细节，但它表达了阴影映射的基本思路。这里一定要记得调用`glViewport`。因为阴影贴图经常和我们原来渲染的场景（通常是窗口分辨率）有着不同的分辨率，我们需要改变视口（viewport）的参数以适应阴影贴图的尺寸。如果我们忘了更新视口参数，最后的深度贴图要么太小要么就不完整。

光源空间的变换

前面那段代码中一个不清楚的函数是`ConfigureShaderAndMatrices`。它是用来在第二个步骤确保为每个物体设置了合适的投影和视图矩阵，以及相关的模型矩阵。然而，第一个步骤中，我们从光的位置的视野下使用了不同的投影和视图矩阵来渲染的场景。

因为我们使用的是一个所有光线都平行的定向光。出于这个原因，我们将为光源使用正交投影矩阵，透视图将没有任何变形：

```
GLfloat near_plane = 1.0f, far_plane = 7.5f;
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
```

这里有个本节教程的demo场景中使用的正交投影矩阵的例子。因为投影矩阵间接决定可视区域的范围，以及哪些东西不会被裁切，你需要保证投影视锥（frustum）的大小，以包含打算在深度贴图中包含的物体。当物体和片段不在深度贴图中时，它们就不会产生阴影。

为了创建一个视图矩阵来变换每个物体，把它们变换到从光源视角可见的空间中，我们将使用`glm::lookAt`函数；这次从光源的位置看向场景中央。

```
glm::mat4 lightView = glm::lookAt(glm::vec(-2.0f, 4.0f, -1.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

二者相结合为我们提供了一个光空间的变换矩阵，它将每个世界空间坐标变换到光源处所见到的那个空间；这正是我们渲染深度贴图所需要的。

```
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

这个`lightSpaceMatrix`正是前面我们称为的那个变换矩阵。有了`lightSpaceMatrix`只要给 shader 提供光空间的投影和视图矩阵，我们就能像往常那样渲染场景了。然而，我们只关心深度值，并非所有片段计算都在我们的着色器中进行。为了提升性能，我们将使用一个与之不同但更为简单的着色器来渲染出深度贴图。

渲染至深度贴图

当我们以光的透视图进行场景渲染的时候，我们会用一个比较简单的着色器，这个着色器除了把顶点变换到光空间以外，不会做得更多了。这个简单的着色器叫做 `simpleDepthShader`，就是使用下面的这个着色器：

```
#version 330 core
layout (location = 0) in vec3 position;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(position, 1.0f);
}
```

这个顶点着色器将一个单独模型的一个顶点，使用`lightSpaceMatrix`变换到光空间中。

由于我们没有颜色缓冲，最后的片段不需要任何处理，所以我们可以简单地使用一个空片段着色器：

```
#version 330 core

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

这个空片段着色器什么也不干，运行完后，深度缓冲会被更新。我们可以取消那行的注释，来显式设置深度，但是这个（指注释掉那行之后）是更有效率的，因为底层无论如何都会默认去设置深度缓冲。

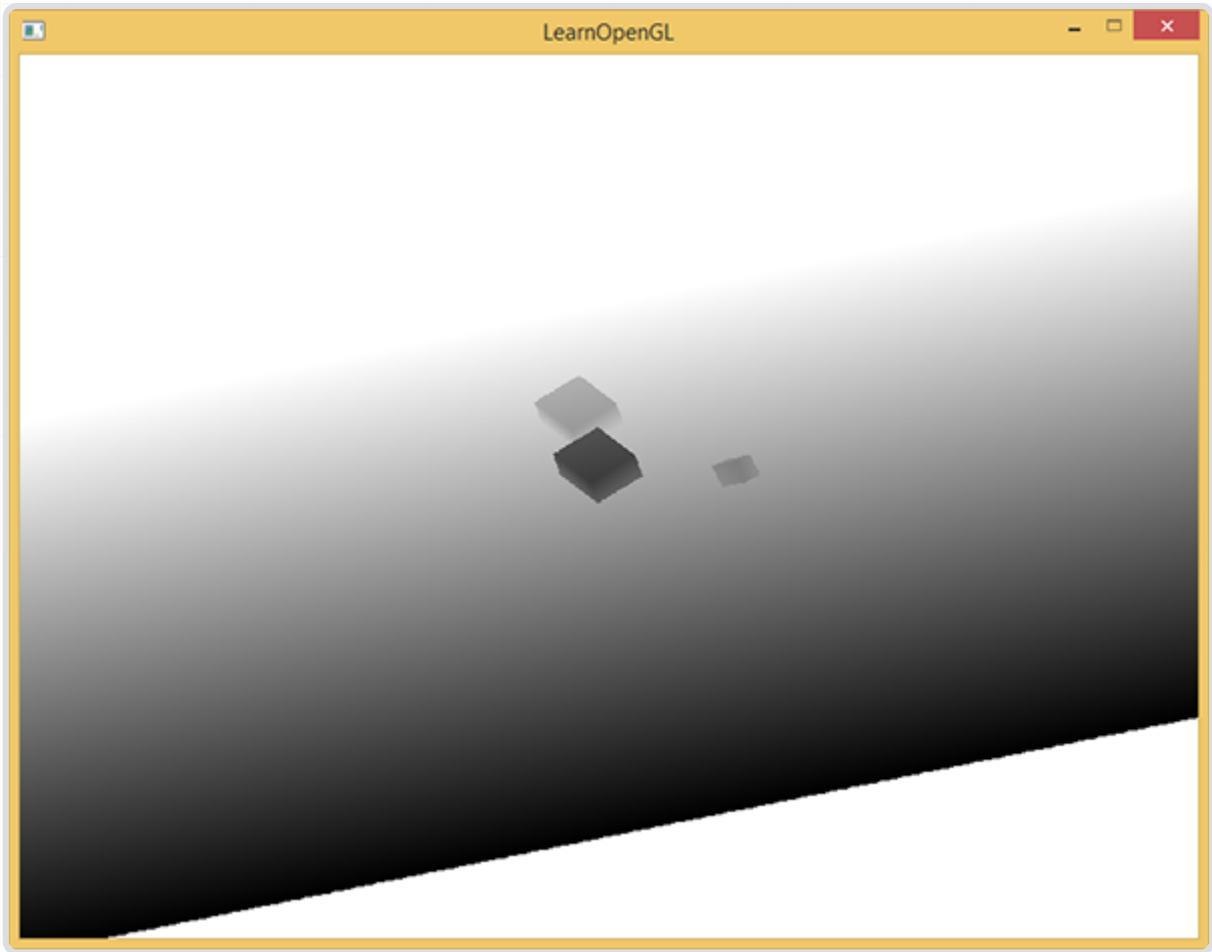
渲染深度缓冲现在成了：

```
simpleDepthShader.Use();
glUniformMatrix4fv(lightSpaceMatrixLocation, 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
 glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
 glClear(GL_DEPTH_BUFFER_BIT);
 RenderScene(simpleDepthShader);
 glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

这里的RenderScene函数的参数是一个着色器程序（shader program），它调用所有相关的绘制函数，并在需要的地方设置相应的模型矩阵。

最后，在光的透视图视角下，很完美地用每个可见片段的最近深度填充了深度缓冲。通过将这个纹理投射到一个2D四边形上（和我们在帧缓冲一节做的后处理过程类似），就能在屏幕上显示出来，我们会获得这样的东西：



将深度贴图渲染到四边形上的片段着色器：

```
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    color = vec4(vec3(depthValue), 1.0);
}
```

要注意的是当用透视投影矩阵取代正交投影矩阵来显示深度时，有一些轻微的改动，因为使用透视投影时，深度是非线性的。本节教程的最后，我们会讨论这些不同之处。

你可以在[这里](#)获得把场景渲染成深度贴图的源码。

渲染阴影

正确地生成深度贴图以后我们就可以开始生成阴影了。这段代码在片段着色器中执行，用来检验一个片段是否在阴影之中，不过我们在顶点着色器中进行光空间的变换：

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out vec2 TexCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    vs_out.FragPos = vec3(model * vec4(position, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * normal;
    vs_out.TexCoords = texCoords;
    vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos, 1.0);
}
```

这儿的新的地方是FragPosLightSpace这个输出向量。我们用同一个lightSpaceMatrix，把世界空间顶点位置转换为光空间。顶点着色器传递一个普通的经变换的世界空间顶点位置vs_out.FragPos和一个光空间的vs_out.FragPosLightSpace给片段着色器。

片段着色器使用Blinn-Phong光照模型渲染场景。我们接着计算出一个shadow值，当fragment在阴影中时是1.0，在阴影外是0.0。然后，diffuse和specular颜色会乘以这个阴影元素。由于阴影不会是全黑的（由于散射），我们把ambient分量从乘法中剔除。

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

float ShadowCalculation(vec4 fragPosLightSpace)
{
    [...]
}

void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(1.0);
    // Ambient
    vec3 ambient = 0.15 * color;
    // Diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // Specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // 计算阴影
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

    FragColor = vec4(lighting, 1.0f);
}
```

片段着色器大部分是从高级光照教程中复制过来，只不过加上了个阴影计算。我们声明一个shadowCalculation函数，用它计算阴影。片段着色器的最后，我们把diffuse和specular乘以(1-阴影元素)，这表示这个片段有多大成分不在阴影中。这个片段着色器还需要两个额外输入，一个是光空间的片段位置和第一个渲染阶段得到的深度贴图。

首先要检查一个片段是否在阴影中，把光空间片段位置转换为裁切空间的标准化设备坐标。当我们在顶点着色器输出一个裁切空间顶点位置到gl_Position时，OpenGL自动进行一个透视除法，将裁切空间坐标的范围-w到w转为-1到1，这要将x、y、z元素除以向量的w元素来实现。由于裁切空间的FragPosLightSpace并不会通过gl_Position传到片段着色器里，我们必须自己做透视除法：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    [...]
}
```

返回了片段在光空间的-1到1的范围。

当使用正交投影矩阵，顶点w元素仍保持不变，所以这一步实际上毫无意义。可是，当使用透视投影的时候就是必须的了，所以为了保证在两种投影矩阵下都有效就得留着这行。

因为来自深度贴图的深度在0到1的范围，我们也打算使用projCoords从深度贴图中去采样，所以我们将NDC坐标变换为0到1的范围。
(译者注：这里的意思是，上面的projCoords的xyz分量都是[-1,1]（下面会指出这对于远平面之类的点才成立），而为了和深度贴图的深度相比较，z分量需要变换到[0,1]；为了作为从深度贴图中采样的坐标，xy分量也需要变换到[0,1]。所以整个projCoords向量都需要变换到[0,1]范围。)

```
projCoords = projCoords * 0.5 + 0.5;
```

有了这些投影坐标，我们就能从深度贴图中采样得到0到1的结果，从第一个渲染阶段的projCoords坐标直接对应于变换过的NDC坐标。我们将得到光的位置视野下最近的深度：

```
float closestDepth = texture(shadowMap, projCoords.xy).r;
```

为了得到片段的当前深度，我们简单获取投影向量的z坐标，它等于来自光的透视线角的片段的深度。

```
float currentDepth = projCoords.z;
```

实际的对比就是简单检查currentDepth是否高于closestDepth，如果是，那么片段就在阴影中。

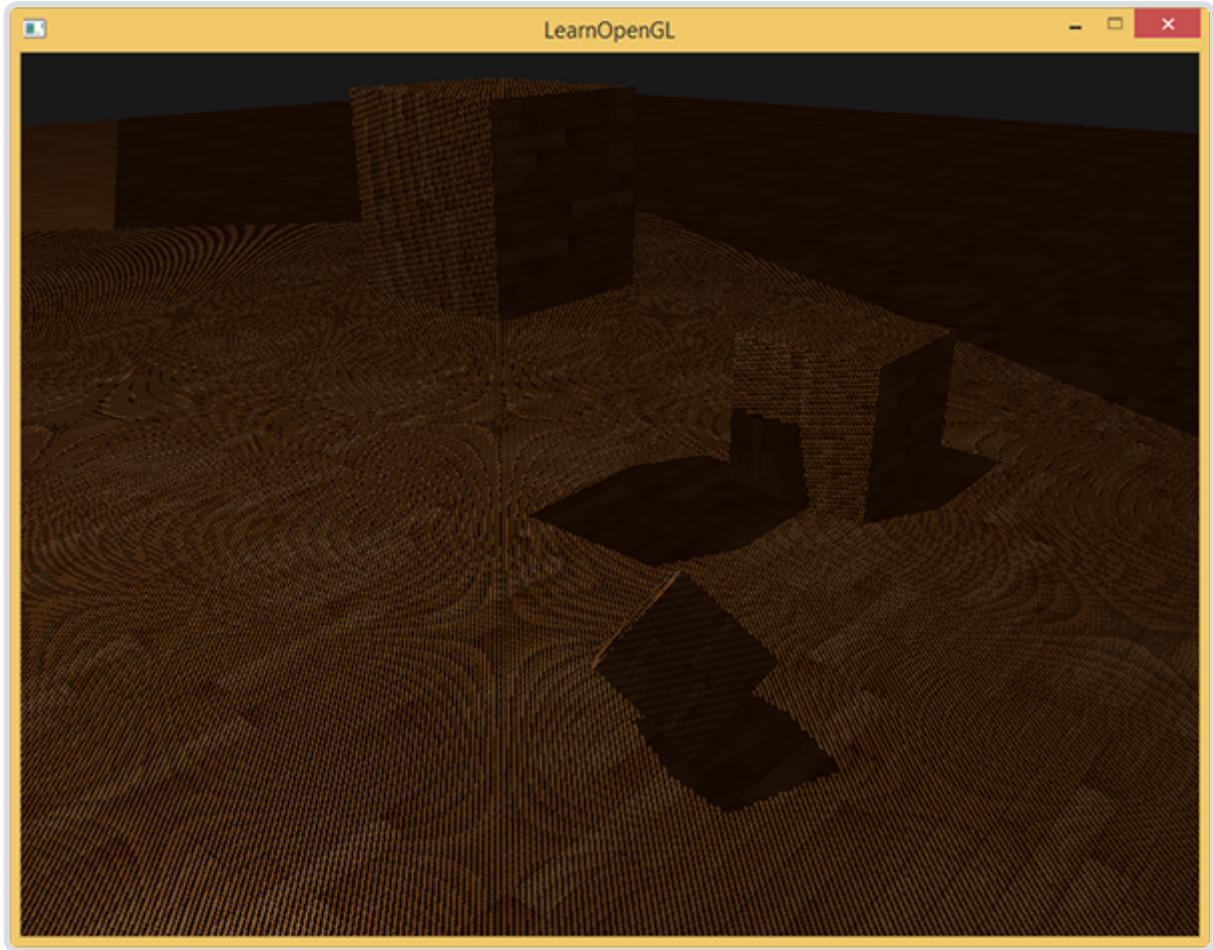
```
float shadow = currentDepth > closestDepth ? 1.0 : 0.0;
```

完整的shadowCalculation函数是这样的：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视线除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // 变换到[0,1]的范围
    projCoords = projCoords * 0.5 + 0.5;
    // 取得最近点的深度(使用[0,1]范围下的fragPosLight当坐标)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // 取得当前片段在光源视角下的深度
    float currentDepth = projCoords.z;
    // 检查当前片段是否在阴影中
    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;

    return shadow;
}
```

激活这个着色器，绑定合适的纹理，激活第二个渲染阶段默认的投影以及视图矩阵，结果如下图所示：



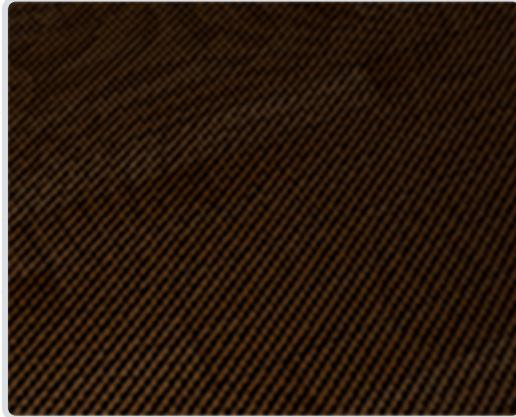
如果你做对了，你会看到地板和上有立方体的阴影。你可以从这里找到demo程序的[源码](#)。

改进阴影贴图

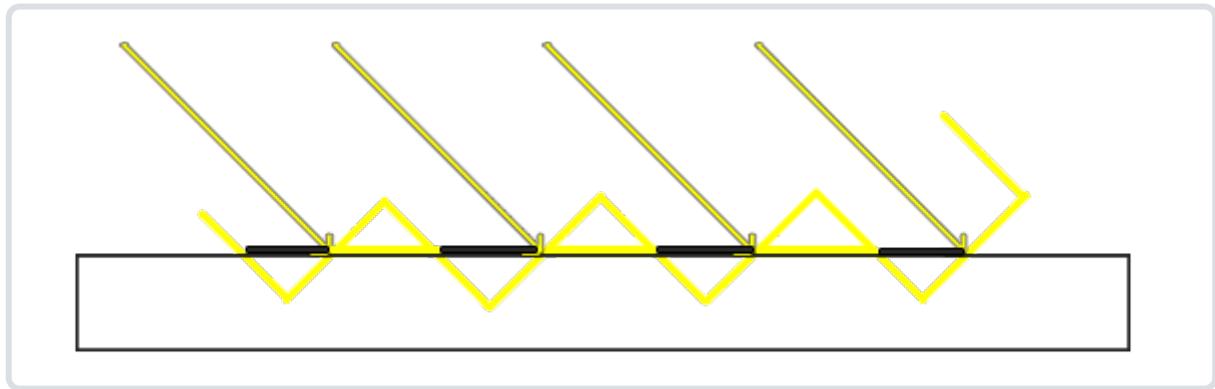
我们试图让阴影映射工作，但是你也看到了，阴影映射还是有点不真实，我们修复它才能获得更好的效果，这是下面的部分所关注的焦点。

阴影失真

前面的图片中明显有不对的地方。放大看会发现明显的线条样式：



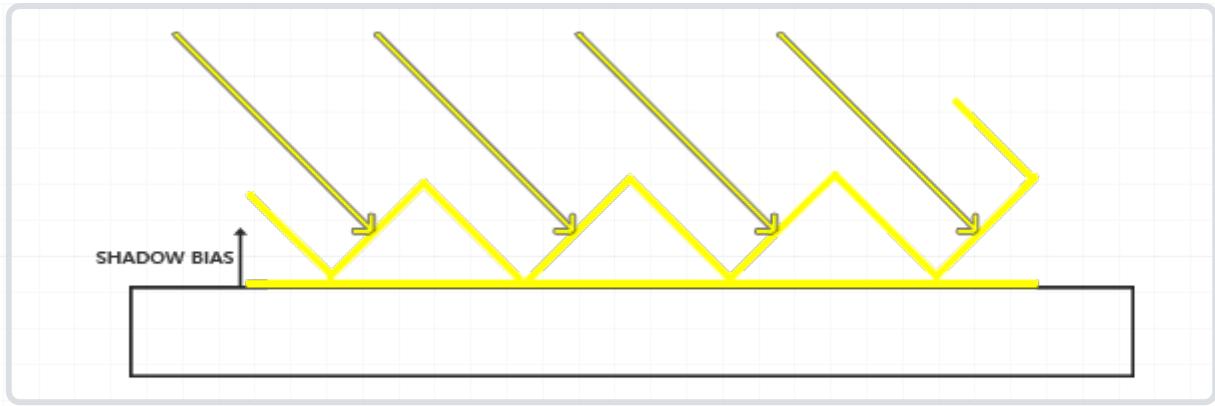
我们可以看到地板四边形渲染出很大一块交替黑线。这种阴影贴图的不真实感叫做阴影失真(Shadow Acne)，下图解释了成因：



因为阴影贴图受限于分辨率，在距离光源比较远的情况下，多个片段可能从深度贴图的同一个值中去采样。图片每个斜坡代表深度贴图一个单独的纹理像素。你可以看到，多个片段从同一个深度值进行采样。

虽然很多时候没问题，但是当光源以一个角度朝向表面的时候就会出问题，这种情况下深度贴图也是从一个角度下进行渲染的。多个片段就会从同一个斜坡的深度纹理像素中采样，有些在地板上面，有些在地板下面；这样我们所得到的阴影就有了差异。因为这个，有些片段被认为是在阴影之中，有些不在，由此产生了图片中的条纹样式。

我们可以用一个叫做阴影偏移（shadow bias）的技巧来解决这个问题，我们简单的对表面的深度（或深度贴图）应用一个偏移量，这样片段就不会被错误地认为在表面之下了。



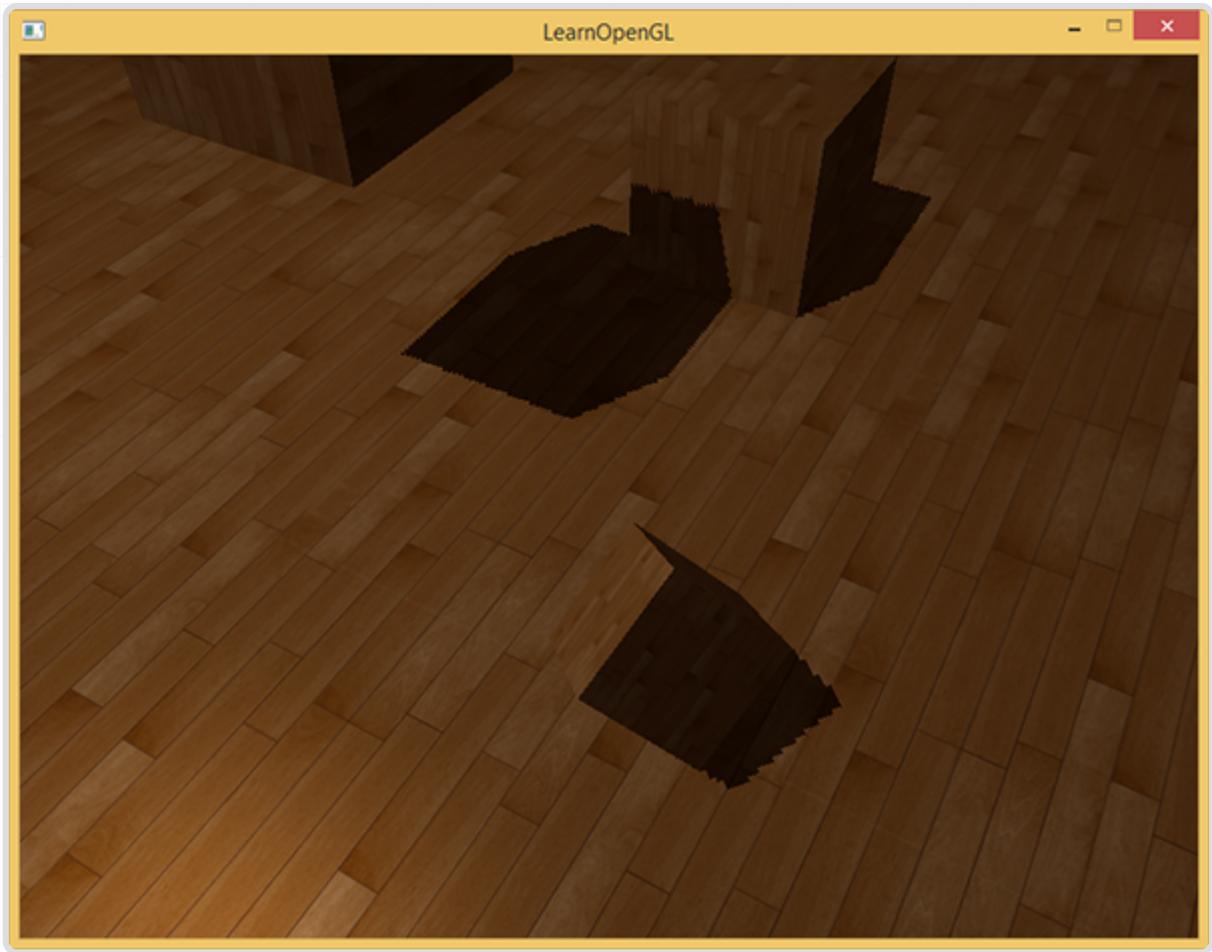
使用了偏移量后，所有采样点都获得了比表面深度更小的深度值，这样整个表面就正确地被照亮，没有任何阴影。我们可以这样实现这个偏移：

```
float bias = 0.005;
float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

一个0.005的偏移就能帮到很大的忙，但是有些表面坡度很大，仍然会产生阴影失真。有一个更加可靠的办法能够根据表面朝向光线的角度更改偏移量：使用点乘：

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

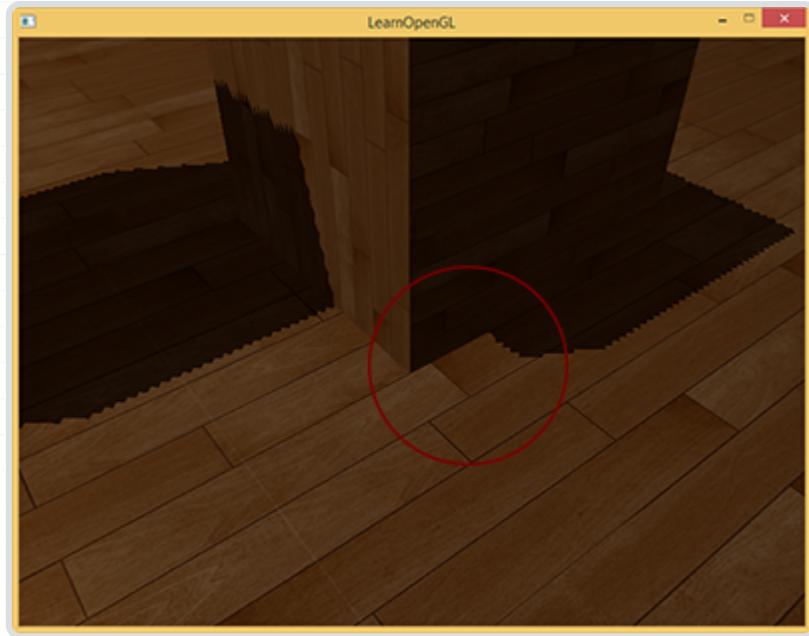
这里我们有一个偏移量的最大值0.05，和一个最小值0.005，它们是基于表面法线和光照方向的。这样像地板这样的表面几乎与光源垂直，得到的偏移就很小，而比如立方体的侧面这种表面得到的偏移就更大。下图展示了同一个场景，但使用了阴影偏移，效果的确更好：



选用正确的偏移数值，在不同的场景中需要一些像这样的轻微调校，但大多情况下，实际上就是增加偏移量直到所有失真都被移除的问题。

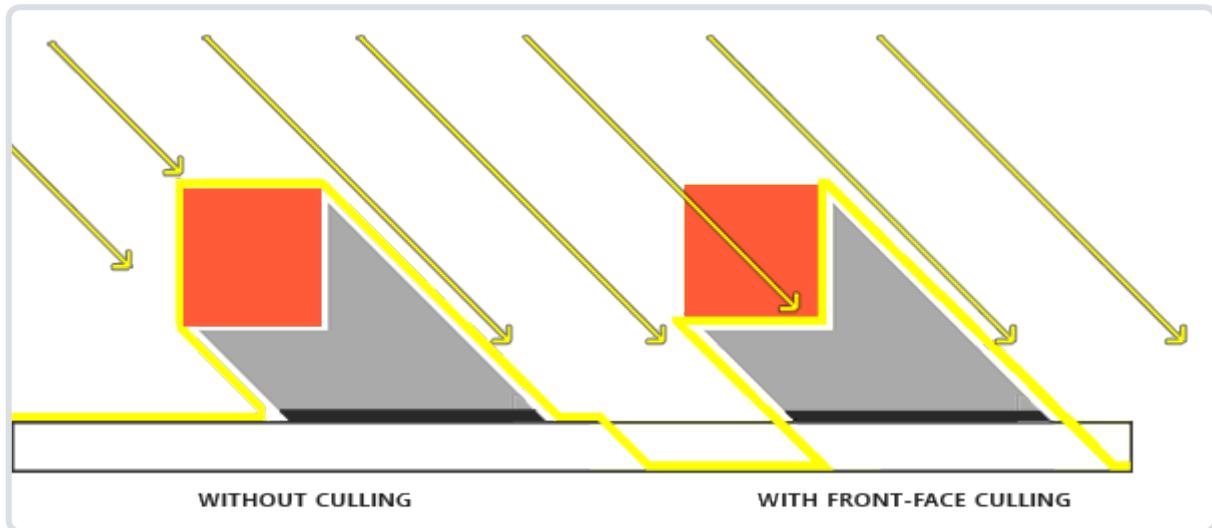
悬浮

使用阴影偏移的一个缺点是你对物体的实际深度应用了平移。偏移有可能足够大，以至于可以看出阴影相对实际物体位置的偏移，你可以从下图看到这个现象（这是一个夸张的偏移值）：



这个阴影失真叫做悬浮(Peter Panning)，因为物体看起来轻轻悬浮在表面之上（译注Peter Pan就是童话彼得潘，而panning有平移、悬浮之意，而且彼得潘是个会飞的男孩...）。我们可以使用一个叫技巧解决大部分的Peter panning问题：当渲染深度贴图时候使用正面剔除(front face culling) 你也许记得在面剔除教程中OpenGL默认是背面剔除。我们要告诉OpenGL我们要剔除正面。

因为我们只需要深度贴图的深度值，对于实体物体无论我们用它们的正面还是背面都没问题。使用背面深度不会有错误，因为阴影在物体内部有错误我们也看不见。



为了修复peter游移，我们要进行正面剔除，先必须开启GL_CULL_FACE：

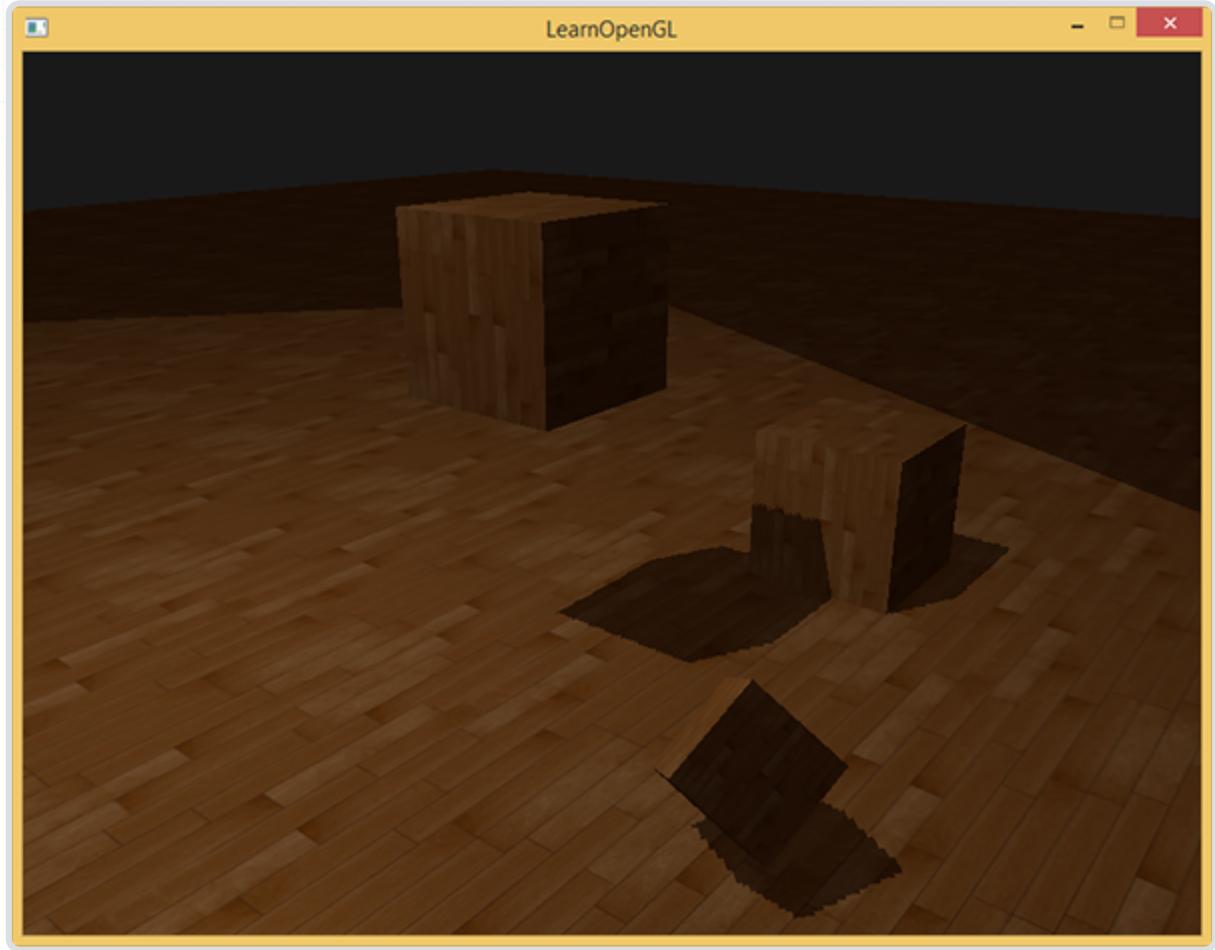
```
glCullFace(GL_FRONT);
RenderSceneToDepthMap();
glCullFace(GL_BACK); // 不要忘记设回原先的culling face
```

这十分有效地解决了peter panning的问题，但只对内部不会对外开口的实体物体有效。我们的场景中，在立方体上工作的很好，但在地板上无效，因为正面剔除完全移除了地板。地面是一个单独的平面，不会被完全剔除。如果有人打算使用这个技巧解决peter panning必须考虑到只有剔除物体的正面才有意义。

另一个要考虑到的地方是接近阴影的物体仍然会出现不正确的效果。必须考虑到何时使用正面剔除对物体才有意义。不过使用普通的偏移值通常就能避免peter panning。

采样过多

无论你喜不喜欢还有一个视觉差异，就是光的视锥不可见的区域一律被认为是处于阴影中，不管它真的处于阴影之中。出现这个状况是因为超出光的视锥的投影坐标比1.0大，这样采样的深度纹理就会超出他默认的0到1的范围。根据纹理环绕方式，我们将会得到不正确的深度结果，它不是基于真实的来自光源的深度值。

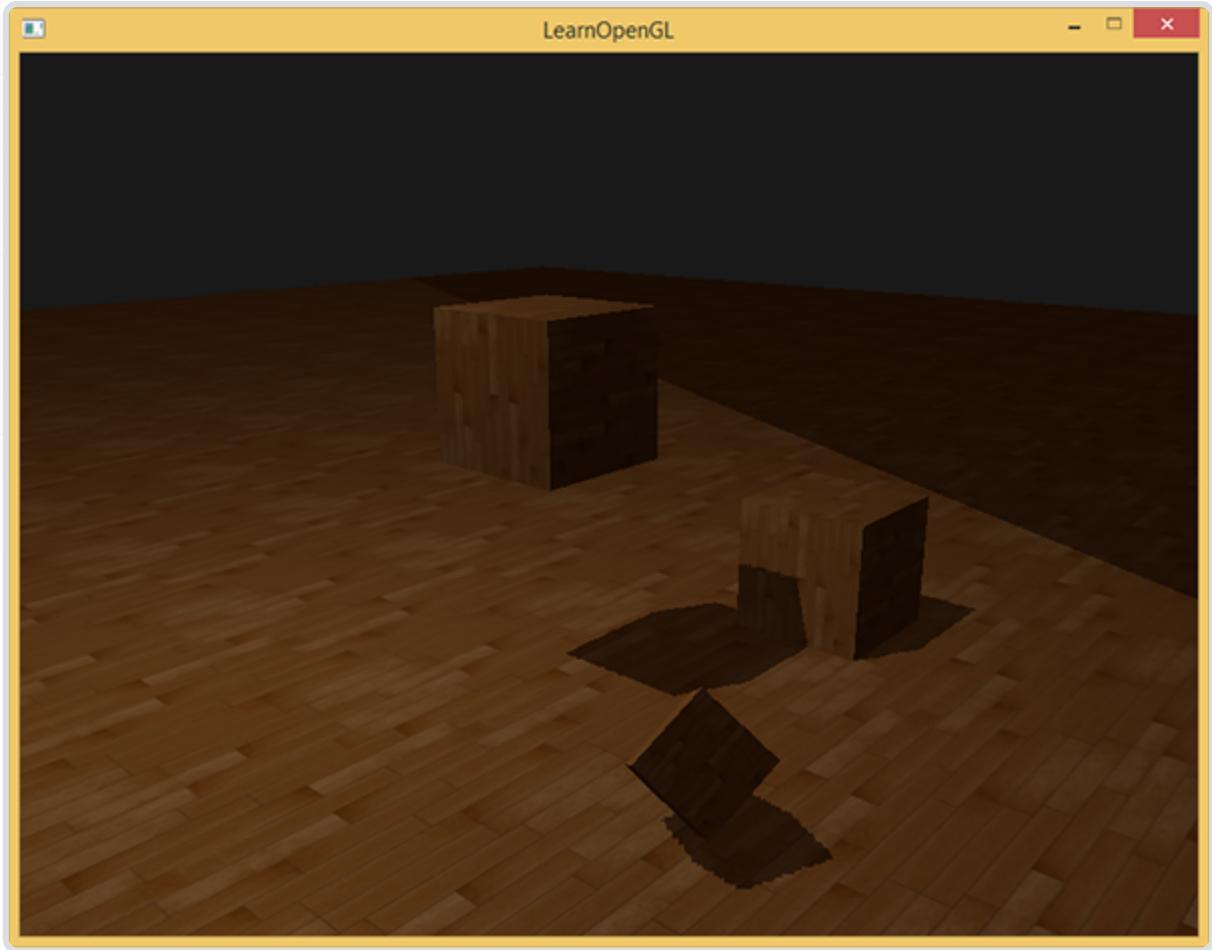


你可以在图中看到，光照有一个区域，超出该区域就成为了阴影；这个区域实际上代表着深度贴图的大小，这个贴图投影到了地板上。发生这种情况的原因是我们之前将深度贴图的环绕方式设置成了GL_REPEAT。

我们宁可让所有超出深度贴图的坐标的深度范围是1.0，这样超出的坐标将永远不在阴影之中。我们可以储存一个边框颜色，然后把深度贴图的纹理环绕选项设置为GL_CLAMP_TO_BORDER：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
GLfloat borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

现在如果我们采样深度贴图0到1坐标范围以外的区域，纹理函数总会返回一个1.0的深度值，阴影值为0.0。结果看起来会更真实：



仍有一部分是黑暗区域。那里的坐标超出了光的正交视锥的远平面。你可以看到这片黑色区域总是出现在光源视锥的极远处。

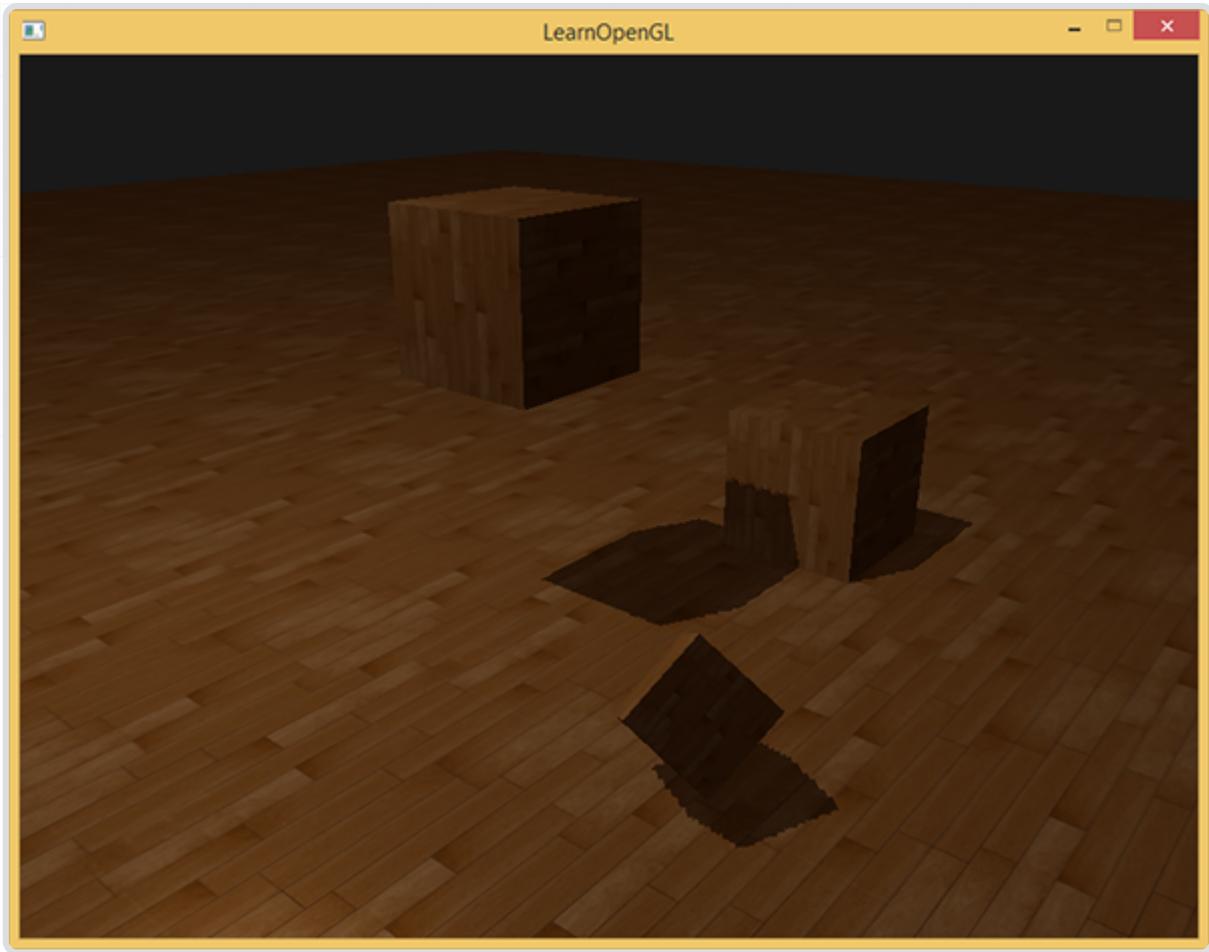
当一个点比光的远平面还要远时，它的投影坐标的z坐标大于1.0。这种情况下，`GL_CLAMP_TO_BORDER`环绕方式不起作用，因为我们把坐标的z元素和深度贴图的值进行了对比；它总是为大于1.0的z返回true。

解决这个问题也很简单，只要投影向量的z坐标大于1.0，我们就把shadow的值强制设为0.0：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    [...]
    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```

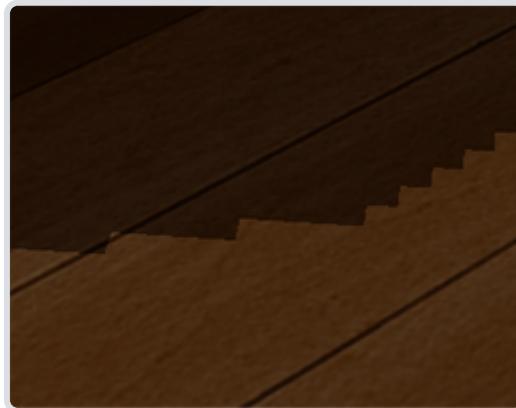
检查远平面，并将深度贴图限制为一个手工指定的边界颜色，就能解决深度贴图采样超出的问题，我们最终会得到下面我们所追求的效果：



这些结果意味着，只有在深度贴图范围以内的被投影的fragment坐标才有阴影，所以任何超出范围的都将会没有阴影。由于在游戏中通常这只发生在远处，就会比我们之前的那个明显的黑色区域效果更真实。

PCF

阴影现在已经附着到场景中了，不过这仍不是我们想要的。如果你放大看阴影，阴影映射对分辨率的依赖很快变得很明显。



因为深度贴图有一个固定的分辨率，多个片段对应于一个纹理像素。结果就是多个片段会从深度贴图的同一个深度值进行采样，这几个片段便得到的是同一个阴影，这就会产生锯齿边。

你可以通过增加深度贴图的分辨率的方式来降低锯齿块，也可以尝试尽可能的让光的视锥接近场景。

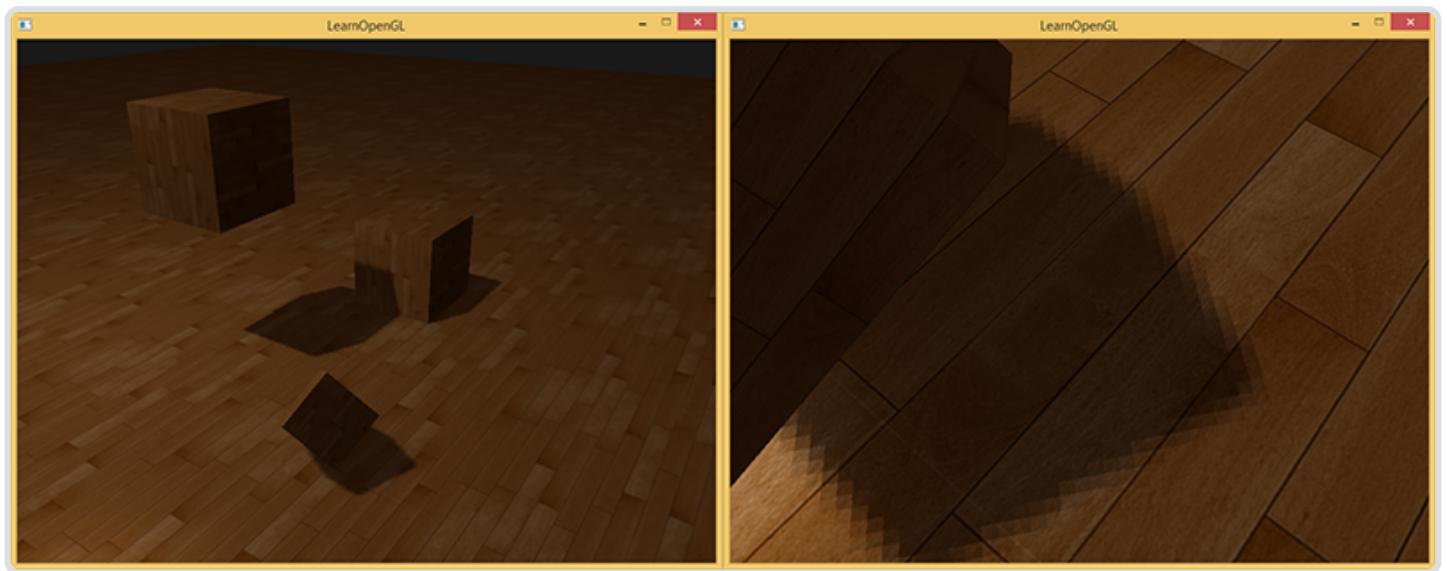
另一个（并不完整的）解决方案叫做PCF（percentage-closer filtering），这是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不在阴影中。所有的次生结果接着结合在一起，进行平均化，我们就得到了柔和阴影。

一个简单的PCF的实现是简单的从纹理像素四周对深度贴图采样，然后把结果平均起来：

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

这个textureSize返回一个给定采样器纹理的0级mipmap的vec2类型的宽和高。用1除以它返回一个单独纹理像素的大小，我们用以对纹理坐标进行偏移，确保每个新样本，来自不同的深度值。这里我们采样得到9个值，它们在投影坐标的x和y值的周围，为阴影阻挡进行测试，并最终通过样本的总数目将结果平均化。

使用更多的样本，更改texelSize变量，你就可以增加阴影的柔和程度。下面你可以看到应用了PCF的阴影：



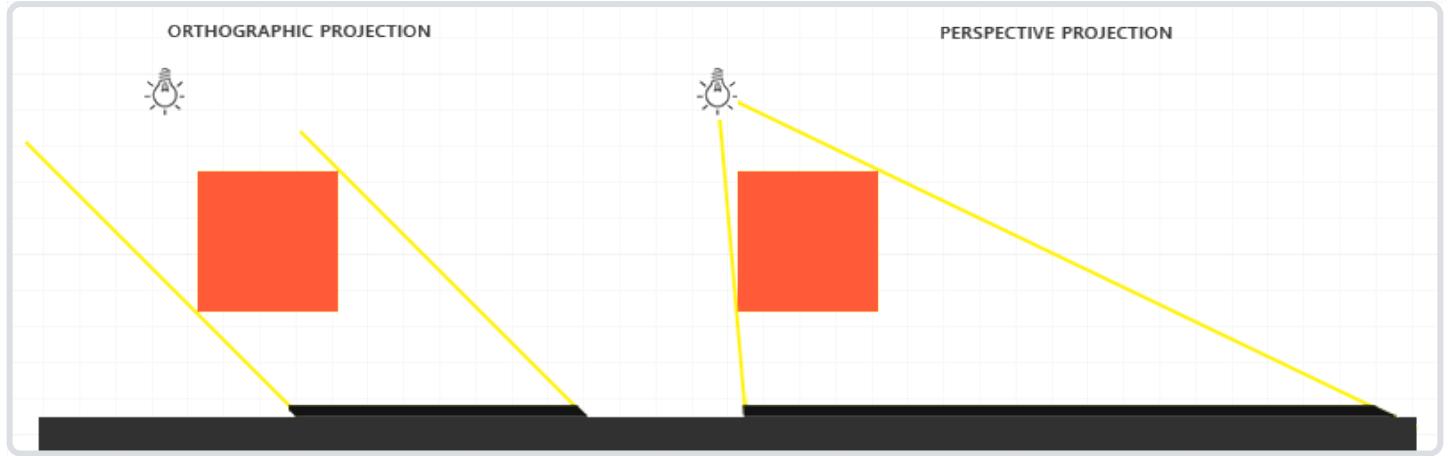
从稍微远一点的距离看去，阴影效果好多了，也不那么生硬了。如果你放大，仍会看到阴影贴图分辨率的不真实感，但通常对于大多数应用来说效果已经很好了。

你可以从[这里](#)找到这个例子的全部源码和第二个阶段的顶点和片段着色器。

实际上PCF还有更多的内容，以及很多技术要点需要考虑以提升柔和阴影的效果，但处于本章内容长度考虑，我们将留在以后讨论。

正交 vs 投影

在渲染深度贴图的时候，正交(Orthographic)和投影(Projection)矩阵之间有所不同。正交投影矩阵并不会将场景用透视图进行变形，所有视线/光线都是平行的，这使它对于定向光来说是个很好的投影矩阵。然而透视投影矩阵，会将所有顶点根据透视关系进行变形，结果因此而不同。下图展示了两种投影方式所产生的不同阴影区域：



透视投影对于光源来说更合理，不像定向光，它是有自己的位置的。透视投影因此更经常用在点光源和聚光灯上，而正交投影经常用在定向光上。

另一个细微差别是，透视投影矩阵，将深度缓冲视觉化经常会得到一个几乎全白的结果。发生这个是因为透视投影下，深度变成了非线性的深度值，它的大多数可辨范围都位于近平面附近。为了可以像使用正交投影一样合适地观察深度值，你必须先将非线性深度值转变为线性的，如我们在深度测试教程中已经讨论过的那样。

```
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D depthMap;
uniform float near_plane;
uniform float far_plane;

float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    color = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    // color = vec4(vec3(depthValue), 1.0); // orthographic
}
```

这个深度值与我们见到的用正交投影的很相似。需要注意的是，这个只适用于调试；正交或投影矩阵的深度检查仍然保持原样，因为相关的深度并没有改变。

附加资源

- [Tutorial 16 : Shadow mapping](#)：提供的类似的阴影映射教程，里面有一些额外的解释。
- [Shadow Mapping – Part 1 : ogldev](#)：提供的另一个阴影映射教程。
- [How Shadow Mapping Works](#)：的一个第三方YouTube视频教程，里面解释了阴影映射及其实现。
- [Common Techniques to Improve Shadow Depth Maps](#)：微软的一篇好文章，其中理出了很多提升阴影贴图质量的技术。

点光源阴影

原文 [Point Shadows](#)

作者 JoeyDeVries

翻译 [Django](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

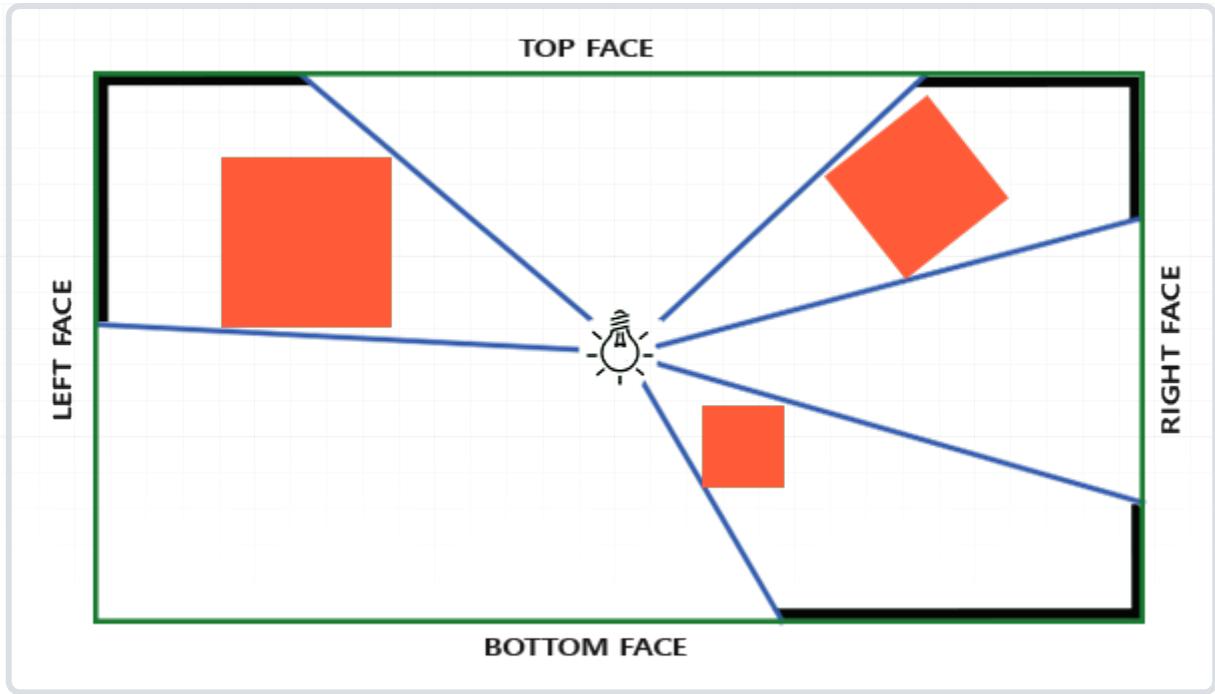
上个教程我们学到了如何使用阴影映射技术创建动态阴影。效果不错，但它只适合定向光，因为阴影只是在单一定向光源下生成的。所以它也叫定向阴影映射，深度（阴影）贴图生成自定向光的视角。

本节我们的焦点是在各种方向生成动态阴影。这个技术可以适用于点光源，生成所有方向上的阴影。

这个技术叫做点光阴影，过去的名字是万向阴影贴图（omnidirectional shadow maps）技术。

本节代码基于前面的阴影映射教程，所以如果你对传统阴影映射不熟悉，还是建议先读一读阴影映射教程。算法和定向阴影映射差不多：我们从光的透视图生成一个深度贴图，基于当前fragment位置来对深度贴图采样，然后用储存的深度值和每个fragment进行对比，看看它是否在阴影中。定向阴影映射和万向阴影映射的主要不同在于深度贴图的使用上。

对于深度贴图，我们需要从一个点光源的所有渲染场景，普通2D深度贴图不能工作；如果我们使用立方体贴图会怎样？因为立方体贴图可以储存6个面的环境数据，它可以将整个场景渲染到立方体贴图的每个面上，把它们当作点光源四周的深度值来采样。



生成后的深度立方体贴图被传递到光照像素着色器，它会用一个方向向量来采样立方体贴图，从而得到当前的fragment的深度（从光的透视图）。大部分复杂的事情已经在阴影映射教程中讨论过了。算法只是在深度立方体贴图生成上稍微复杂一点。

生成深度立方体贴图

为创建一个光周围的深度值的立方体贴图，我们必须渲染场景6次：每次一个面。显然渲染场景6次需要6个不同的视图矩阵，每次把一个不同的立方体贴图面附加到帧缓冲对象上。这看起来是这样的：

```
for(int i = 0; i < 6; i++)
{
    GLuint face = GL_TEXTURE_CUBE_MAP_POSITIVE_X + i;
    glBindFramebuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, face, depthCubemap, 0);
    BindViewMatrix(lightViewMatrices[i]);
    RenderScene();
}
```

这会很耗费性能因为一个深度贴图下需要进行很多渲染调用。这个教程中我们将转而使用另外的一个小技巧来做这件事，几何着色器允许我们使用一次渲染过程来建立深度立方体贴图。

首先，我们需要创建一个立方体贴图：

```
GLuint depthCubemap;
 glGenTextures(1, &depthCubemap);
```

然后生成立方体贴图的每个面，将它们作为2D深度值纹理图像：

```
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
for (GLuint i = 0; i < 6; ++i)
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_DEPTH_COMPONENT,
                 SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

不要忘记设置合适的纹理参数：

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

正常情况下，我们把立方体贴图纹理的一个面附加到帧缓冲对象上，渲染场景6次，每次将帧缓冲的深度缓冲目标改成不同立方体贴图面。由于我们将使用一个几何着色器，它允许我们把所有面在一个过程渲染，我们可以使用glFramebufferTexture直接把立方体贴图附加成帧缓冲的深度附件：

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthCubemap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

还要记得调用glDrawBuffer和glReadBuffer：当生成一个深度立方体贴图时我们只关心深度值，所以我们必须显式告诉OpenGL这个帧缓冲对象不会渲染到一个颜色缓冲里。

万向阴影贴图有两个渲染阶段：首先我们生成深度贴图，然后我们正常使用深度贴图渲染，在场景中创建阴影。帧缓冲对象和立方体贴图的处理看起是这样的：

```
// 1. first render to depth cubemap
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. then render scene as normal with shadow mapping (using depth cubemap)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
RenderScene();
```

这个过程和默认的阴影映射一样，尽管这次我们渲染和使用的是一个立方体贴图深度纹理，而不是2D深度纹理。在我们实际开始从光的视角的所有方向渲染场景之前，我们先得计算出合适的变换矩阵。

光空间的变换

设置了帧缓冲和立方体贴图，我们需要一些方法来讲场景的所有几何体变换到6个光的方向中相应的光空间。与阴影映射教程类似，我们将需要一个光空间的变换矩阵T，但是这次是每个面都有一个。

每个光空间的变换矩阵包含了投影和视图矩阵。对于投影矩阵来说，我们将使用一个透视投影矩阵；光源代表一个空间中的点，所以透视投影矩阵更有意义。每个光空间变换矩阵使用同样的投影矩阵：

```
GLfloat aspect = (GLfloat)SHADOW_WIDTH/(GLfloat)SHADOW_HEIGHT;
GLfloat near = 1.0f;
GLfloat far = 25.0f;
glm::mat4 shadowProj = glm::perspective(glm::radians(90.0f), aspect, near, far);
```

非常重要的一点是，这里`glm::perspective`的视野参数，设置为90度。90度我们才能保证视野足够大到可以合适地填满立方体贴图的一个面，立方体贴图的所有面都能与其他面在边缘对齐。

因为投影矩阵在每个方向上并不会改变，我们可以在6个变换矩阵中重复使用。我们要为每个方向提供一个不同的视图矩阵。用`glm::lookAt`创建6个观察方向，每个都按顺序注视着立方体贴图的一个方向：右、左、上、下、近、远：

```
std::vector<glm::mat4> shadowTransforms;
shadowTransforms.push_back(shadowProj *
    glm::lookAt(lightPos, lightPos + glm::vec3(1.0, 0.0, 0.0), glm::vec3(0.0, -1.0, 0.0)));
shadowTransforms.push_back(shadowProj *
    glm::lookAt(lightPos, lightPos + glm::vec3(-1.0, 0.0, 0.0), glm::vec3(0.0, -1.0, 0.0)));
shadowTransforms.push_back(shadowProj *
    glm::lookAt(lightPos, lightPos + glm::vec3(0.0, 1.0, 0.0), glm::vec3(0.0, 0.0, 1.0)));
shadowTransforms.push_back(shadowProj *
    glm::lookAt(lightPos, lightPos + glm::vec3(0.0, -1.0, 0.0), glm::vec3(0.0, 0.0, -1.0)));
shadowTransforms.push_back(shadowProj *
    glm::lookAt(lightPos, lightPos + glm::vec3(0.0, 0.0, 1.0), glm::vec3(0.0, -1.0, 0.0)));
shadowTransforms.push_back(shadowProj *
    glm::lookAt(lightPos, lightPos + glm::vec3(0.0, 0.0, -1.0), glm::vec3(0.0, -1.0, 0.0));
```

这里我们创建了6个视图矩阵，把它们乘以投影矩阵，来得到6个不同的光空间变换矩阵。`glm::lookAt`的`target`参数是它注视的立方体贴图的面的一个方向。

这些变换矩阵发送到着色器渲染到立方体贴图里。

深度着色器

为了把值渲染到深度立方体贴图，我们将需要3个着色器：顶点和像素着色器，以及一个它们之间的几何着色器。

几何着色器是负责将所有世界空间的顶点变换到6个不同的光空间的着色器。因此顶点着色器简单地将顶点变换到世界空间，然后直接发送到几何着色器：

```
#version 330 core
layout (location = 0) in vec3 position;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(position, 1.0);
}
```

紧接着几何着色器以3个三角形的顶点作为输入，它还有一个光空间变换矩阵的uniform数组。几何着色器接下来会负责将顶点变换到光空间；这里它开始变得有趣了。

几何着色器有一个内建变量叫做`gl_Layer`，它指定发散出基本图形送到立方体贴图的哪个面。当不管它时，几何着色器就会像往常一样把它的基本图形发送到输送管道的下一阶段，但当我们更新这个变量就能控制每个基本图形将渲染到立方体贴图的哪一个面。当然这只有当我们有了一个附加到激活的帧缓冲的立方体贴图纹理才有效：

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;

uniform mat4 shadowMatrices[6];

out vec4 FragPos; // FragPos from GS (output per emitvertex)

void main()
{
    for(int face = 0; face < 6; ++face)
    {
        gl_Layer = face; // built-in variable that specifies to which face we render.
        for(int i = 0; i < 3; ++i) // for each triangle's vertices
        {
            FragPos = gl_in[i].gl_Position;
            gl_Position = shadowMatrices[face] * FragPos;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

几何着色器相对简单。我们输入一个三角形，输出总共6个三角形（6*3顶点，所以总共18个顶点）。在`main`函数中，我们遍历立方体贴图的6个面，我们每个面指定为一个输出面，把这个面的integer（整数）存到`gl_Layer`。然后，我们通过把面的光空间变换矩阵乘以`FragPos`，将每个世界空间顶点变换到相关的光空间，生成每个三角形。注意，我们还要将最后的`FragPos`变量发送给像素着色器，我们需要计算一个深度值。

上个教程，我们使用的是一个空的像素着色器，让OpenGL配置深度贴图的深度值。这次我们将计算自己的深度，这个深度就是每个fragment位置和光源位置之间的线性距离。计算自己的深度值使得之后的阴影计算更加直观。

```
#version 330 core
in vec4 FragPos;

uniform vec3 lightPos;
uniform float far_plane;

void main()
{
    // get distance between fragment and light source
    float lightDistance = length(FragPos.xyz - lightPos);

    // map to [0;1] range by dividing by far_plane
    lightDistance = lightDistance / far_plane;

    // write this as modified depth
    gl_FragDepth = lightDistance;
}
```

像素着色器将来自几何着色器的`FragPos`、光的位置向量和视锥的远平面值作为输入。这里我们把fragment和光源之间的距离，映射到0到1的范围，把它写入为fragment的深度值。

使用这些着色器渲染场景，立方体贴图附加的帧缓冲对象激活以后，你会得到一个完全填充的深度立方体贴图，以便于进行第二阶段的阴影计算。

万向阴影贴图

所有事情都做好了，是时候来渲染万向阴影(Omnidirectional Shadow)了。这个过程和定向阴影映射教程相似，尽管这次我们绑定的深度贴图是一个立方体贴图，而不是2D纹理，并且将光的投影的远平面发送给了着色器。

```
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.Use();
// ... send uniforms to shader (including light's far_plane value)
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
// ... bind other textures
RenderScene();
```

这里的renderScene函数在一个大立方体房间中渲染一些立方体，它们散落在大立方体各处，光源在场景中央。

顶点着色器和像素着色器和原来的阴影映射着色器大部分都一样：不同之处是在光空间中像素着色器不再需要一个fragment位置，现在我们可以使用一个方向向量采样深度值。

因为这个顶点着色器不再需要将他的位置向量变换到光空间，所以我们可以去掉FragPosLightSpace变量：

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

out vec2 TexCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    vs_out.FragPos = vec3(model * vec4(position, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * normal;
    vs_out.TexCoords = texCoords;
}
```

片段着色器的Blinn-Phong光照代码和我们之前阴影相乘的结尾部分一样：

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
} fs_in;

uniform sampler2D diffuseTexture;
uniform samplerCube depthMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

uniform float far_plane;

float ShadowCalculation(vec3 fragPos)
{
    [...]
}

void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.3);
    // Ambient
    vec3 ambient = 0.3 * color;
    // Diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // Specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // Calculate shadow
    float shadow = ShadowCalculation(fs_in.FragPos);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

    FragColor = vec4(lighting, 1.0f);
}
```

有一些细微的不同：光照代码一样，但我们现在有了一个uniform变量samplerCube，shadowCalculation函数用fragment的位置作为它的参数，取代了光空间的fragment位置。我们现在还要引入光的视锥的远平面值，后面我们会需要它。像素着色器的最后，我们计算出阴影元素，当fragment在阴影中时它是1.0，不在阴影中时是0.0。我们使用计算出来的阴影元素去影响光照的diffuse和specular元素。

在ShadowCalculation函数中有很多不同之处，现在是从立方体贴图中进行采样，不再使用2D纹理了。我们来一步一步的讨论一下它的内容。

我们需要做的第一件事是获取立方体贴图的深度。你可能已经从教程的立方体贴图部分想到，我们已经将深度储存为fragment和光位置之间的距离了；我们这里采用相似的处理方式：

```
float ShadowCalculation(vec3 fragPos)
{
    vec3 fragToLight = fragPos - lightPos;
    float closestDepth = texture(depthMap, fragToLight).r;
}
```

在这里，我们得到了fragment的位置与光的位置之间的不同的向量，使用这个向量作为一个方向向量去对立方体贴图进行采样。方向向量不需要是单位向量，所以无需对它进行标准化。最后的closestDepth是光源和它最接近的可见fragment之间的标准化的深度值。

closestDepth值现在在0到1的范围内了，所以我们先将其转换回0到far_plane的范围，这需要把他乘以far_plane：

```
closestDepth *= far_plane;
```

下一步我们获取当前fragment和光源之间的深度值，我们可以简单的使用fragToLight的长度来获取它，这取决于我们如何计算立方体贴图中的深度值：

```
float currentDepth = length(fragToLight);
```

返回的是和closestDepth范围相同的深度值。

现在我们可以将两个深度值对比一下，看看哪一个更接近，以此决定当前的fragment是否在阴影当中。我们还要包含一个阴影偏移，所以才能避免阴影失真，这在前面教程中已经讨论过了。

```
float bias = 0.05;
float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

完整的ShadowCalculation现在变成了这样：

```
float ShadowCalculation(vec3 fragPos)
{
    // Get vector between fragment position and light position
    vec3 fragToLight = fragPos - lightPos;
    // Use the light to fragment vector to sample from the depth map
    float closestDepth = texture(depthMap, fragToLight).r;
    // It is currently in linear range between [0,1]. Re-transform back to original value
    closestDepth *= far_plane;
    // Now get current linear depth as the length between the fragment and light position
    float currentDepth = length(fragToLight);
    // Now test for shadows
    float bias = 0.05;
    float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;

    return shadow;
}
```

有了这些着色器，我们已经能得到非常好的阴影效果了，这次从一个点光源所有周围方向上都有阴影。有一个位于场景中心的点光源，看起来会像这样：



你可以从这里找到这个[demo的源码](#)、[顶点](#)和[片段](#)着色器。

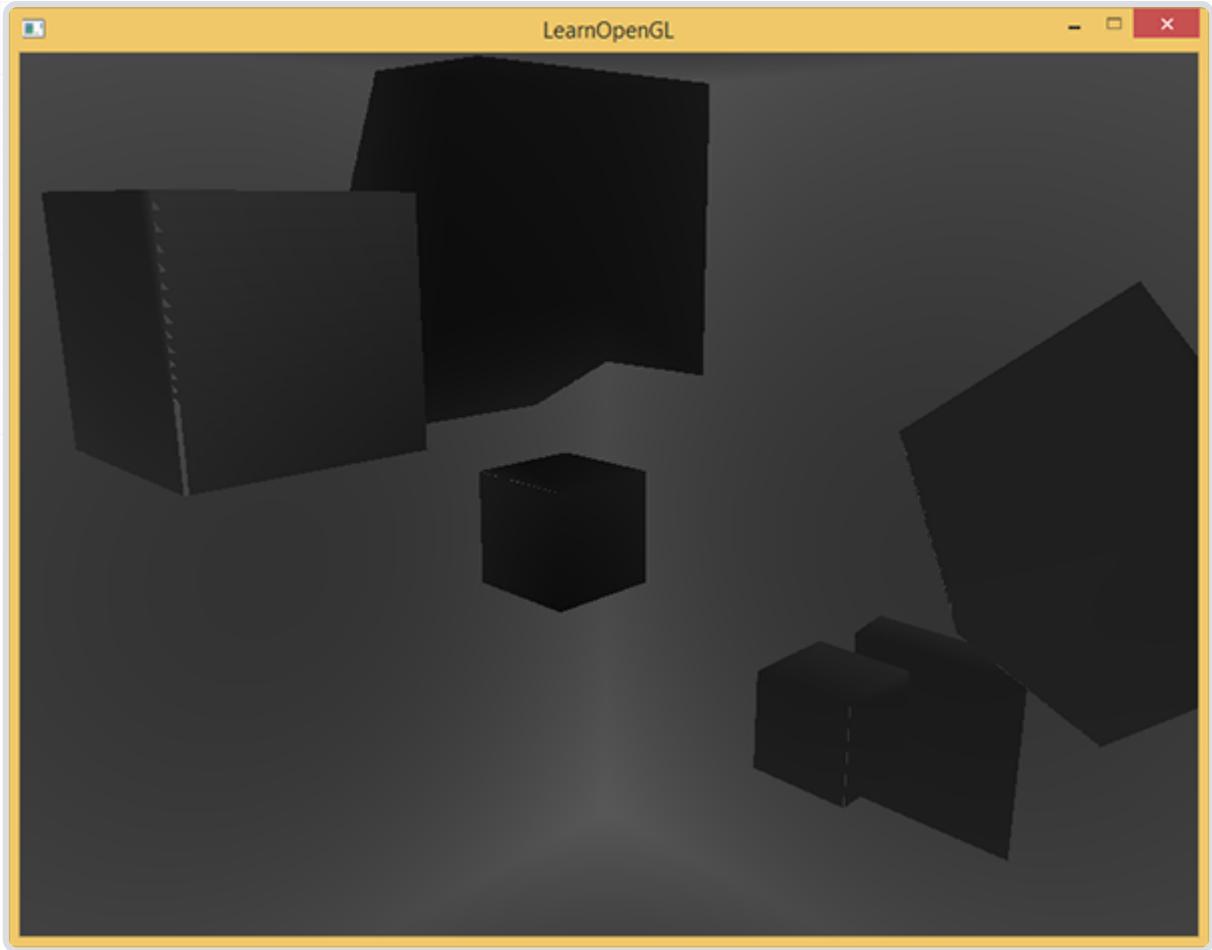
显示立方体贴图深度缓冲

如果你想我一样第一次并没有做对，那么就要进行调试排错，将深度贴图显示出来以检查其是否正确。因为我们不再用2D深度贴图纹理，深度贴图的显示不会那么显而易见。

一个简单的把深度缓冲显示出来的技巧是，在ShadowCalculation函数中计算标准化的closestDepth变量，把变量显示为：

```
FragColor = vec4(vec3(closestDepth / far_plane), 1.0);
```

结果是一个灰度场景，每个颜色代表着场景的线性深度值：



你可能也注意到了带阴影部分在墙外。如果看起来和这个差不多，你就知道深度立方体贴图生成的没错。否则你可能做错了什么，也许是closestDepth仍然还在0到far_plane的范围。

PCF

由于万向阴影贴图基于传统阴影映射的原则，它便也继承了由解析度产生的非真实感。如果你放大就会看到锯齿边了。PCF或称Percentage-closer filtering允许我们通过对fragment位置周围过滤多个样本，并对结果平均化。

如果我们用和前面教程同样的那个简单的PCF过滤器，并加入第三个维度，就是这样的：

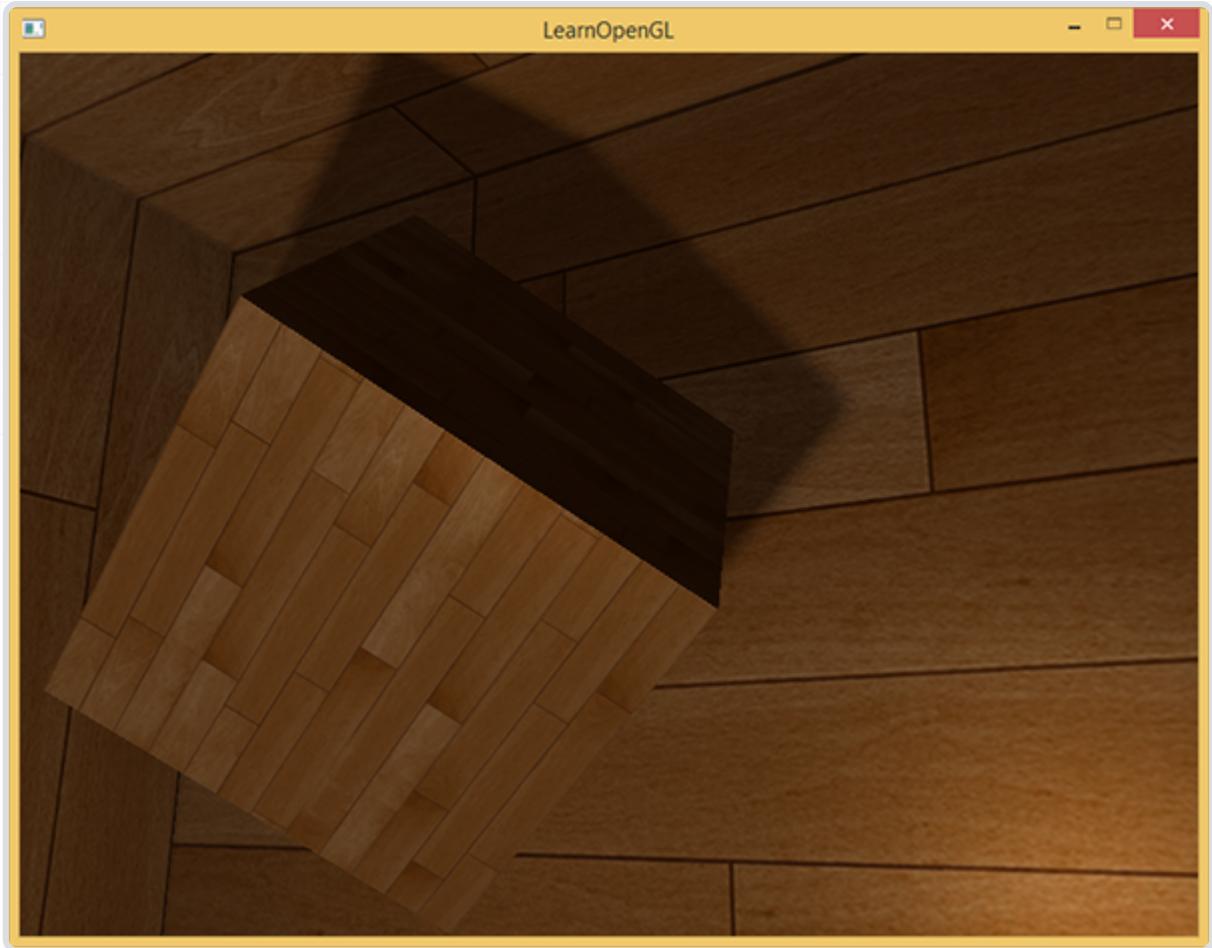
```

float shadow = 0.0;
float bias = 0.05;
float samples = 4.0;
float offset = 0.1;
for(float x = -offset; x < offset; x += offset / (samples * 0.5))
{
    for(float y = -offset; y < offset; y += offset / (samples * 0.5))
    {
        for(float z = -offset; z < offset; z += offset / (samples * 0.5))
        {
            float closestDepth = texture(depthMap, fragToLight + vec3(x, y, z)).r;
            closestDepth *= far_plane; // Undo mapping [0;1]
            if(currentDepth - bias > closestDepth)
                shadow += 1.0;
        }
    }
}
shadow /= (samples * samples * samples);

```

这段代码和我们传统的阴影映射没有多少不同。这里我们根据样本的数量动态计算了纹理偏移量，我们在三个轴向采样三次，最后对子样本进行平均化。

现在阴影看起来更加柔和平滑了，由此得到更加真实的效果：



然而，samples设置为4.0，每个fragment我们会得到总共64个样本，这太多了！

大多数这些采样都是多余的，与其在原始方向向量附近处采样，不如在采样方向向量的垂直方向进行采样更有意义。可是，没有（简单的）方式能够指出哪一个子方向是多余的，这就难了。有个技巧可以使用，用一个偏移量方向数组，它们差不多都是分开的，每一个指向完全不同的方向，剔除彼此接近的那些子方向。下面就是一个有着20个偏移方向的数组：

```
vec3 sampleOffsetDirections[20] = vec3[]
(
    vec3( 1,  1,  1), vec3( 1, -1,  1), vec3(-1, -1,  1), vec3(-1,  1,  1),
    vec3( 1,  1, -1), vec3( 1, -1, -1), vec3(-1, -1, -1), vec3(-1,  1, -1),
    vec3( 1,  1,  0), vec3( 1, -1,  0), vec3(-1, -1,  0), vec3(-1,  1,  0),
    vec3( 1,  0,  1), vec3(-1,  0,  1), vec3( 1,  0, -1), vec3(-1,  0, -1),
    vec3( 0,  1,  1), vec3( 0, -1,  1), vec3( 0, -1, -1), vec3( 0,  1, -1)
);
```

然后我们把PCF算法与从sampleOffsetDirections得到的样本数量进行适配，使用它们从立方体贴图里采样。这么做的好处是与之前的PCF算法相比，我们需要的样本数量变少了。

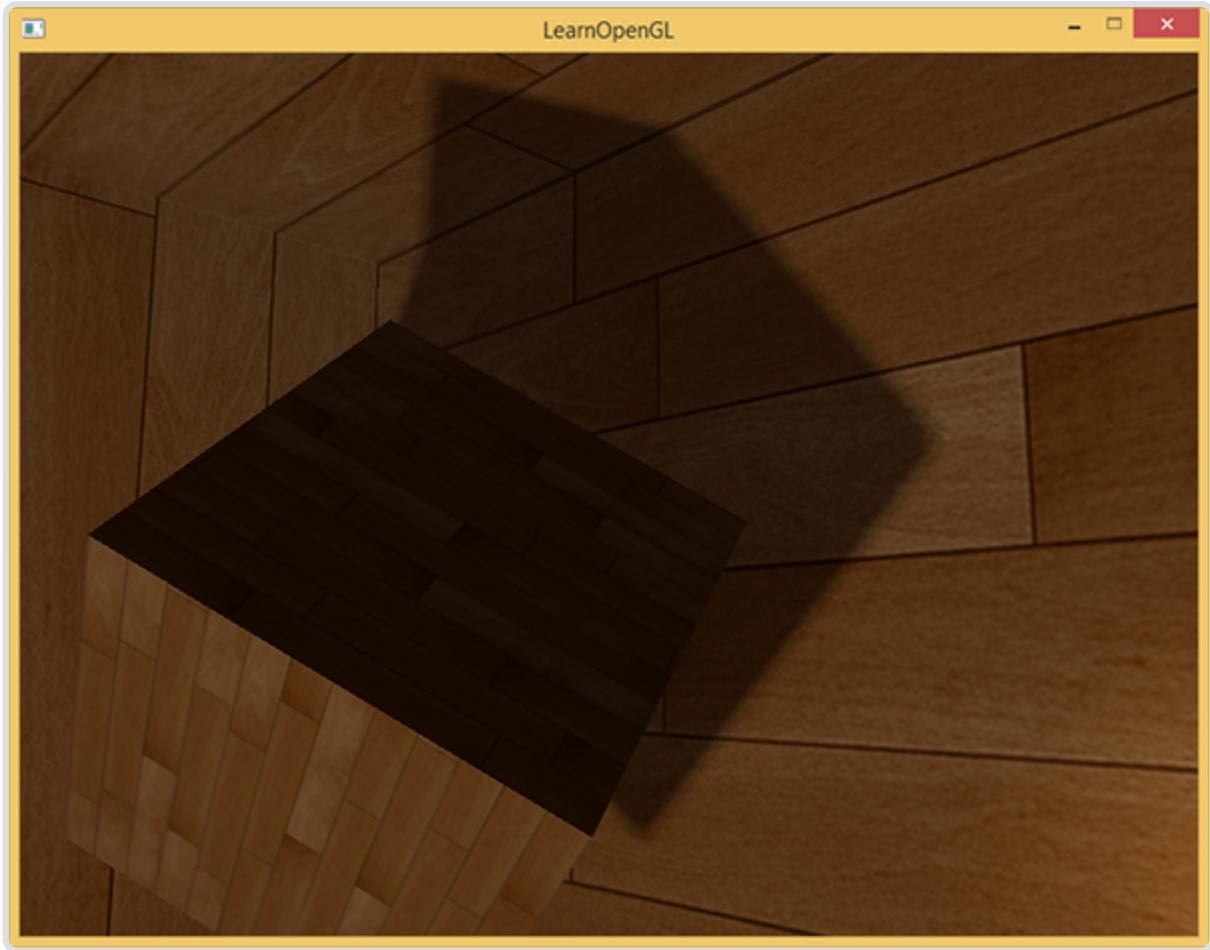
```
float shadow = 0.0;
float bias = 0.15;
int samples = 20;
float viewDistance = length(viewPos - fragPos);
float diskRadius = 0.05;
for(int i = 0; i < samples; ++i)
{
    float closestDepth = texture(depthMap, fragToLight + sampleOffsetDirections[i] * diskRadius).r;
    closestDepth *= far_plane; // Undo mapping [0;1]
    if(currentDepth - bias > closestDepth)
        shadow += 1.0;
}
shadow /= float(samples);
```

这里我们把一个偏移量添加到指定的diskRadius中，它在fragToLight方向向量周围从立方体贴图里采样。

另一个在这里可以应用的有意思的技巧是，我们可以基于观察者里一个fragment的距离来改变diskRadius；这样我们就能根据观察者的距离来增加偏移半径了，当距离更远的时候阴影更柔和，更近了就更锐利。

```
float diskRadius = (1.0 + (viewDistance / far_plane)) / 25.0;
```

PCF算法的结果如果没有变得更好，也是非常不错的，这是柔和的阴影效果：



当然了，我们添加到每个样本的bias（偏移）高度依赖于上下文，总是要根据场景进行微调的。试试这些值，看看怎样影响了场景。这里是最终版本的顶点和像素着色器。

我还要提醒一下使用几何着色器来生成深度贴图不会一定比每个面渲染场景6次更快。使用几何着色器有它自己的性能局限，在第一个阶段使用它可能获得更好的性能表现。这取决于环境的类型，以及特定的显卡驱动等等，所以如果你很关心性能，就要确保对两种方法有大致了解，然后选择对你场景来说更高效的那个。我个人还是喜欢使用几何着色器来进行阴影映射，原因很简单，因为它们使用起来更简单。

附加资源

- [Shadow Mapping for point light sources in OpenGL](#) : sunandblackcat的万向阴影映射教程。
- [Multipass Shadow Mapping With Point Lights](#) : ogldev的万向阴影映射教程。
- [Omni-directional Shadows](#) : Peter Houska的关于万向阴影映射的一组很好的ppt。

CSM

未完成

这篇教程暂时还没有完成，您可以经常来刷新看看是否有更新的进展。



2.6.4 法线贴图

原文 [Normal Mapping](#)

作者 JoeyDeVries

翻译 [Django](#)

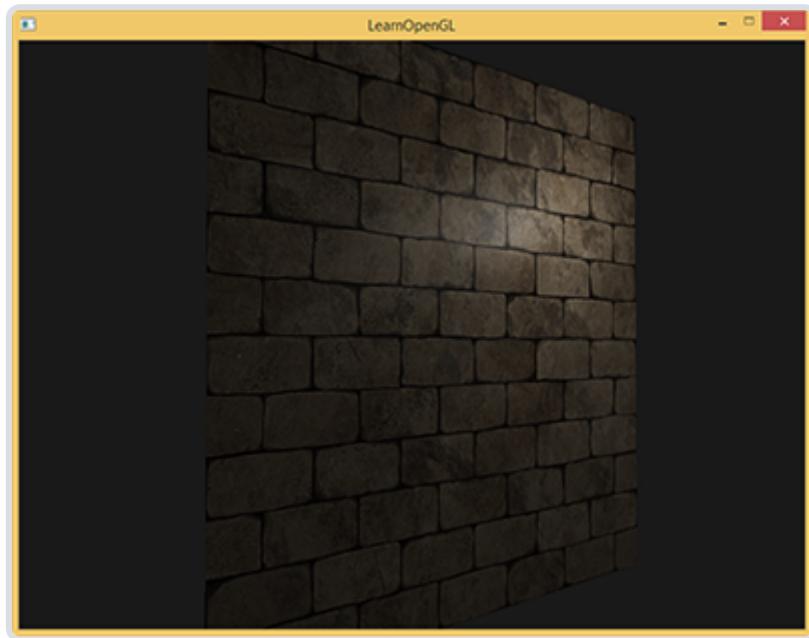
校对 [KenLee](#)

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

我们的场景中已经充满了多边形物体，其中每个都可能由成百上千平坦的三角形组成。我们以向三角形上附加纹理的方式来增加额外细节，提升真实感，隐藏多边形几何体是由无数三角形组成的事。纹理确有助益，然而当你近看它们时，这个事实便隐藏不住了。现实中的物体表面并非是平坦的，而是表现出无数（凹凸不平的）细节。

例如，砖块的表面。砖块的表面非常粗糙，显然不是完全平坦的：它包含着接缝处水泥凹痕，以及非常多的细小的空洞。如果我们在一个有光的场景中看这样一个砖块的表面，问题就出来了。下图中我们可以看到砖块纹理应用到了平坦的表面，并被一个点光源照亮。

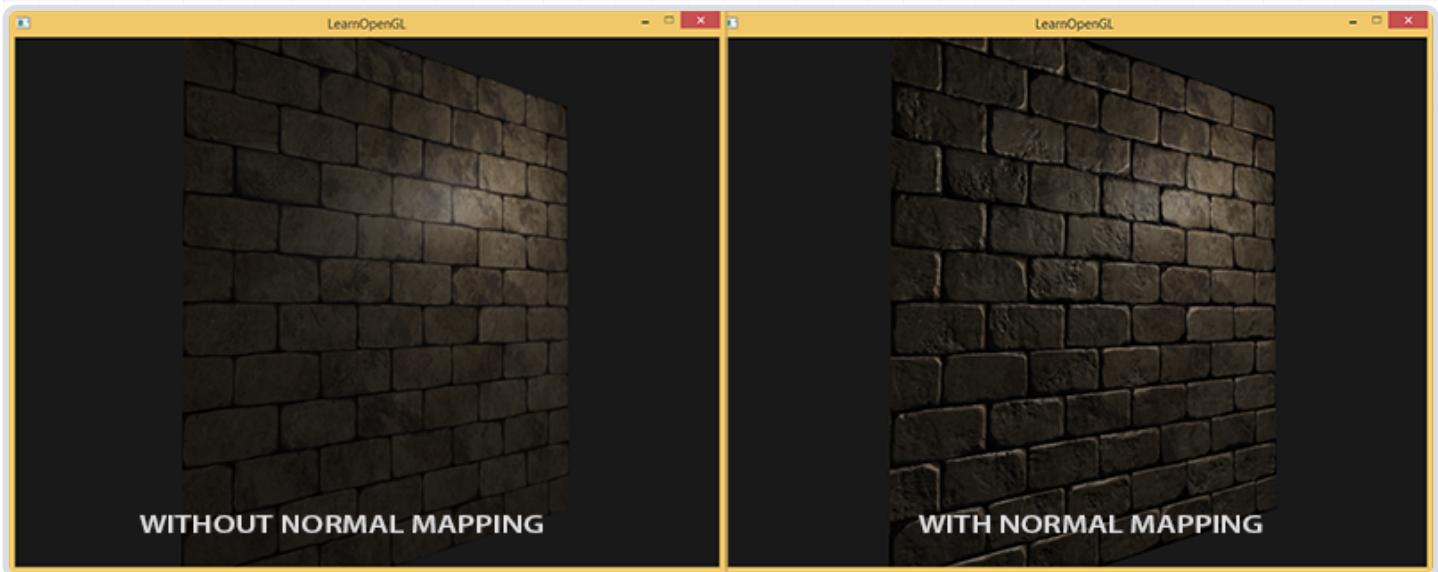


光照并没有呈现出任何裂痕和孔洞，完全忽略了砖块之间凹进去的线条；表面看起来完全就是平的。我们可以使用specular贴图根据深度或其他细节阻止部分表面被照的更亮，以此部分地解决问题，但这并不是一个好方案。我们需要的是某种可以告知光照系统给所有有关物体表面类似深度这样的细节的方式。

如果我们以光的视角来看这个问题：是什么使表面被视为完全平坦的表面来照亮？答案会是表面的法线向量。以光照算法的视角考虑的话，只有一件事决定物体的形状，这就是垂直于它的法线向量。砖块表面只有一个法线向量，表面完全根据这个法线向量被以一致的方式照亮。如果每个fragment都是用自己的不同的法线会怎样？这样我们就可以根据表面细微的细节对法线向量进行改变；这样就会获得一种表面看起来要复杂得多的幻觉：



每个fragment使用了自己的法线，我们就可以让光照相信一个表面由很多微小的（垂直于法线向量的）平面所组成，物体表面的细节将会得到极大提升。这种每个fragment使用各自的法线，替代一个面上所有fragment使用同一个法线的技术叫做法线贴图（normal mapping）或凹凸贴图（bump mapping）。应用到砖墙上，效果像这样：



你可以看到细节获得了极大提升，开销却不大。因为我们只需要改变每个fragment的法线向量，并不需要改变所有光照公式。现在我们是为每个fragment传递一个法线，不再使用插值表面法线。这样光照使表面拥有了自己的细节。

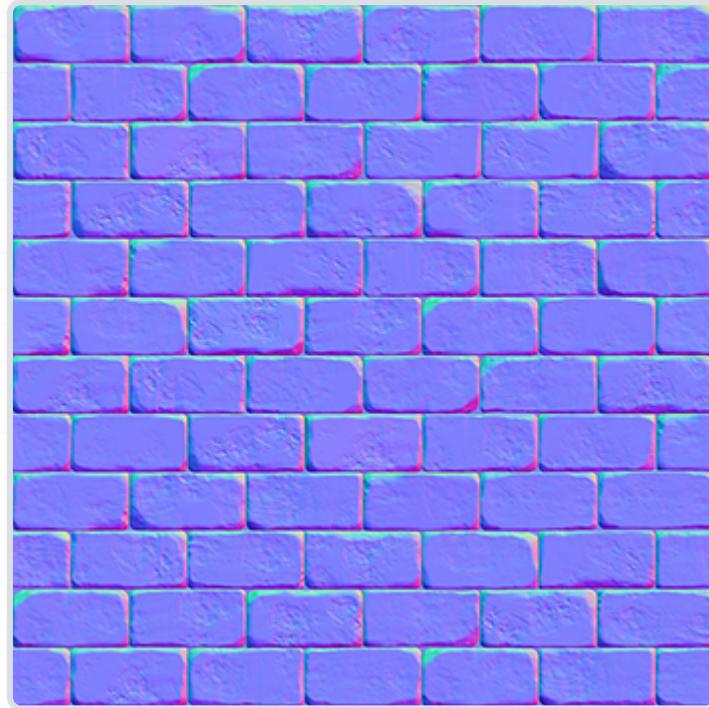
法线贴图

为使法线贴图工作，我们需要为每个fragment提供一个法线。像diffuse贴图和specular贴图一样，我们可以使用一个2D纹理来储存法线数据。2D纹理不仅可以储存颜色和光照数据，还可以储存法线向量。这样我们可以从2D纹理中采样得到特定纹理的法线向量。

由于法线向量是个几何工具，而纹理通常只用于储存颜色信息，用纹理储存法线向量不是非常直接。如果你想一想，就会知道纹理中的颜色向量用r、g、b元素代表一个3D向量。类似的我们也可以将法线向量的x、y、z元素储存到纹理中，代替颜色的r、g、b元素。法线向量的范围在-1到1之间，所以我们先要将其映射到0到1的范围：

```
vec3 rgb_normal = normal * 0.5 + 0.5; // 从 [-1,1] 转换至 [0,1]
```

将法线向量变换为像这样的RGB颜色元素，我们就能把根据表面的形状的fragment的法线保存在2D纹理中。教程开头展示的那个砖块的例子的法线贴图如下所示：



这是一种偏蓝色调的纹理（你在网上找到的几乎所有法线贴图都是这样的）。这是因为所有法线的指向都偏向z轴（0, 0, 1）这是一种偏蓝的颜色。法线向量从z轴方向也向其他方向轻微偏移，颜色也就发生了轻微变化，这样看起来便有了一种深度。例如，你可以看到在每个砖块的顶部，颜色倾向于偏绿，这是因为砖块的顶部的法线偏向于指向正y轴方向（0, 1, 0），这样它就是绿色的了。

在一个简单的朝向正z轴的平面上，我们可以用[这个diffuse纹理](#)和[这个法线贴图](#)来渲染前面部分的图片。要注意的是这个链接里的法线贴图和上面展示的那个不一样。原因是OpenGL读取的纹理的y（或V）坐标和纹理通常被创建的方式相反。链接里的法线贴图的y（或绿色）元素是相反的（你可以看到绿色现在在下边）；如果你没考虑这个，光照就不正确了（译注：如果你现在不再使用SOIL了，那就不要用链接里的那个法线贴图，这个问题是SOIL载入纹理上下颠倒所致，它也会把法线在y方向上颠倒）。加载纹理，把它们绑定到合适的纹理单元，然后使用下面的改变了的像素着色器来渲染一个平面：

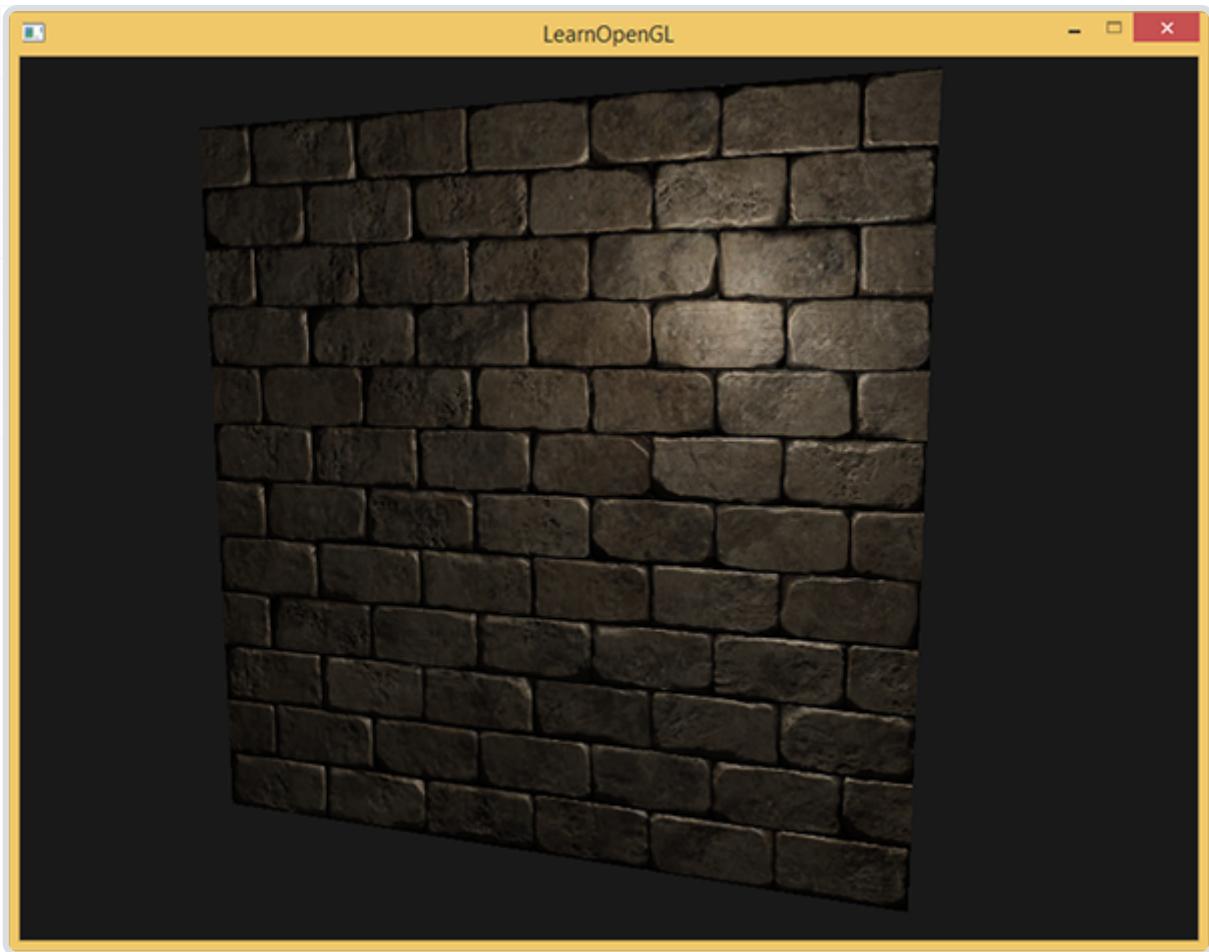
```
uniform sampler2D normalMap;

void main()
{
    // 从法线贴图范围[0,1]获取法线
    normal = texture(normalMap, fs_in.TexCoords).rgb;
    // 将法线向量转换为范围[-1,1]
    normal = normalize(normal * 2.0 - 1.0);

    [...]
    // 像往常那样处理光照
}
```

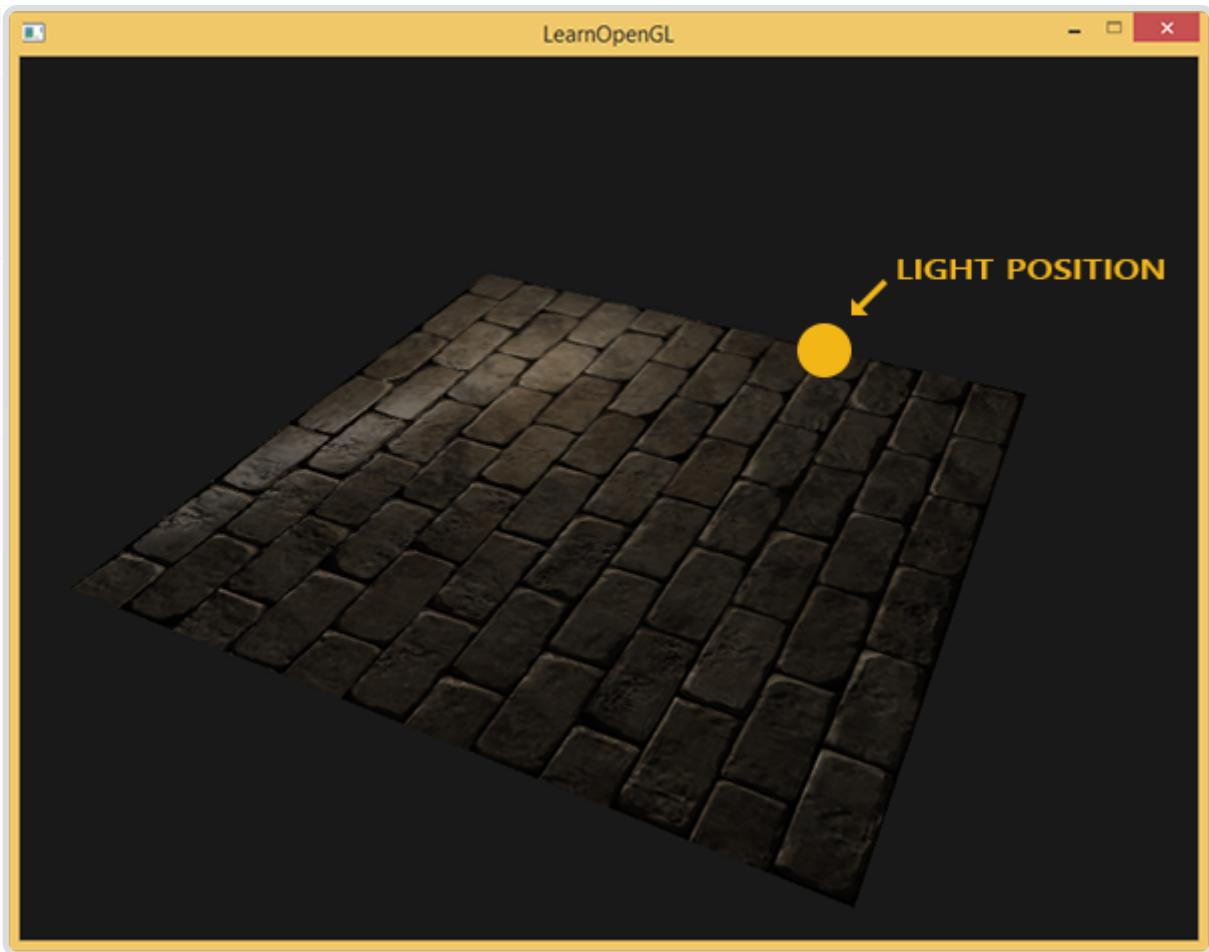
这里我们将被采样的法线颜色从0到1重新映射回-1到1，便能将RGB颜色重新处理成法线，然后使用采样出的法线向量应用于光照的计算。在例子中我们使用的是Blinn-Phong着色器。

通过慢慢随着时间慢慢移动光源，你就能明白法线贴图是什么意思了。运行这个例子你就能得到本教程开始的那个效果：

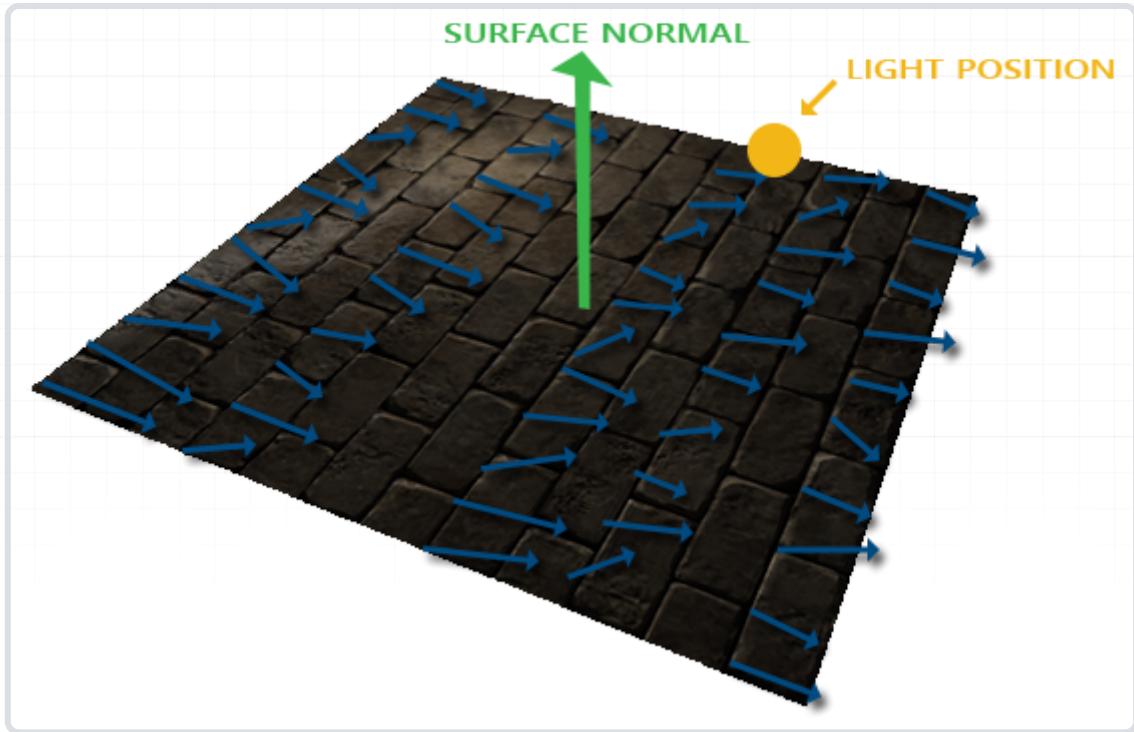


你可以在这里找到这个简单demo的源代码及其顶点和像素着色器。

然而有个问题限制了刚才讲的那种法线贴图的使用。我们使用的那个法线贴图里面的所有法线向量都是指向正z方向的。上面的例子能用，是因为那个平面的表面法线也是指向正z方向的。可是，如果我们在表面法线指向正y方向的平面上使用同一个法线贴图会发生什么？



光照看起来完全不对！发生这种情况是平面的表面法线现在指向了y，而采样得到的法线仍然指向的是z。结果就是光照仍然认为表面法线和之前朝向正z方向时一样；这样光照就不对了。下面的图片展示了这个表面上采样的法线的近似情况：



你可以看到所有法线都指向z方向，它们本该朝着表面法线指向y方向的。一个可行方案是为每个表面制作一个单独的法线贴图。如果是一个立方体的话我们就需要6个法线贴图，但是如果模型上有无数的朝向不同方向的表面，这就不可行了（译注：实际上对于复杂模型可以把朝向各个方向的法线储存在同一张贴图上，你可能看到过不只是蓝色的法线贴图，不过用那样的法线贴图有个问题是必须记住模型的起始朝向，如果模型运动了还要记录模型的变换，这是非常不方便的；此外就像作者所说的，如果把一个diffuse纹理应用在同一个物体的不同表面上，就像立方体那样的，就需要做6个法线贴图，这也不可取）。

另一个稍微有点难的解决方案是，在一个不同的坐标空间中进行光照，这个坐标空间里，法线贴图向量总是指向这个坐标空间的正z方向；所有的光照向量都相对与这个正z方向进行变换。这样我们就能始终使用同样的法线贴图，不管朝向问题。这个坐标空间叫做切线空间（tangent space）。

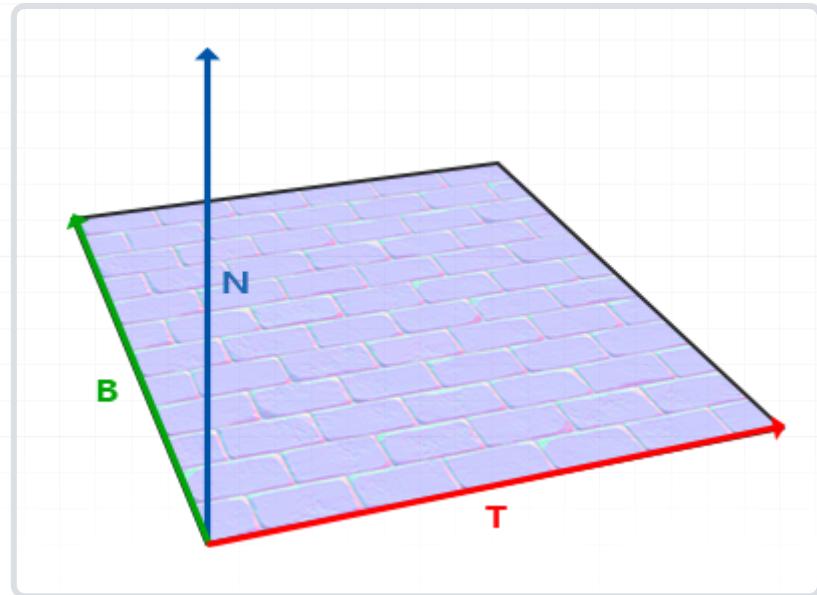
切线空间

法线贴图中的法线向量定义在切线空间中，在切线空间中，法线永远指着正z方向。切线空间是位于三角形表面之上的空间：法线相对于单个三角形的本地参考框架。它就像法线贴图向量的本地空间；它们都被定义为指向正z方向，无论最终变换到什么方向。使用一个特定的矩阵我们就能将本地/切线空间中的法线向量转成世界或视图空间下，使它们转向到最终的贴图表面的方向。

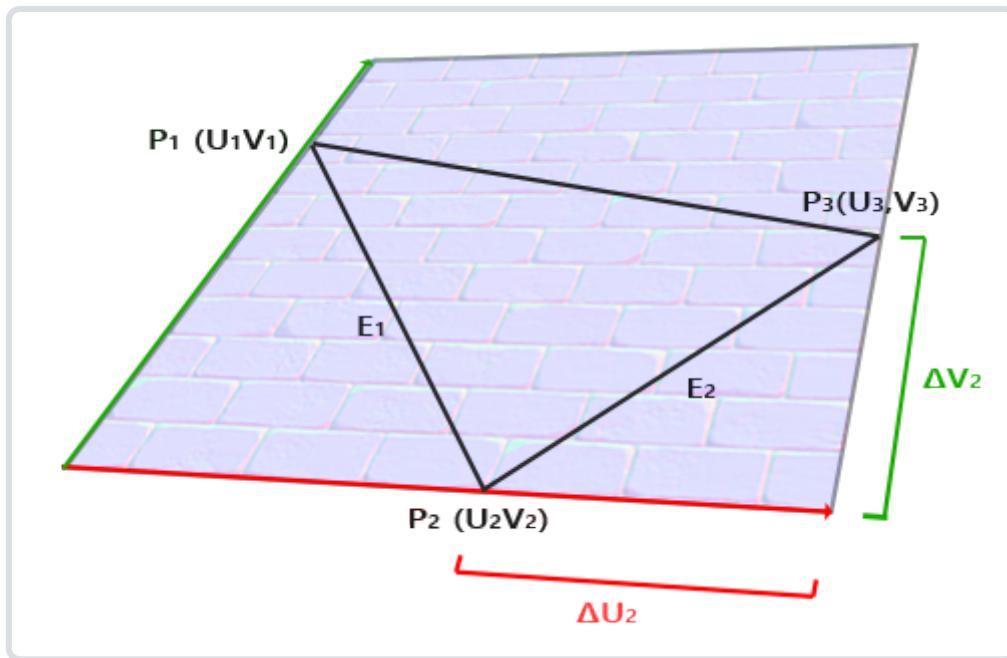
我们可以说，上个部分那个朝向正y的法线贴图错误的贴到了表面上。法线贴图被定义在切线空间中，所以一种解决问题的方式是计算出一种矩阵，把法线从切线空间变换到一个不同的空间，这样它们就能和表面法线方向对齐了：法线向量都会指向正y方向。切线空间的一大好处是我们可以为任何类型的表面计算出一个这样的矩阵，由此我们可以把切线空间的z方向和表面的法线方向对齐。

这种矩阵叫做TBN矩阵这三个字母分别代表tangent、bitangent和normal向量。这是建构这个矩阵所需的向量。要建构这样一个把切线空间转变为不同空间的变异矩阵，我们需要三个相互垂直的向量，它们沿一个表面的法线贴图对齐于：上、右、前；这和我们在[摄像机教程](#)中做的类似。

已知上向量是表面的法线向量。右和前向量是切线(Tangent)和副切线(Bitangent)向量。下面的图片展示了一个表面的三个向量：



计算出切线和副切线并不像法线向量那么容易。从图中可以看到法线贴图的切线和副切线与纹理坐标的两个方向对齐。我们就是用到这个特性计算每个表面的切线和副切线的。需要用到一些数学才能得到它们；请看下图：



注意上图中边与纹理坐标的差，构成一个三角形。与切线向量方向相同，而与副切线向量方向相同。这也就是说，所以我们可以将三角形的边与写成切线向量和副切线向量的线性组合：

我们也可以写成这样：

是两个向量位置的差，和是纹理坐标的差。然后我们得到两个未知数（切线和副切线）和两个等式。你可能想起你的代数课了，这是让我们去解和。

上面的方程允许我们把它们写成另一种格式：矩阵乘法

尝试会意一下矩阵乘法，它们确实是同一种等式。把等式写成矩阵形式的好处是，解和会因此变得很容易。两边都乘以的逆矩阵等于：

这样我们就可以解出和了。这需要我们计算出delta纹理坐标矩阵的逆矩阵。我不打算讲解计算逆矩阵的细节，但大致是把它变化为， $\frac{1}{\det A}$ 除以矩阵的行列式，再乘以它的伴随矩阵(Adjugate Matrix)。

有了最后这个等式，我们就可以用公式、三角形的两条边以及纹理坐标计算出切线向量和副切线。

如果你对这些数学内容不理解也不用担心。当你知道我们可以用一个三角形的顶点和纹理坐标（因为纹理坐标和切线向量在同一空间中）计算出切线和副切线你就已经部分地达到目的了（译注：上面的推导已经很清楚了，如果你不明白可以参考任意线性代数教材，就像作者所说的记住求得切线空间的公式也行，不过不管怎样都得理解切线空间的含义）。

手工计算切线和副切线

这个教程的demo场景中有一个简单的2D平面，它朝向正z方向。这次我们会使用切线空间来实现法线贴图，所以我们可以使平面朝向任意方向，法线贴图仍然能够工作。使用前面讨论的数学方法，我们来手工计算出表面的切线和副切线向量。

假设平面使用下面的向量建立起来（1、2、3和1、3、4，它们是两个三角形）：

```
// positions
glm::vec3 pos1(-1.0, 1.0, 0.0);
glm::vec3 pos2(-1.0, -1.0, 0.0);
glm::vec3 pos3(1.0, -1.0, 0.0);
glm::vec3 pos4(1.0, 1.0, 0.0);
// texture coordinates
glm::vec2 uv1(0.0, 1.0);
glm::vec2 uv2(0.0, 0.0);
glm::vec2 uv3(1.0, 0.0);
glm::vec2 uv4(1.0, 1.0);
// normal vector
glm::vec3 nm(0.0, 0.0, 1.0);
```

我们先计算第一个三角形的边和deltaUV坐标：

```
glm::vec3 edge1 = pos2 - pos1;
glm::vec3 edge2 = pos3 - pos1;
glm::vec2 deltaUV1 = uv2 - uv1;
glm::vec2 deltaUV2 = uv3 - uv1;
```

有了计算切线和副切线的必备数据，我们就可以开始写出来自于前面部分中的下列等式：

```
GLfloat f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);

tangent1.x = f * (deltaUV2.y * edge1.x - deltaUV1.y * edge2.x);
tangent1.y = f * (deltaUV2.y * edge1.y - deltaUV1.y * edge2.y);
tangent1.z = f * (deltaUV2.y * edge1.z - deltaUV1.y * edge2.z);
tangent1 = glm::normalize(tangent1);

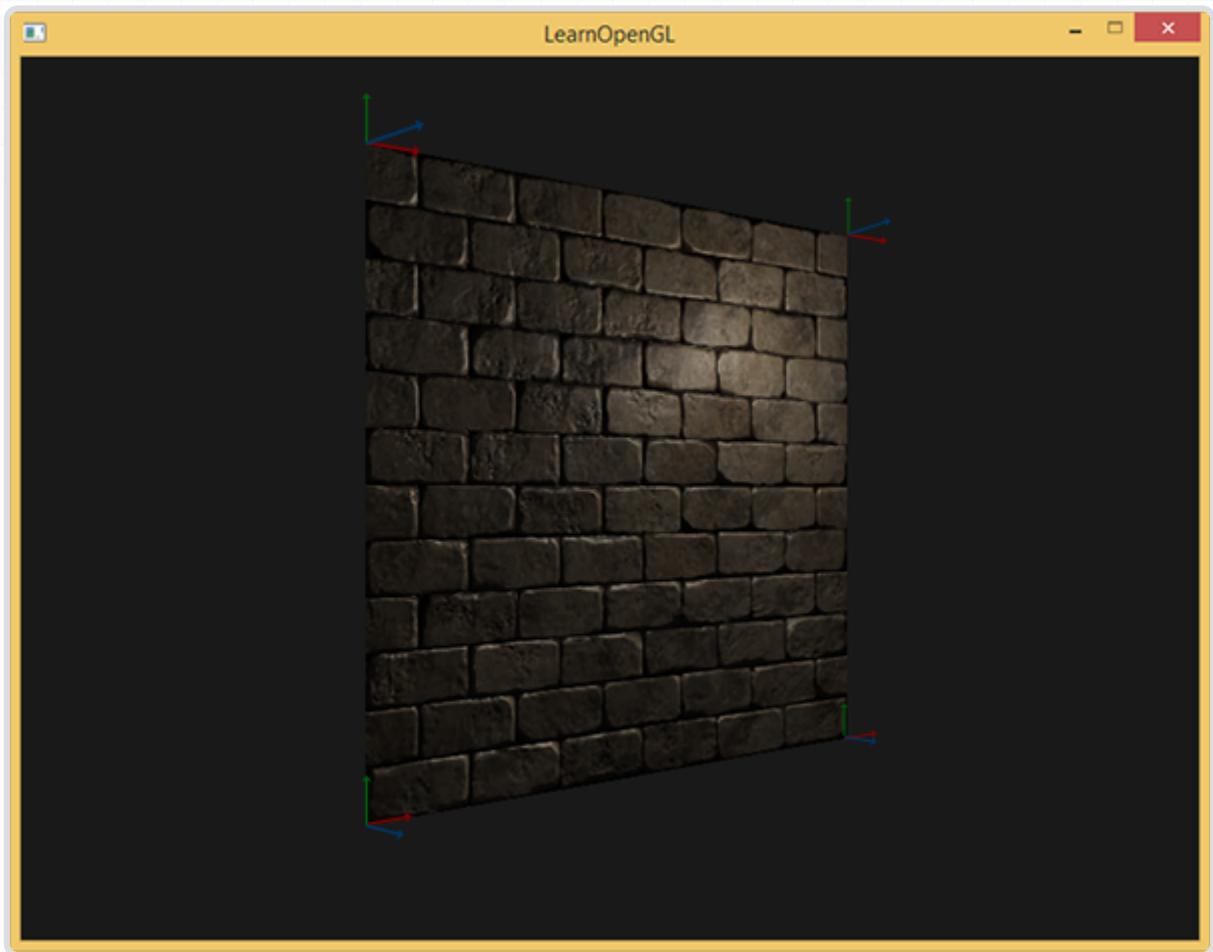
bitangent1.x = f * (-deltaUV2.x * edge1.x + deltaUV1.x * edge2.x);
bitangent1.y = f * (-deltaUV2.x * edge1.y + deltaUV1.x * edge2.y);
bitangent1.z = f * (-deltaUV2.x * edge1.z + deltaUV1.x * edge2.z);
bitangent1 = glm::normalize(bitangent1);

[...] // 对平面的第二个三角形采用类似步骤计算切线和副切线
```

我们预先计算出等式的分数部分 f ，然后把它和每个向量的元素进行相应矩阵乘法。如果你把代码和最终的等式对比你会发现，这就是直接套用。最后我们还要进行标准化，来确保切线/副切线向量最后是单位向量。

因为一个三角形永远是平坦的形状，我们只需为每个三角形计算一个切线/副切线，它们对于每个三角形上的顶点都是一样的。要注意的是大多数实现通常三角形和三角形之间都会共享顶点。这种情况下开发者通常将每个顶点的法线和切线/副切线等顶点属性平均化，以获得更加柔和的效果。我们的平面的三角形之间分享了一些顶点，但是因为两个三角形相互平行，因此并不需要将结果平均化，但无论何时只要你遇到这种情况记住它就是件好事。

最后的切线和副切线向量的值应该是 $(1, 0, 0)$ 和 $(0, 1, 0)$ ，它们和法线 $(0, 0, 1)$ 组成相互垂直的TBN矩阵。在平面上显示出来TBN应该是这样的：



每个顶点定义了切线和副切线向量，我们就可以开始实现正确的法线贴图了。

切线空间法线贴图

为让法线贴图工作，我们先得在着色器中创建一个TBN矩阵。我们先将前面计算出来的切线和副切线向量传给顶点着色器，作为它的属性：

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;
layout (location = 3) in vec3 tangent;
layout (location = 4) in vec3 bitangent;
```

在顶点着色器的main函数中我们创建TBN矩阵：

```
void main()
{
    [...]
    vec3 T = normalize(vec3(model * vec4(tangent, 0.0)));
    vec3 B = normalize(vec3(model * vec4(bitangent, 0.0)));
    vec3 N = normalize(vec3(model * vec4(normal, 0.0)));
    mat3 TBN = mat3(T, B, N)
}
```

我们先将所有TBN向量变换到我们所操作的坐标系中，现在是世界空间，我们可以乘以model矩阵。然后我们创建实际的TBN矩阵，直接把相应的向量应用到mat3构造器就行。注意，如果我们希望更精确的话就不要将TBN向量乘以model矩阵，而是使用法线矩阵，因为我们只关心向量的方向，不关心平移和缩放。

从技术上讲，顶点着色器中无需副切线。所有的这三个TBN向量都是相互垂直的所以我们可以在顶点着色器中用T和N向量的叉乘，自己计算出副切线： $\text{vec3 } B = \text{cross}(T, N)$ ；

现在我们有了TBN矩阵，如果来使用它呢？通常来说有两种方式使用它，我们会把这两种方式都说明一下：

1. 我们直接使用TBN矩阵，这个矩阵可以把切线坐标空间的向量转换到世界坐标空间。因此我们把它传给片段着色器中，把通过采样得到的法线坐标左乘上TBN矩阵，转换到世界坐标空间中，这样所有法线和其他光照变量就在同一个坐标系中了。
2. 我们也可以使用TBN矩阵的逆矩阵，这个矩阵可以把世界坐标空间的向量转换到切线坐标空间。因此我们使用这个矩阵左乘其他光照变量，把他们转换到切线空间，这样法线和其他光照变量再一次在一个坐标系中了。

我们来看看第一种情况。我们从法线贴图采样得来的法线向量，是在切线空间表示的，尽管其他光照向量都是在世界空间表示的。把TBN传给像素着色器，我们就能将采样得来的切线空间的法线乘以这个TBN矩阵，将法线向量变换到和其他光照向量一样的参考空间中。这种方式随后所有光照计算都可以简单的理解。

把TBN矩阵发给像素着色器很简单：

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    mat3 TBN;
} vs_out;

void main()
{
    [...]
    vs_out.TBN = mat3(T, B, N);
}
```

在像素着色器中我们用mat3作为输入变量：

```
in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    mat3 TBN;
} fs_in;
```

有了TBN矩阵我们现在就可以更新法线贴图代码，引入切线到世界空间变换：

```
normal = texture(normalMap, fs_in.TexCoords).rgb;
normal = normalize(normal * 2.0 - 1.0);
normal = normalize(fs_in.TBN * normal);
```

因为最后的normal现在在世界空间中了，就不用改变其他像素着色器的代码了，因为光照代码就是假设法线向量在世界空间中。

我们同样看看第二种情况。我们用TBN矩阵的逆矩阵将所有相关的世界空间向量转变到采样所得法线向量的空间：切线空间。TBN的建构还是一样，但我们在将其发送给像素着色器之前先要求逆矩阵：

```
vs_out.TBN = transpose(mat3(T, B, N));
```

注意，这里我们使用transpose函数，而不是inverse函数。正交矩阵（每个轴既是单位向量同时相互垂直）的一大属性是一个正交矩阵的置换矩阵与它的逆矩阵相等。这个属性很重要因为逆矩阵的求得比求置换开销大；结果却是一样的。

在像素着色器中我们不用对法线向量变换，但我们要把其他相关向量转换到切线空间，它们是lightDir和viewDir。这样每个向量还是在同一个空间（切线空间）中了。

```
void main()
{
    vec3 normal = texture(normalMap, fs_in.TexCoords).rgb;
    normal = normalize(normal * 2.0 - 1.0);

    vec3 lightDir = fs_in.TBN * normalize(lightPos - fs_in.FragPos);
    vec3 viewDir = fs_in.TBN * normalize(viewPos - fs_in.FragPos);
    [...]
}
```

第二种方法看似要做的更多，它还需要在像素着色器中进行更多的乘法操作，所以为何还用第二种方法呢？

将向量从世界空间转换到切线空间有个额外好处，我们可以把所有相关向量在顶点着色器中转换到切线空间，不用在像素着色器中做这件事。这是可行的，因为lightPos和viewPos不是每个fragment运行都要改变，对于fs_in.FragPos，我们也可以在顶点着色器计算它的切线空间位置。基本上，不需要把任何向量在像素着色器中进行变换，而第一种方法中就是必须的，因为采样出来的法线向量对于每个像素着色器都不一样。

所以现在不是把TBN矩阵的逆矩阵发送给像素着色器，而是将切线空间的光源位置，观察位置以及顶点位置发送给像素着色器。这样我们就不用在像素着色器里进行矩阵乘法了。这是一个极佳的优化，因为顶点着色器通常比像素着色器运行的少。这也是为什么这种方法是一种更好的实现方式的原因。

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} vs_out;

uniform vec3 lightPos;
uniform vec3 viewPos;

[...]

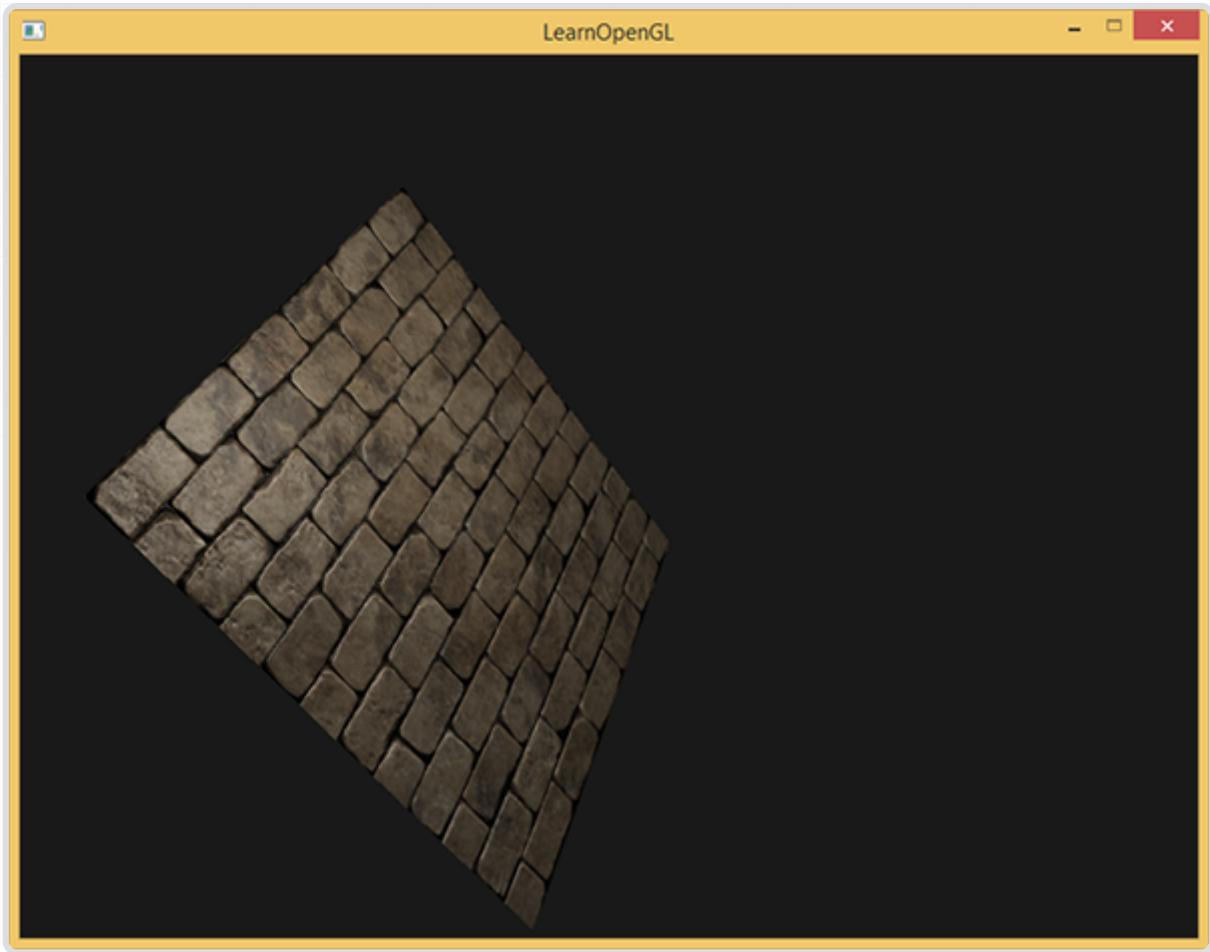
void main()
{
    [...]
    mat3 TBN = transpose(mat3(T, B, N));
    vs_out.TangentLightPos = TBN * lightPos;
    vs_out.TangentViewPos = TBN * viewPos;
    vs_out.TangentFragPos = TBN * vec3(model * vec4(position, 0.0));
}
```

在像素着色器中我们使用这些新的输入变量来计算切线空间的光照。因为法线向量已经在切线空间中了，光照就有意义了。

将法线贴图应用到切线空间上，我们会得到混合教程一开始那个例子相似的结果，但这次我们可以将平面朝向各个方向，光照一直都会是正确的：

```
glm::mat4 model;
model = glm::rotate(model, (GLfloat)glfwGetTime() * -10, glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
RenderQuad();
```

看起来是正确的法线贴图：



你可以在这里找到[源代码](#)、[顶点](#)和[像素](#)着色器。

复杂物体

我们已经说明了如何通过手工计算切线和副切线向量，来使用切线空间和法线贴图。幸运的是，计算这些切线和副切线向量对于你来说不是经常能遇到的事；大多数时候，在模型加载器中实现了一次就行了，我们是在使用了Assimp的那个加载器中实现的。

Assimp有个很有用的配置，在我们加载模型的时候调用aiProcess_CalcTangentSpace。当aiProcess_CalcTangentSpace应用到Assimp的ReadFile函数时，Assimp会为每个加载的顶点计算出柔和的切线和副切线向量，它所使用的方法和我们本教程使用的类似。

```
const aiScene* scene = importer.ReadFile(
    path, aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_CalcTangentSpace
);
```

我们可以通过下面的代码用Assimp获取计算出来的切线空间：

```
vector.x = mesh->mTangents[i].x;
vector.y = mesh->mTangents[i].y;
vector.z = mesh->mTangents[i].z;
vertex.Tangent = vector;
```

然后，你还必须更新模型加载器，用以从带纹理模型中加载法线贴图。wavefront的模型格式（.obj）导出的法线贴图有点不一样，Assimp的aiTextureType_NORMAL并不会加载它的法线贴图，而aiTextureType_HEIGHT却能，所以我们经常这样加载它们：

```
vector<Texture> specularMaps = this->loadMaterialTextures(
    material, aiTextureType_HEIGHT, "texture_normal"
);
```

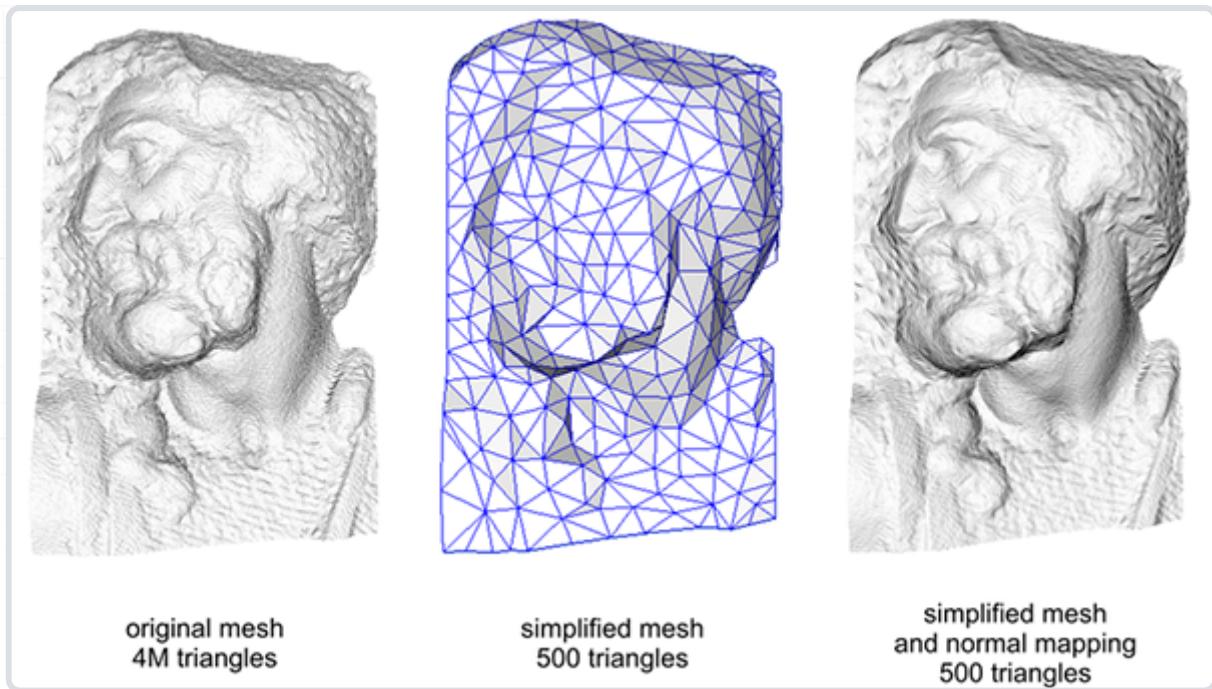
当然，对于每个模型的类型和文件格式来说都是不同的。同样了解aiProcess_CalcTangentSpace并不能总是很好的工作也很重要。计算切线是需要根据纹理坐标的，有些模型制作者使用一些纹理小技巧比如镜像一个模型上的纹理表面时也镜像了另一半的纹理坐标；这样当不考虑这个镜像的特别操作的时候（Assimp就不考虑）结果就不对了。

运行程序，用新的模型加载器，加载一个有specular和法线贴图的模型，看起来会像这样：



你可以看到在没有太多点的额外开销的情况下法线贴图难以置信地提升了物体的细节。

使用法线贴图也是一种提升你的场景的表现的重要方式。在使用法线贴图之前你不得不使用相当多的顶点才能表现出一个更精细的网格，但使用了法线贴图我们可以使用更少的顶点表现出同样丰富的细节。下图来自Paolo Cignoni，图中对比了两种方式：



高精度网格和使用法线贴图的低精度网格几乎区分不出来。所以法线贴图不仅看起来漂亮，它也是一个将高精度多边形转换为低精度多边形而不失细节的重要工具。

最后一件事

关于法线贴图还有最后一个技巧要讨论，它可以在不必花费太多性能开销的情况下稍稍提升画质表现。

当在更大的网格上计算切线向量的时候，它们往往有很大量数的共享顶点，当法向贴图应用到这些表面时将切线向量平均化通常能获得更好更平滑的结果。这样做有个问题，就是TBN向量可能会不能互相垂直，这意味着TBN矩阵不再是正交矩阵了。法线贴图可能会稍稍偏移，但这仍然可以改进。

使用叫做格拉姆-施密特正交化过程（Gram-Schmidt process）的数学技巧，我们可以对TBN向量进行重正交化，这样每个向量就又会重新垂直了。在顶点着色器中我们这样做：

```
vec3 T = normalize(vec3(model * vec4(tangent, 0.0)));
vec3 N = normalize(vec3(model * vec4(normal, 0.0)));
// re-orthogonalize T with respect to N
T = normalize(T - dot(T, N) * N);
// then retrieve perpendicular vector B with the cross product of T and N
vec3 B = cross(T, N);

mat3 TBN = mat3(T, B, N)
```

这样稍微花费一些性能开销就能对法线贴图进行一点提升。看看最后的那个附加资源：Normal Mapping Mathematics视频，里面有对这个过程的解释。

附加资源

- [Tutorial 26: Normal Mapping](#) : ogldev的法线贴图教程。
- [How Normal Mapping Works](#) : TheBennyBox的讲述法线贴图如何工作的视频。
- [Normal Mapping Mathematics](#) : TheBennyBox关于法线贴图的数学原理的教程。

- [Tutorial 13: Normal Mapping](#) : opengl-tutorial.org提供的法线贴图教程。

2.6.5 视差贴图

原文 [Parallax Mapping](#)

作者 JoeyDeVries

翻译 [Django](#)

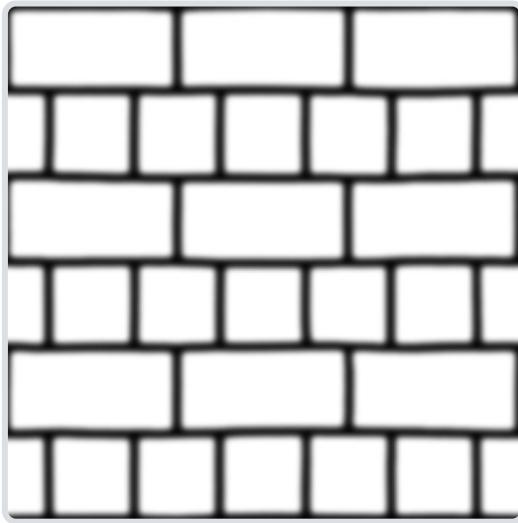
校对 暂无

Note

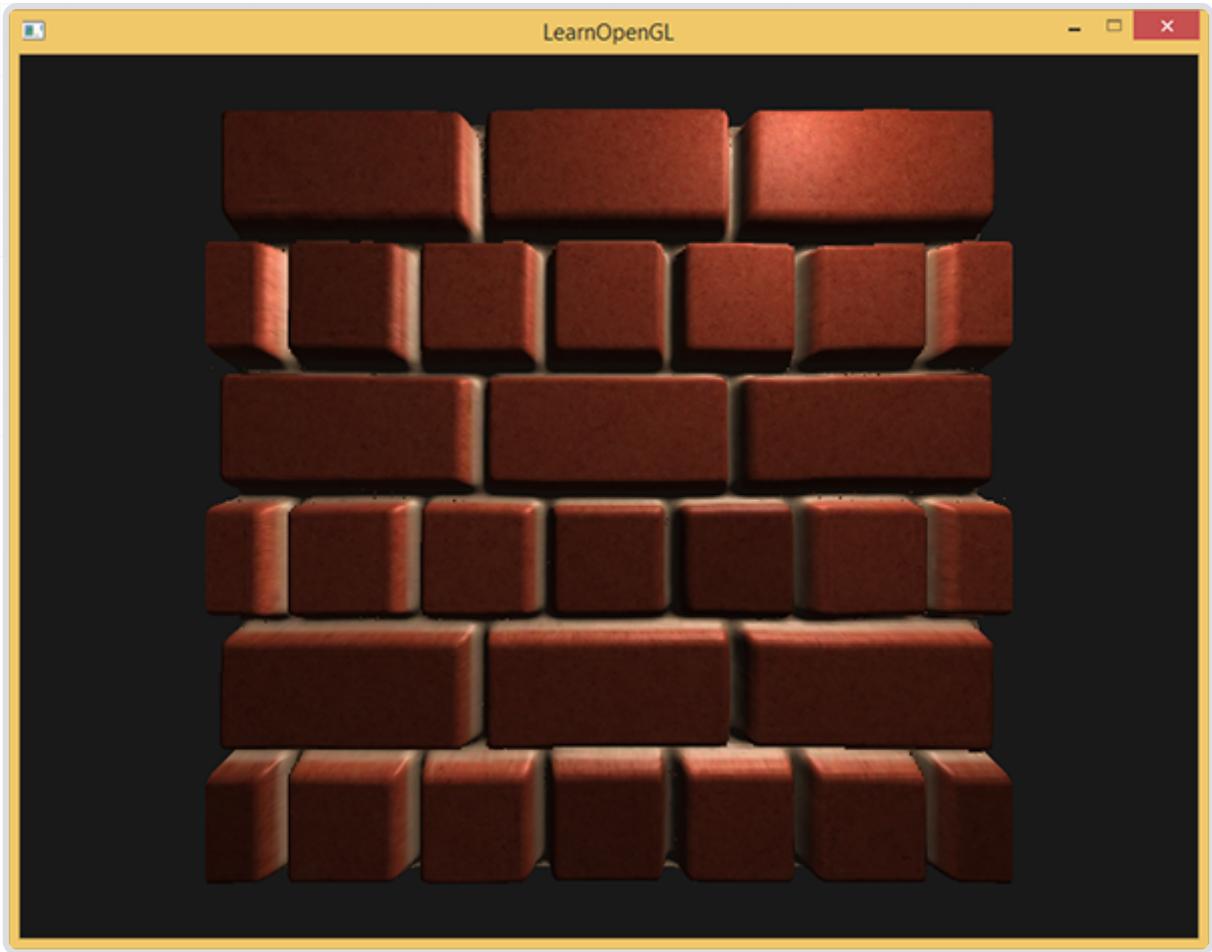
本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

视差贴图(Parallax Mapping)技术和法线贴图差不多，但它有着不同的原则。和法线贴图一样视差贴图能够极大提升表面细节，使之具有深度感。它也是利用了视错觉，然而对深度有着更好的表达，与法线贴图一起用能够产生难以置信的效果。视差贴图和光照无关，我在这里是作为法线贴图的技术延续来讨论它的。需要注意的是在开始学习视差贴图之前强烈建议先对法线贴图，特别是切线空间有较好的理解。

视差贴图属于位移贴图(Displacement Mapping)技术的一种，它对根据储存在纹理中的几何信息对顶点进行位移或偏移。一种实现的方式是比如有1000个顶点，根据纹理中的数据对平面特定区域的顶点的高度进行位移。这样的每个纹理像素包含了高度值纹理叫做高度贴图。一张简单的砖块表面的高度贴图如下所示：



整个平面上的每个顶点都根据从高度贴图采样出来的高度值进行位移，根据材质的几何属性平坦的平面变换为凹凸不平的表面。例如一个平坦的平面利用上面的高度贴图进行置换能得到以下结果：

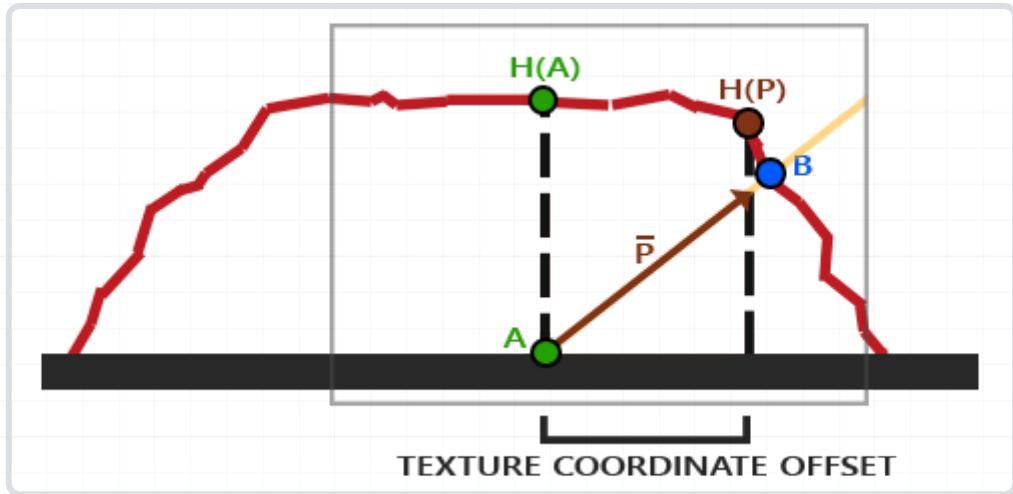


置换顶点有一个问题就是平面必须由很多顶点组成才能获得具有真实感的效果，否则看起来效果并不会很好。一个平坦的表面上有1000个顶点计算量太大了。我们能否不用这么多的顶点就能取得相似的效果呢？事实上，上面的表面就是用6个顶点渲染出来的（两个三角形）。上面的那个表面使用视差贴图技术渲染，位移贴图技术不需要额外的顶点数据来表达深度，它像法线贴图一样采用一种聪明的手段欺骗用户的眼睛。

视差贴图背后的思想是修改纹理坐标使一个fragment的表面看起来比实际的更高或者更低，所有这些都根据观察方向和高度贴图。为了理解它如何工作，看看下面砖块表面的图片：

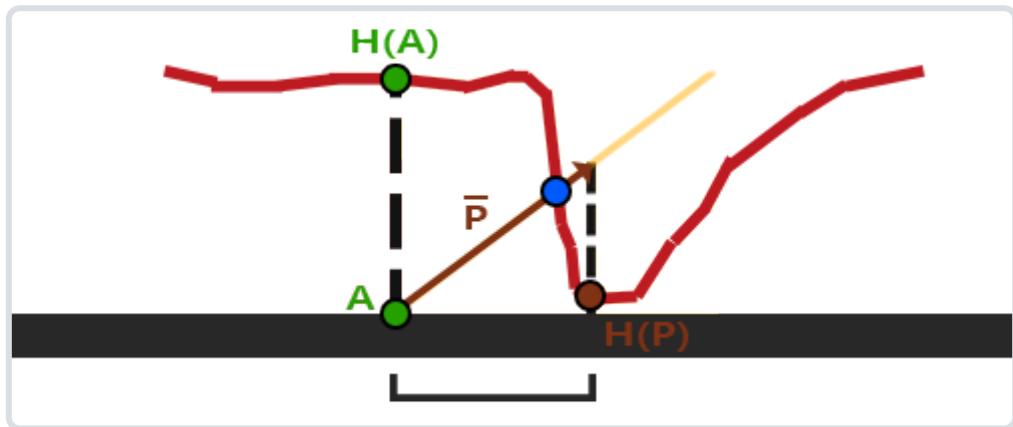
这里粗糙的红线代表高度贴图中的数值的立体表达，向量代表观察方向。如果平面进行实际位移，观察者会在点看到表面。然而我们的平面没有实际上进行位移，观察方向将在点与平面接触。视差贴图的目的是，在位置上的fragment不再使用点的纹理坐标而是使用点的。随后我们用点的纹理坐标采样，观察者就像看到了点一样。

这个技巧就是描述如何从点得到点的纹理坐标。视差贴图尝试通过对从fragment到观察者的方向向量进行缩放的方式解决这个问题，缩放的大小是处fragment的高度。所以我们将的长度缩放为高度贴图在点处采样得来的值。下图展示了经缩放得到的向量：



我们随后选出以及这个向量与平面对齐的坐标作为纹理坐标的偏移量。这能工作是因为向量是使用从高度贴图得到的高度值计算出来的，所以一个fragment的高度越高位移的量越大。

这个技巧在大多数时候都没问题，但点是粗略估算得到的。当表面的高度变化很快的时候，看起来就不会真实，因为向量最终不会和接近，就像下图这样：



视差贴图的另一个问题是，当表面被任意旋转以后很难指出从获取哪一个坐标。我们在视差贴图中使用了另一个坐标空间，这个空间向量的x和y元素总是与纹理表面对齐。如果你看了法线贴图教程，你也许猜到了，我们实现它的方法，是的，我们还是在切线空间中实现视差贴图。

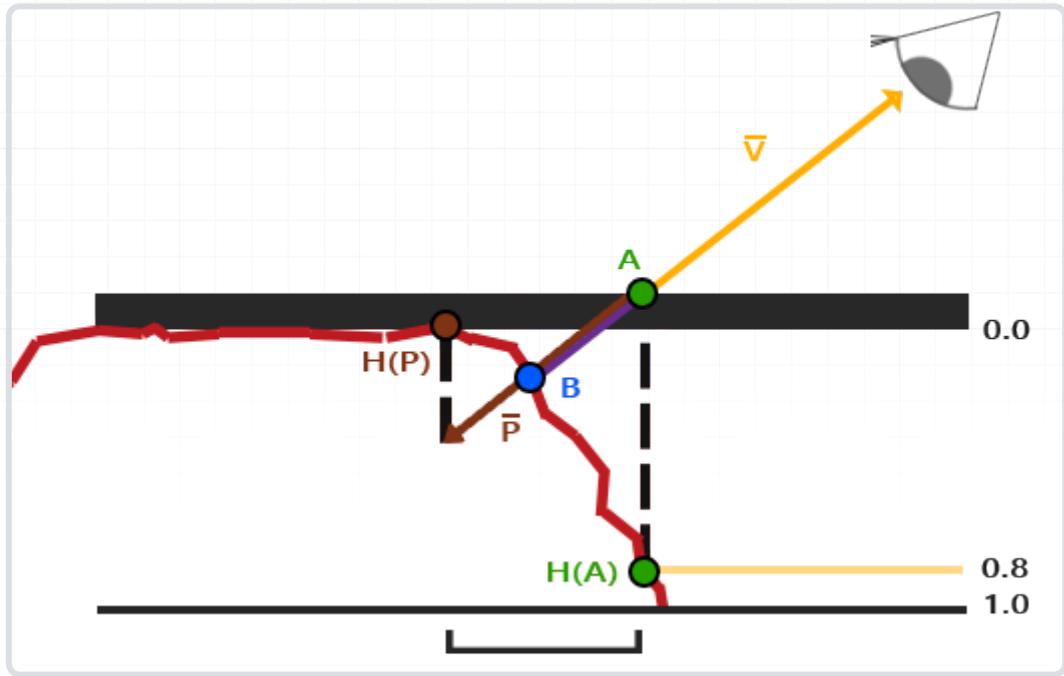
将fragment到观察者的向量转换到切线空间中，经变换的向量的x和y元素将于表面的切线和副切线向量对齐。由于切线和副切线向量与表面纹理坐标的方向相同，我们可以用的x和y元素作为纹理坐标的偏移量，这样就不用考虑表面的方向了。

理论都有了，下面我们来动手实现视差贴图。

视差贴图

我们将使用一个简单的2D平面，在把它发送给GPU之前我们先计算它的切线和副切线向量；和法线贴图教程做的差不多。我们将向平面贴diffuse纹理、法线贴图以及一个位移贴图，你可以点击链接下载。这个例子中我们将视差贴图和法线贴图连用。因为视差贴图生成表面位移了的幻觉，当光照不匹配时这种幻觉就被破坏了。法线贴图通常根据高度贴图生成，法线贴图和高度贴图一起用能保证光照能和位移想匹配。

你可能已经注意到，上面链接上的那个位移贴图和教程一开始的那个高度贴图相比是颜色是相反的。这是因为使用反色高度贴图（也叫深度贴图）去模拟深度比模拟高度更容易。下图反映了这个轻微的改变：



我们再次获得和，但是这次我们用向量减去点的纹理坐标得到。我们通过在着色器中用1.0减去采样得到的高度贴图中的值来取得深度值，而不再是高度值，或者简单地在图片编辑软件中把这个纹理进行反色操作，就像我们对连接中的那个深度贴图所做的一样。

位移贴图是在像素着色器中实现的，因为三角形表面的所有位移效果都不同。在像素着色器中我们将需要计算fragment到观察者到方向向量所以我们需要观察者位置和在切线空间中的fragment位置。法线贴图教程中我们已经有了一个顶点着色器，它把这些向量发送到切线空间，所以我们可以复制那个顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;
layout (location = 3) in vec3 tangent;
layout (location = 4) in vec3 bitangent;

out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

uniform vec3 lightPos;
uniform vec3 viewPos;

void main()
{
    gl_Position      = projection * view * model * vec4(position, 1.0f);
    vs_out.FragPos   = vec3(model * vec4(position, 1.0));
    vs_out.TexCoords = texCoords;

    vec3 T   = normalize(mat3(model) * tangent);
    vec3 B   = normalize(mat3(model) * bitangent);
    vec3 N   = normalize(mat3(model) * normal);
    mat3 TBN = transpose(mat3(T, B, N));

    vs_out.TangentLightPos = TBN * lightPos;
    vs_out.TangentViewPos = TBN * viewPos;
    vs_out.TangentFragPos = TBN * vs_out.FragPos;
}
```

在这里有件事很重要，我们需要把position和在切线空间中的观察者的位置viewPos发送给像素着色器。

在像素着色器中，我们实现视差贴图的逻辑。像素着色器看起来会是这样的：

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} fs_in;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D depthMap;

uniform float height_scale;

vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir);

void main()
{
    // Offset texture coordinates with Parallax Mapping
    vec3 viewDir = normalize(fs_in.TangentViewPos - fs_in.TangentFragPos);
    vec2 texCoords = ParallaxMapping(fs_in.TexCoords, viewDir);

    // then sample textures with new texture coords
    vec3 diffuse = texture(diffuseMap, texCoords);
    vec3 normal = texture(normalMap, texCoords);
    normal = normalize(normal * 2.0 - 1.0);
    // proceed with lighting code
    [...]
}
```

我们定义了一个叫做ParallaxMapping的函数，它把fragment的纹理坐标和切线空间中的fragment到观察者的方向向量为输入。这个函数返回经位移的纹理坐标。然后我们使用这些经位移的纹理坐标进行diffuse和法线贴图的采样。最后fragment的diffuse颜色和法线向量就正确的对应于表面的经位移的位置上了。

我们来看看ParallaxMapping函数的内部：

```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    float height = texture(depthMap, texCoords).r;
    vec2 p = viewDir.xy / viewDir.z * (height * height_scale);
    return texCoords - p;
}
```

这个相对简单的函数是我们所讨论过的内容的直接表述。我们用本来的纹理坐标texCoords从高度贴图中来采样，得到当前fragment的高度。然后计算出，x和y元素在切线空间中，viewDir向量除以它的z元素，用fragment的高度对它进行缩放。我们同时引入一个height_scale的uniform，来进行一些额外的控制，因为视差效果如果没有一个缩放参数通常会过于强烈。然后我们用减去纹理坐标来获得最终的经过位移纹理坐标。

有一个地方需要注意，就是viewDir.xy除以viewDir.z那里。因为viewDir向量是经过了标准化的，viewDir.z会在0.0到1.0之间的某处。当viewDir大致平行于表面时，它的z元素接近于0.0，除法会返回比viewDir垂直于表面的时候更大的向量。所以，从本质上，相比正朝向表面，当带有角度地看向平面时，我们会更大程度地缩放的大小，从而增加纹理坐标的偏移；这样做在视角上会获得更大的真实度。

有些人更喜欢不在等式中使用`viewDir.z`，因为普通的视差贴图会在角度上产生不尽如人意的结果；这个技术叫做有偏移量限制的视差贴图（Parallax Mapping with Offset Limiting）。选择哪一个技术是个人偏好问题，但我倾向于普通的视差贴图。

最后的纹理坐标随后被用来进行采样（diffuse和法线）贴图，下图所展示的位移效果中`height_scale`等于1：

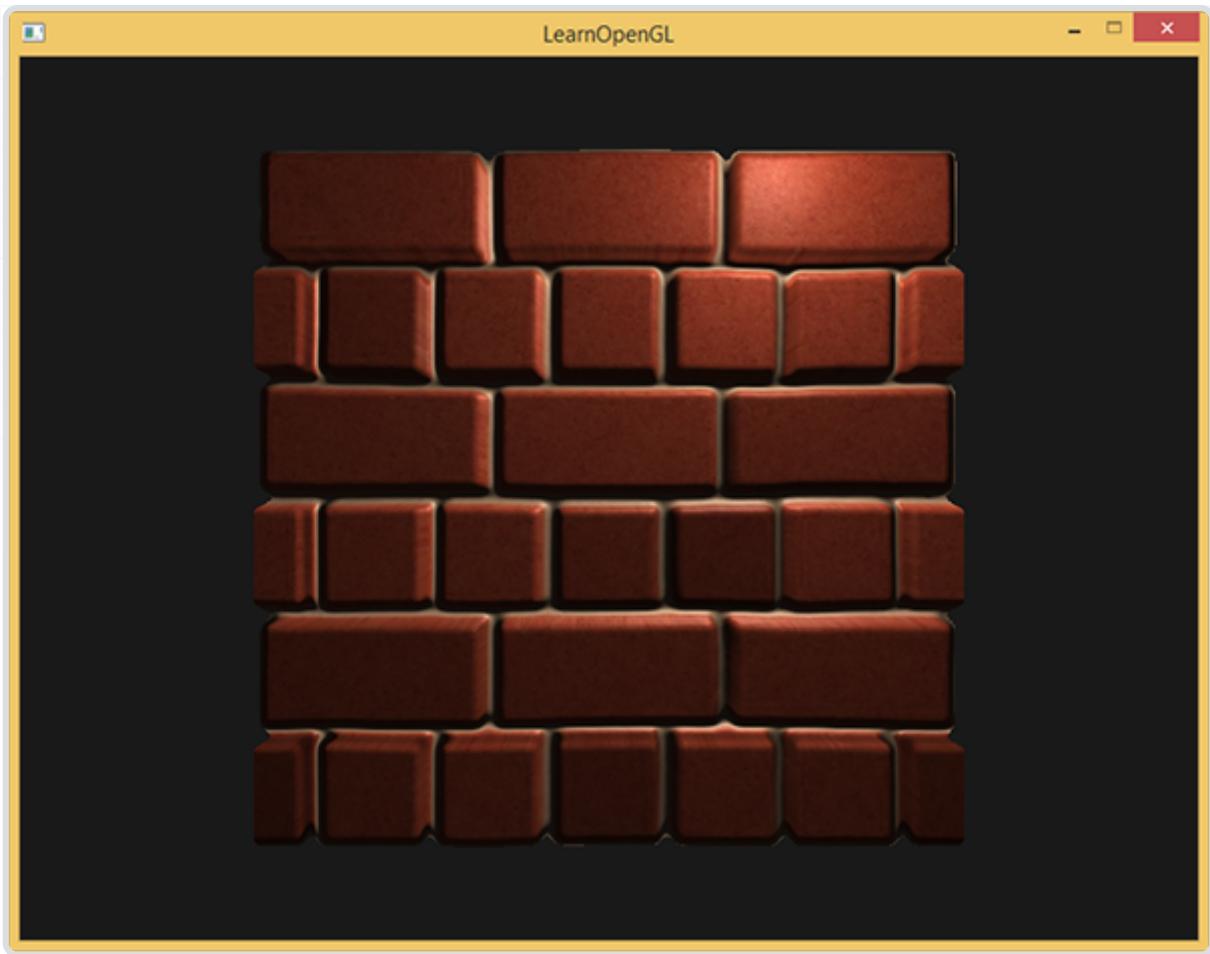


这里你会看到只用法线贴图和与视差贴图相结合的法线贴图的不同之处。因为视差贴图尝试模拟深度，它实际上能够根据你观察它们的方向使砖块叠加到其他砖块上。

在视差贴图的那个平面里你仍然能看到在边上有古怪的失真。原因是在平面的边缘上，纹理坐标超出了0到1的范围进行采样，根据纹理的环绕方式导致了不真实的结果。解决的方法是当它超出默认纹理坐标范围进行采样的时候就丢弃这个fragment：

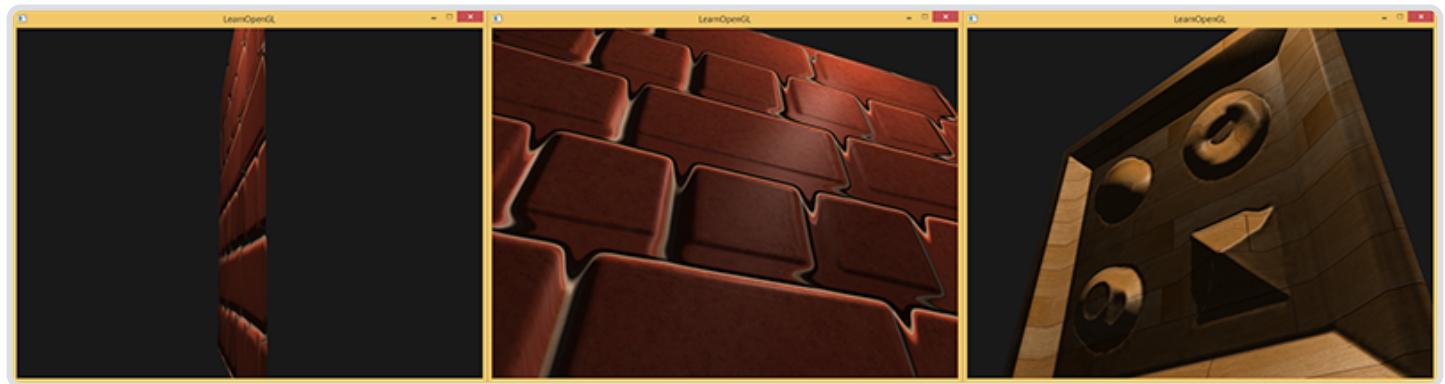
```
texCoords = ParallaxMapping(fs_in.TexCoords, viewDir);
if(texCoords.x > 1.0 || texCoords.y > 1.0 || texCoords.x < 0.0 || texCoords.y < 0.0)
    discard;
```

丢弃了超出默认范围的纹理坐标的所有fragment，视差贴图的表面边缘给出了正确的结果。注意，这个技巧不能在所有类型的表面上都能工作，但是应用于平面上它还是能够使平面看起来真的进行位移了：



你可以在这里找到[源代码](#), 以及[顶点](#)和[像素着色器](#)。

看起来不错, 运行起来也很快, 因为我们只要给视差贴图提供一个额外的纹理样本就能工作。当从一个角度看过去的时候, 会有一些问题产生 (和法线贴图相似), 陡峭的地方会产生不正确的结果, 从下图你可以看到:

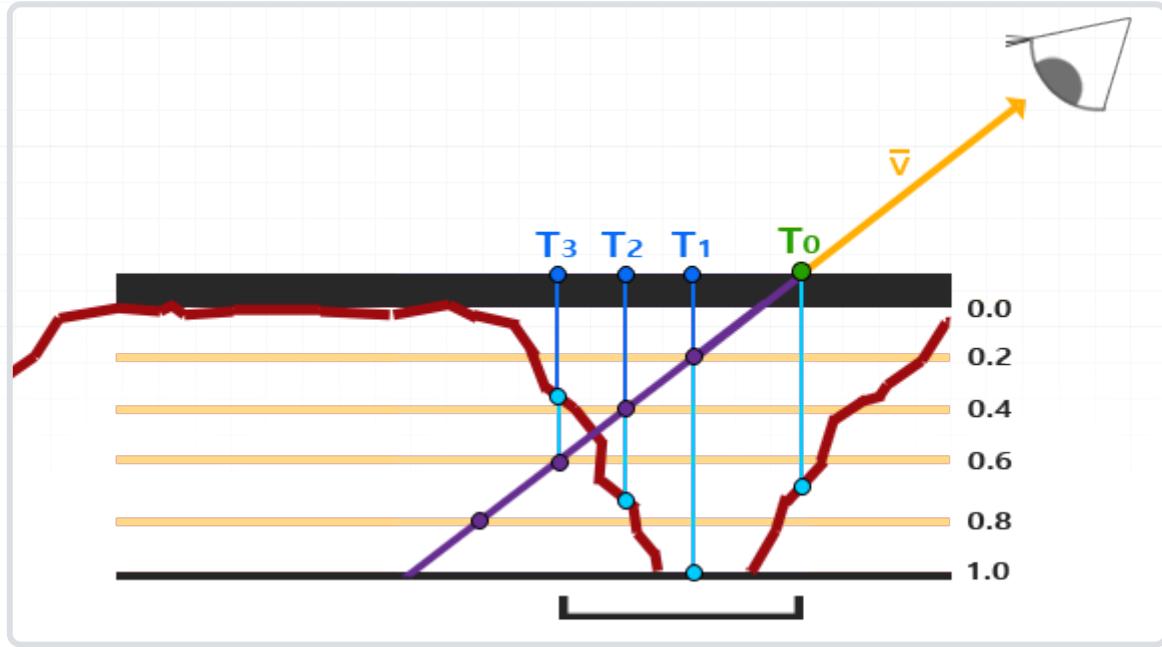


问题的原因是这只是一个大致近似的视差映射。还有一些技巧让我们在陡峭的高度上能够获得几乎完美的结果, 即使当以一定角度观看的时候。例如, 我们不再使用单一样本, 取而代之使用多样本来找到最近点会得到怎样的结果?

陡峭视差映射

陡峭视差映射(Steep Parallax Mapping)是视差映射的扩展, 原则是一样的, 但不是使用一个样本而是多个样本来确定向量到。即使在陡峭的高度变化的情况下, 它也能得到更好的结果, 原因在于该技术通过增加采样的数量提高了精确性。

陡峭视差映射的基本思想是将总深度范围划分为同一个深度/高度的多个层。从每个层中我们沿着方向移动采样纹理坐标，直到我们找到一个采样低于当前层的深度值。看看下面的图片：



我们从上到下遍历深度层，我们把每个深度层和储存在深度贴图中的它的深度值进行对比。如果这个层的深度值小于深度贴图的值，就意味着这一层的向量部分在表面之下。我们继续这个处理过程直到有一层的深度大于储存在深度贴图中的值：这个点就在（经过位移的）表面下方。

这个例子中我们可以看到第二层($D(2) = 0.73$)的深度贴图的值仍低于第二层的深度值0.4，所以我们继续。下一次迭代，这一层的深度值0.6大于深度贴图中采样的深度值($D(3) = 0.37$)。我们便可以假设第三层向量是可用的位移几何位置。我们可以用从向量的纹理坐标偏移来对fragment的纹理坐标进行位移。你可以看到随着深度层的增加精确度也在提高。

为实现这个技术，我们只需要改变ParallaxMapping函数，因为所有需要的变量都有了：

```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    // number of depth layers
    const float numLayers = 10;
    // calculate the size of each layer
    float layerDepth = 1.0 / numLayers;
    // depth of current layer
    float currentLayerDepth = 0.0;
    // the amount to shift the texture coordinates per layer (from vector P)
    vec2 P = viewDir.xy * height_scale;
    float deltaTexCoords = P / numLayers;

    [...]
}
```

我们先定义层的数量，计算每一层的深度，最后计算纹理坐标偏移，每一层我们必须沿着的方向进行移动。

然后我们遍历所有层，从上开始，知道找到小于这一层的深度值的深度贴图值：

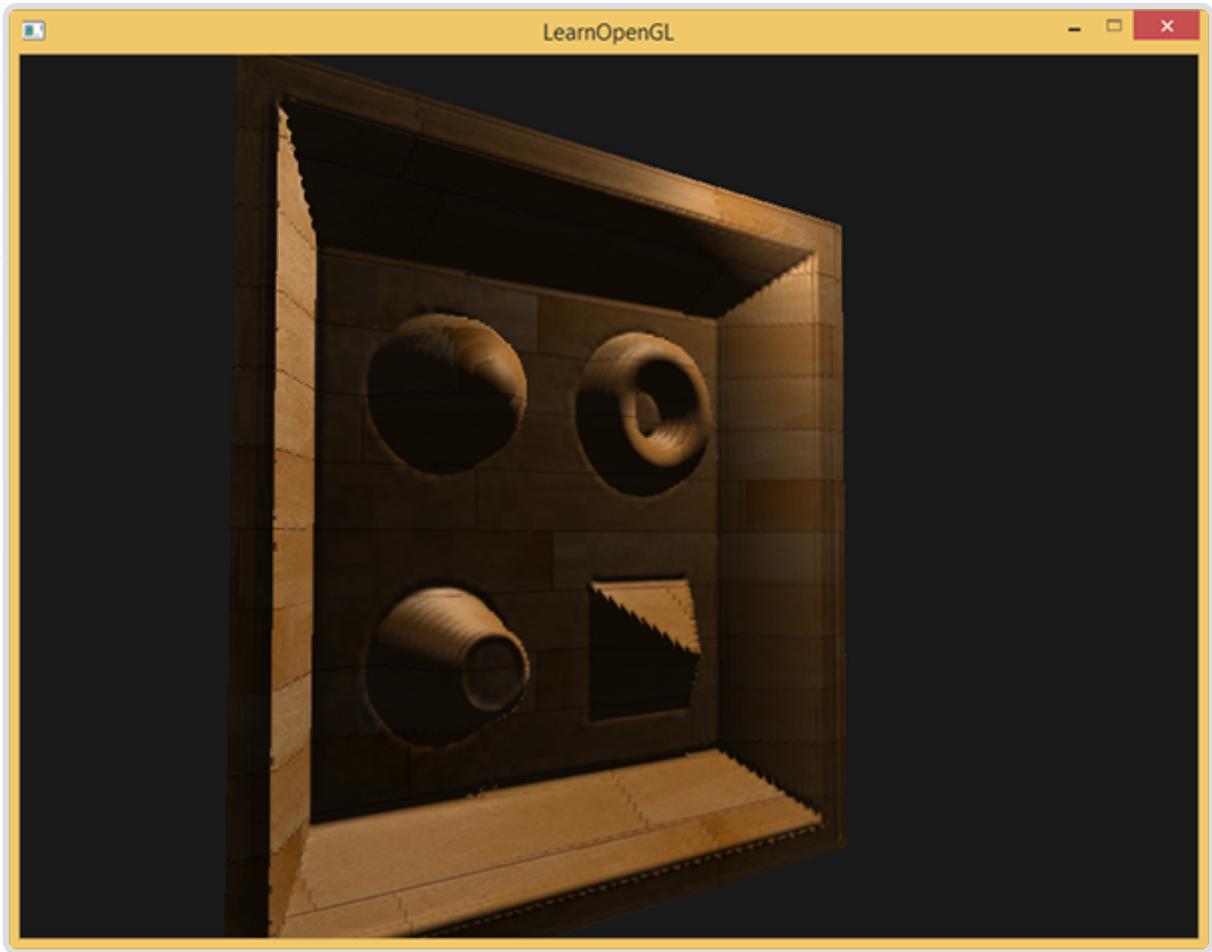
```
// get initial values
vec2 currentTexCoords = texCoords;
float currentDepthMapValue = texture(depthMap, currentTexCoords).r;

while(currentLayerDepth < currentDepthMapValue)
{
    // shift texture coordinates along direction of P
    currentTexCoords -= deltaTexCoords;
    // get depthmap value at current texture coordinates
    currentDepthMapValue = texture(depthMap, currentTexCoords).r;
    // get depth of next layer
    currentLayerDepth += layerDepth;
}

return texCoords - currentTexCoords;
```

这里我们循环每一层深度，直到沿着向量找到第一个返回低于（位移）表面的深度的纹理坐标偏移量。从fragment的纹理坐标减去最后的偏移量，来得到最终的经过位移的纹理坐标向量，这次就比传统的视差映射更精确了。

有10个样本砖墙从一个角度看上去就已经很好了，但是当有一个强前面展示的木制表面一样陡峭的表面时，陡峭的视差映射的威力就显示出来了：



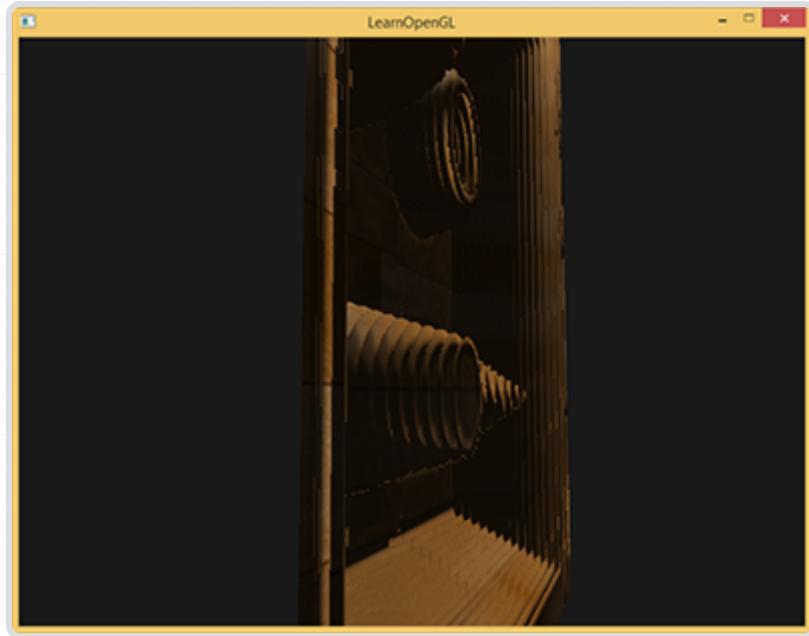
我们可以通过对视差贴图的一个属性的利用，对算法进行一点提升。当垂直看一个表面的时候纹理时位移比以一定角度看时的小。我们可以在垂直看时使用更少的样本，以一定角度看时增加样本数量：

```
const float minLayers = 8;
const float maxLayers = 32;
float numLayers = mix(maxLayers, minLayers, abs(dot(vec3(0.0, 0.0, 1.0), viewDir)));
```

这里我们得到viewDir和正z方向的点乘，使用它的结果根据我们看向表面的角度调整样本数量（注意正z方向等于切线空间中的表面的法线）。如果我们所看的方向平行于表面，我们就是用32层。

你可以在这里找到最新的像素着色器代码。这里也提供木制玩具箱的表面贴图：diffuse、法线、深度。

陡峭视差贴图同样有自己的问题。因为这个技术是基于有限的样本数量的，我们会遇到锯齿效果以及图层之间有明显的断层：



我们可以通过增加样本的方式减少这个问题，但是很快就会花费很多性能。有些旨在修复这个问题的方法：不适用低于表面的第一个位置，而是在两个接近的深度层进行插值找出更匹配的。

两种最流行的解决方法叫做Relief Parallax Mapping和Parallax Occlusion Mapping，Relief Parallax Mapping更精确一些，但是比Parallax Occlusion Mapping性能开销更多。因为Parallax Occlusion Mapping的效果和前者差不多但是效率更高，因此这种方式更经常使用，所以我们将在下面讨论一下。

视差遮蔽映射

视差遮蔽映射(Parallax Occlusion Mapping)和陡峭视差映射的原则相同，但不是用触碰的第一个深度层的纹理坐标，而是在触碰之前和之后，在深度层之间进行线性插值。我们根据表面的高度距离哪个深度层的深度层值的距离来确定线性插值的大小。看看下面的图片就能了解它是如何工作的：

你可以看到大部分和陡峭视差映射一样，不一样的地方是有个额外的步骤，两个深度层的纹理坐标围绕着交叉点的线性插值。这也是近似的，但是比陡峭视差映射更精确。

视差遮蔽映射的代码基于陡峭视差映射，所以并不难：

```
[...] // steep parallax mapping code here

// get texture coordinates before collision (reverse operations)
vec2 prevTexCoords = currentTexCoords + deltaTexCoords;

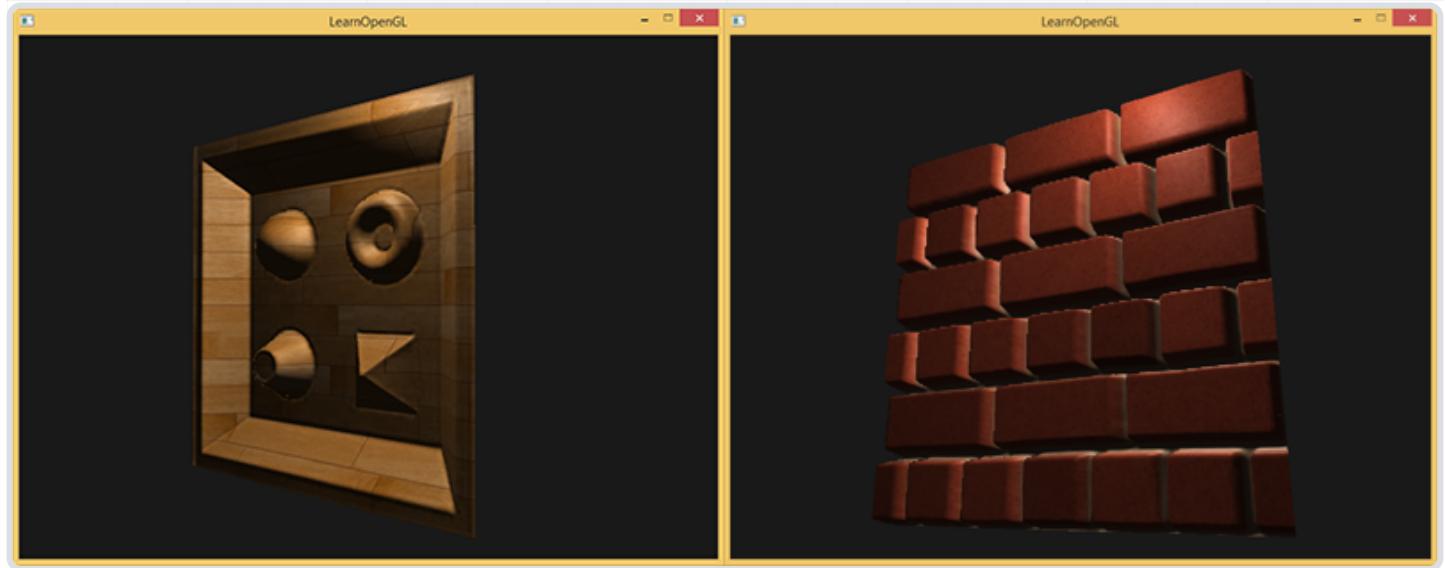
// get depth after and before collision for linear interpolation
float afterDepth = currentDepthMapValue - currentLayerDepth;
float beforeDepth = texture(depthMap, prevTexCoords).r - currentLayerDepth + layerDepth;

// interpolation of texture coordinates
float weight = afterDepth / (afterDepth - beforeDepth);
vec2 finalTexCoords = prevTexCoords * weight + currentTexCoords * (1.0 - weight);

return finalTexCoords;
```

在对（位移的）表面几何进行交叉，找到深度层之后，我们获取交叉前的纹理坐标。然后我们计算来自相应深度层的几何之间的深度之间的距离，并在两个值之间进行插值。线性插值的方式是在两个层的纹理坐标之间进行的基础插值。函数最后返回最终的经过插值的纹理坐标。

视差遮蔽映射的效果非常好，尽管有一些可以看到的轻微的不真实和锯齿的问题，这仍是一个好交易，因为除非是放得非常大或者观察角度特别陡，否则也看不到。



你可以在这里找到[源代码](#)，及其[顶点](#)和[像素着色器](#)。

视差贴图是提升场景细节非常好的技术，但是使用的时候还是要考虑到它会带来一点不自然。大多数时候视差贴图用在地面和墙壁表面，这种情况下查明表面的轮廓并不容易，同时观察角度往往趋于垂直于表面。这样视差贴图的不自然也就很难被注意到了，对于提升物体的细节可以祈祷难以置信的效果。

附加资源

- [Parallax Occlusion Mapping in GLSL](#) : sunandblackcat.com上的视差贴图教程。
- [How Parallax Displacement Mapping Works](#) : TheBennyBox的关于视差贴图原理的视频教程。

2.6.6 HDR

原文 [HDR](#)

作者 JoeyDeVries

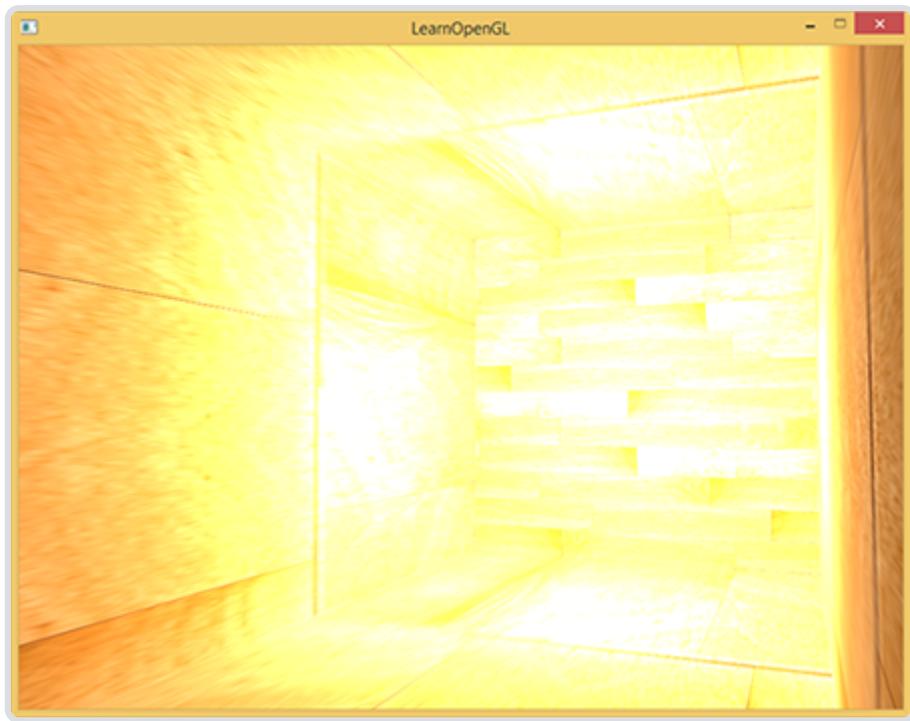
翻译 Krasjet

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

一般来说，当存储在帧缓冲(Framebuffer)中时，亮度和颜色的值是默认被限制在0.0到1.0之间的。这个看起来无辜的语句使我们一直将亮度与颜色的值设置在这个范围内，尝试着与场景契合。这样是能够运行的，也能给出还不错的效果。但是如果我们遇上了一个特定的区域，其中有多个亮光源使这些数值总和超过了1.0，又会发生什么呢？答案是这些片段中超过1.0的亮度或者颜色值会被约束在1.0，从而导致场景混成一片，难以分辨：



这是由于大量片段的颜色值都非常接近1.0，在很大一个区域内每一个亮的片段都有相同的白色。这损失了很多的细节，使场景看起来非常假。

解决这个问题的一个方案是减小光源的强度从而保证场景内没有一个片段亮于1.0。然而这并不是一个好的方案，因为你需要使用不切实际的光照参数。一个更好的方案是让颜色暂时超过1.0，然后将其转换至0.0到1.0的区间内，从而防止损失细节。

显示器被限制为只能显示值为0.0到1.0间的颜色，但是在光照方程中却没有这个限制。通过使片段的颜色超过1.0，我们有了一个更大的颜色范围，这也被称作**HDR(High Dynamic Range, 高动态范围)**。有了HDR，亮的东西可以变得非常亮，暗的东西可以变得非常暗，而且充满细节。

HDR原本只是被运用在摄影上，摄影师对同一个场景采取不同曝光拍多张照片，捕捉大范围的色彩值。这些图片被合成为HDR图片，从而综合不同的曝光等级使得大范围的细节可见。看下面这个例子，左边这张图片在被照亮的区域充满细节，但是在黑暗的区域就什么都看不到了；但是右边这张图的高曝光却可以让之前看不出来的黑暗区域显现出来。



这与我们眼睛工作的原理非常相似，也是HDR渲染的基础。当光线很弱的时候，人眼会自动调整从而使过暗和过亮的部分变得更清晰，就像人眼有一个能自动根据场景亮度调整的自动曝光滑块。

HDR渲染和其很相似，我们允许用更大范围的颜色值渲染从而获取大范围的黑暗与明亮的场景细节，最后将所有HDR值转换成[0.0, 1.0]范围的LDR(Low Dynamic Range,低动态范围)。转换HDR值到LDR值得过程叫做色调映射(Tone Mapping)，现在现存有很多的色调映射算法，这些算法致力于在转换过程中保留尽可能多的HDR细节。这些色调映射算法经常会包含一个选择性倾向黑暗或者明亮区域的参数。

在实时渲染中，HDR不仅允许我们超过LDR的范围[0.0, 1.0]与保留更多的细节，同时还让我们能够根据光源的真实强度指定它的强度。比如太阳有比闪光灯之类的东西更高的强度，那么我们为什么不这样子设置呢？(比如说设置一个10.0的漫亮度) 这允许我们用更现实的光照参数恰当地配置一个场景的光照，而这在LDR渲染中是不能实现的，因为他们会被上限约束在1.0。

因为显示器只能显示在0.0到1.0范围之内的颜色，我们肯定要做一些转换从而使得当前的HDR颜色值符合显示器的范围。简单地取平均值重新转换这些颜色值并不能很好的解决这个问题，因为明亮的地方会显得更加显著。我们能做的是用一个不同的方程与/或曲线来转换这些HDR值到LDR值，从而给我们对于场景的亮度完全掌控，这就是之前说的色调变换，也是HDR渲染的最终步骤。

浮点帧缓冲

在实现HDR渲染之前，我们首先需要一些防止颜色值在每一个片段着色器运行后被限制约束的方法。当帧缓冲使用了一个标准化的定点格式(像 `GL_RGB`)为其颜色缓冲的内部格式，OpenGL会在将这些值存入帧缓冲前自动将其约束到0.0到1.0之间。这一操作对大部分帧缓冲格式都是成立的，除了专门用来存放被拓展范围值的浮点格式。

当一个帧缓冲的颜色缓冲的内部格式被设定成了 `GL_RGB16F`, `GL_RGBA16F`, `GL_RGB32F` 或者 `GL_RGBA32F` 时，这些帧缓冲被叫做浮点帧缓冲(Floating Point Framebuffer)，浮点帧缓冲可以存储超过0.0到1.0范围的浮点值，所以非常适合HDR渲染。

想要创建一个浮点帧缓冲，我们只需要改变颜色缓冲的内部格式参数就行了 (注意 `GL_FLOAT` 参数)：

```
glBindTexture(GL_TEXTURE_2D, colorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
```

默认的帧缓冲默认一个颜色分量只占用8位(bits)。当使用一个使用32位每颜色分量的浮点帧缓冲时(使用 `GL_RGB32F` 或者 `GL_RGBA32F`)，我们需要四倍的内存来存储这些颜色。所以除非你需要一个非常高的精确度，32位不是必须的，使用 `GL_RGB16F` 就足够了。

有了一个带有浮点颜色缓冲的帧缓冲，我们可以放心渲染场景到这个帧缓冲中。在这个教程的例子当中，我们先渲染一个光照的场景到浮点帧缓冲中，之后再在一个铺屏四边形(Screen-filling Quad)上应用这个帧缓冲的颜色缓冲，代码会是这样子：

```
glBindFramebuffer(GL_FRAMEBUFFER, hdrFB0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // [...] 渲染(光照的)场景
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 现在使用一个不同的着色器将HDR颜色缓冲渲染至2D铺屏四边形上
hdrShader.Use();
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, hdrColorBufferTexture);
RenderQuad();
```

这里场景的颜色值存在一个可以包含任意颜色值的浮点颜色缓冲中，值可能是超过1.0的。这个简单的演示中，场景被创建为一个被拉伸的立方体通道和四个点光源，其中一个非常亮的在隧道的尽头：

```
std::vector<glm::vec3> lightColors;
lightColors.push_back(glm::vec3(200.0f, 200.0f, 200.0f));
lightColors.push_back(glm::vec3(0.1f, 0.0f, 0.0f));
lightColors.push_back(glm::vec3(0.0f, 0.0f, 0.2f));
lightColors.push_back(glm::vec3(0.0f, 0.1f, 0.0f));
```

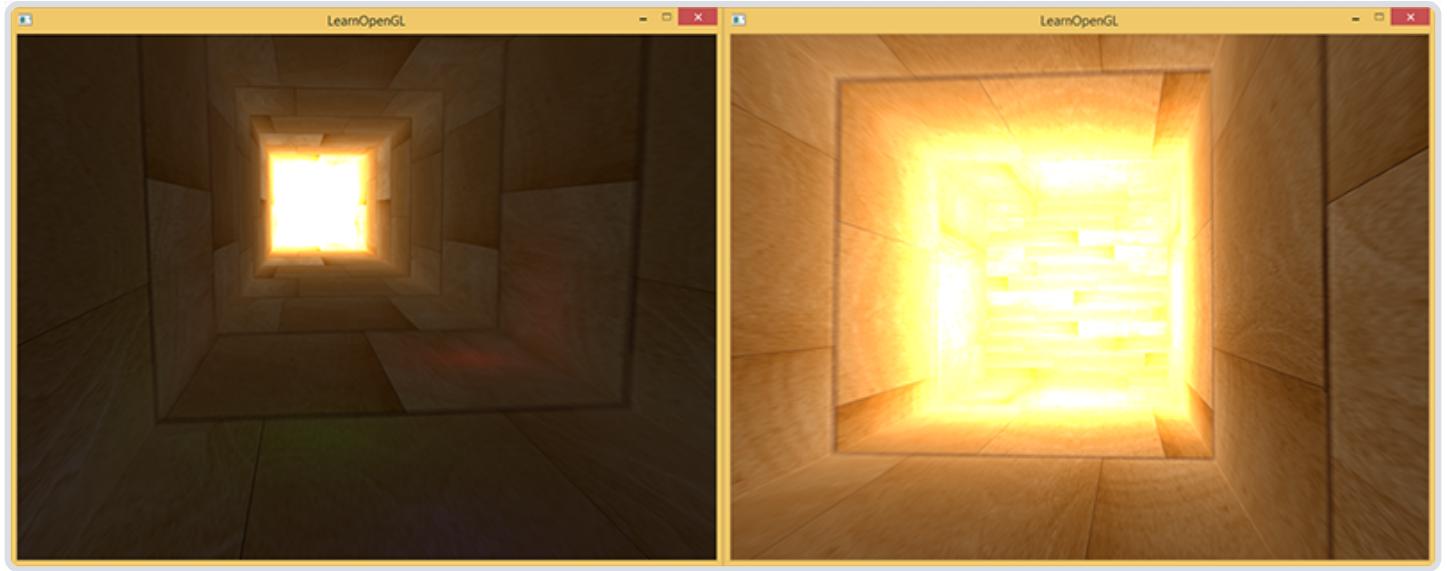
渲染至浮点帧缓冲和渲染至一个普通的帧缓冲是一样的。新的东西就是这个的 `hdrShader` 的片段着色器，用来渲染最终拥有浮点颜色缓冲纹理的2D四边形。我们来定义一个简单的直通片段着色器(Pass-through Fragment Shader)：

```
#version 330 core
out vec4 color;
in vec2 TexCoords;

uniform sampler2D hdrBuffer;

void main()
{
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;
    color = vec4(hdrColor, 1.0);
}
```

这里我们直接采样了浮点颜色缓冲并将其作为片段着色器的输出。然而，这个2D四边形的输出是被直接渲染到默认的帧缓冲中，导致所有片段着色器的输出值被约束在0.0到1.0间，尽管我们已经有了一些存在浮点颜色纹理的值超过了1.0。



很明显，在隧道尽头的强光的值被约束在1.0，因为一大块区域都是白色的，过程中超过1.0的地方损失了所有细节。因为我们直接转换HDR值到LDR值，这就像我们根本就没有应用HDR一样。为了修复这个问题我们需要做的是无损转化所有浮点颜色值回0.0–1.0范围内。我们需要应用到色调映射。

色调映射

色调映射(Tone Mapping)是一个损失很小的转换浮点颜色值至我们所需的LDR[0.0, 1.0]范围内的过程，通常会伴有特定的风格的色平衡(Stylistic Color Balance)。

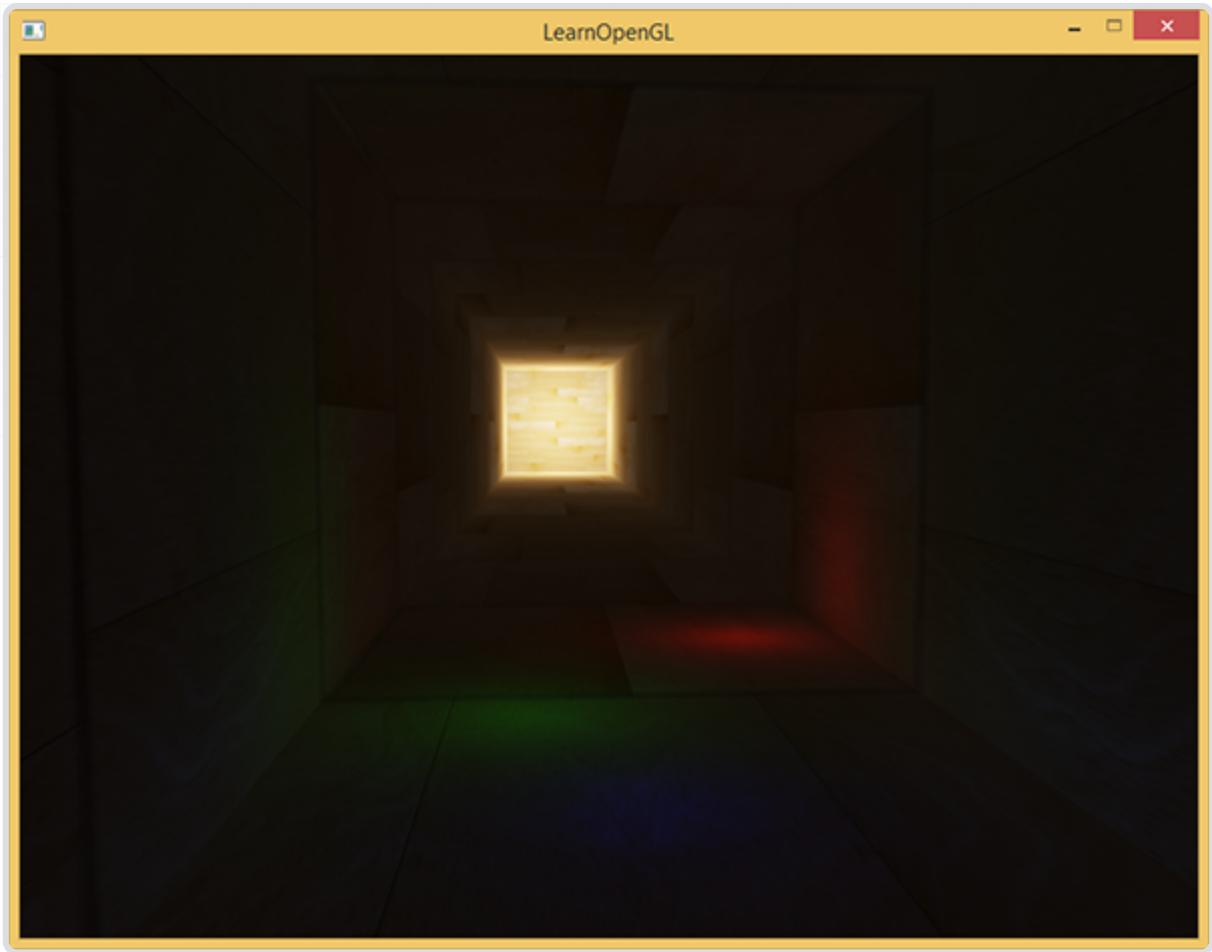
最简单的色调映射算法是Reinhard色调映射，它涉及到分散整个HDR颜色值到LDR颜色值上，所有的值都有对应。Reinhard色调映射算法平均地将所有亮度值分散到LDR上。我们将Reinhard色调映射应用到之前的片段着色器上，并且为了更好的测量加上一个Gamma校正过滤(包括SRGB纹理的使用)：

```
void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;

    // Reinhard色调映射
    vec3 mapped = hdrColor / (hdrColor + vec3(1.0));
    // Gamma校正
    mapped = pow(mapped, vec3(1.0 / gamma));

    color = vec4(mapped, 1.0);
}
```

有了Reinhard色调映射的应用，我们不再会在场景明亮的地方损失细节。当然，这个算法是倾向明亮的区域的，暗的区域会不那么精细也不那么有区分度。



现在你可以看到在隧道的尽头木头纹理变得可见了。用了这个非常简单地色调映射算法，我们可以合适的看到存在浮点帧缓冲中整个范围的HDR值，使我们能在不丢失细节的前提下，对场景光照有精确的控制。

另一个有趣的色调映射应用是曝光(Exposure)参数的使用。你可能还记得之前我们在介绍里讲到的，HDR图片包含在不同曝光等级的细节。如果我们有一个场景要展现日夜交替，我们当然会在白天使用低曝光，在夜间使用高曝光，就像人眼调节方式一样。有了这个曝光参数，我们可以去设置可以同时在白天和夜晚不同光照条件工作的光照参数，我们只需要调整曝光参数就行了。

一个简单的曝光色调映射算法会像这样：

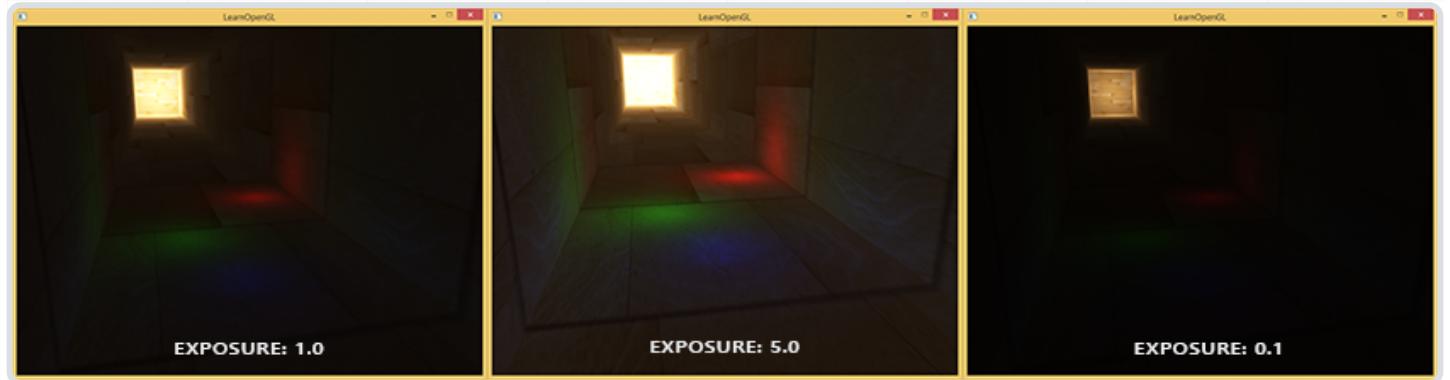
```
uniform float exposure;

void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;

    // 曝光色调映射
    vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure);
    // Gamma校正
    mapped = pow(mapped, vec3(1.0 / gamma));

    color = vec4(mapped, 1.0);
}
```

在这里我们将 `exposure` 定义为默认为1.0的 `uniform`，从而允许我们更加精确设定我们是要注重黑暗还是明亮的区域的HDR颜色值。举例来说，高曝光值会使隧道的黑暗部分显示更多的细节，然而低曝光值会显著减少黑暗区域的细节，但允许我们看到更多明亮区域的细节。下面这组图片展示了在不同曝光值下的通道：



这个图片清晰地展示了HDR渲染的优点。通过改变曝光等级，我们可以看见场景的很多细节，而这些细节可能在LDR渲染中都被丢失了。比如说隧道尽头，在正常曝光下木头结构隐约可见，但用低曝光木头的花纹就可以清晰看见了。对于近处的木头花纹来说，在高曝光下会能更好的看见。

你可以[在这里](#)找到这个演示的源码和HDR的[顶点](#)和[片段着色器](#)。

HDR拓展

在这里展示的两个色调映射算法仅仅是大量(更先进)的色调映射算法中的一小部分，这些算法各有长短。一些色调映射算法倾向于特定的某种颜色/强度，也有一些算法同时显示低高曝光颜色从而能够显示更加多彩和精细的图像。也有一些技巧被称作自动曝光调整(Automatic Exposure Adjustment)或者叫人眼适应(Eye Adaptation)技术，它能够检测前一帧场景的亮度并且缓慢调整曝光参数模仿人眼使得场景在黑暗区域逐渐变亮或者在明亮区域逐渐变暗。

HDR渲染的真正优点在庞大和复杂的场景中应用复杂光照算法会被显示出来，但是出于教学目的创建这样复杂的演示场景是很困难的，这个教程用的场景是很小的，而且缺乏细节。但是如此简单的演示也是能够显示出HDR渲染的一些优点：在明亮和黑暗区域无细节损失，因为它们可以通过色调映射重新获得；多个光照的叠加不会导致亮度被截断的区域的出现，光照可以被设定为它们原来的亮度值而不是被LDR值限制。而且，HDR渲染也使一些有趣的效果更加可行和真实；其中一个效果叫做泛光(Bloom)，我们将在下一节讨论。

附加资源

- [如果泛光效果不被应用HDR渲染还有好处吗？](#)：一个StackExchange问题，其中有一个答案非常详细地解释HDR渲染的好处。
- [什么是色调映射？它与HDR有什么联系？](#)：另一个非常有趣的答案，用了大量图片解释色调映射。

2.6.7 泛光

原文 [Bloom](#)

作者 JoeyDeVries

翻译 [Django](#)

校对 gwy_1992

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

明亮的光源和区域经常很难向观察者表达出来，因为监视器的亮度范围是有限的。一种区分明亮光源的方式是使它们在监视器上发出光芒，光源的光芒向四周发散。这样观察者就会产生光源或亮区的确是强光区。（译注：这个问题的提出简单来说是为了解决这样的问题：例如有一张在阳光下的白纸，白纸在监视器上显示出是出白色，而前方的太阳也是纯白色的，所以基本上白纸和太阳就是一样的了，给太阳加一个光晕，这样太阳看起来似乎就比白纸更亮了）

光晕效果可以使用一个后处理特效泛光来实现。泛光使所有明亮区域产生光晕效果。下面是一个使用了和没有使用光晕的对比（图片生成自虚幻引擎）：

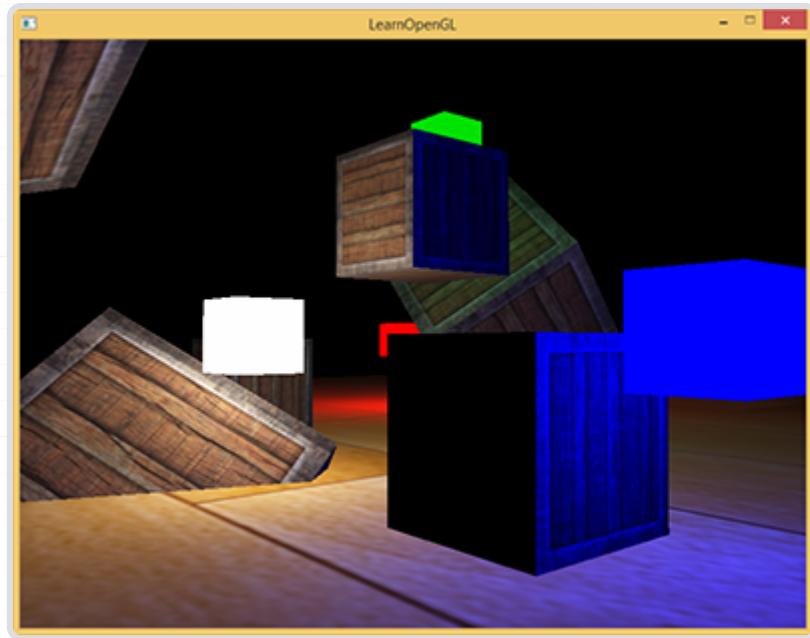


Bloom是我们能够注意到一个明亮的物体真的有种明亮的感觉。泛光可以极大提升场景中的光照效果，并提供了极大的效果提升，尽管做到这一切只需一点改变。

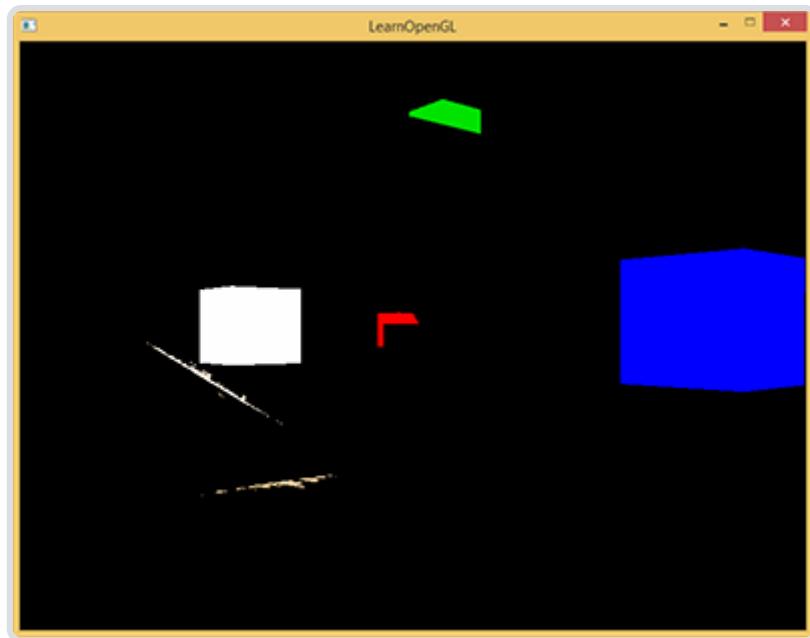
Bloom和HDR结合使用效果很好。常见的一个误解是HDR和泛光是一样的，很多人认为两种技术是可以互换的。但是它们是两种不同的技术，用于各自不同的目的上。可以使用默认的8位精确度的帧缓冲，也可以在不使用泛光效果的时候，使用HDR。只不过在有了HDR之后再实现泛光就更简单了。

为实现泛光，我们像平时那样渲染一个有光场景，提取出场景的HDR颜色缓冲以及只有这个场景明亮区域可见的图片。被提取的带有亮度的图片接着被模糊，结果被添加到HDR场景上面。

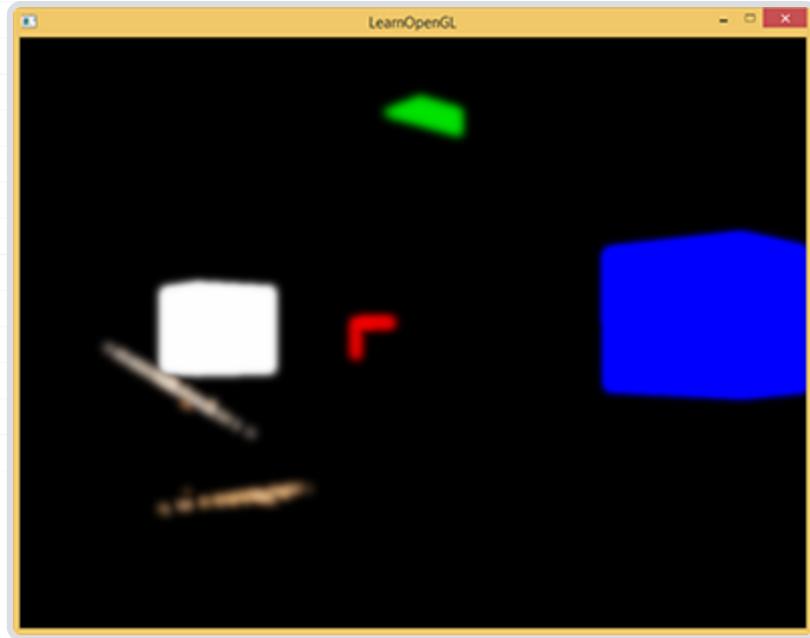
我们来一步一步解释这个处理过程。我们在场景中渲染一个带有4个立方体形式不同颜色的明亮的光源。带有颜色的发光立方体的亮度在1.5到15.0之间。如果我们将其渲染至HDR颜色缓冲，场景看起来会是这样的：



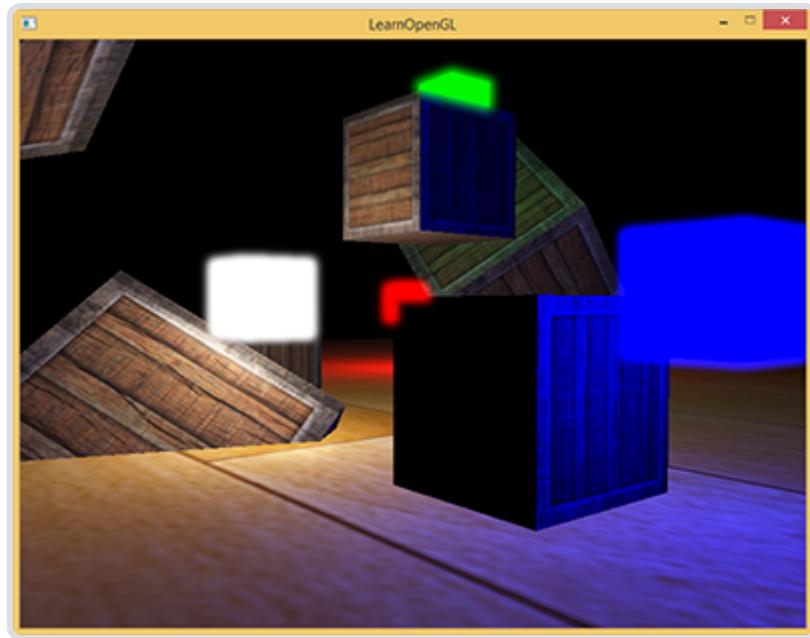
我们得到这个HDR颜色缓冲纹理，提取所有超出一定亮度的fragment。这样我们就会获得一个只有fragment超过了一定阈限的颜色区域：



我们将这个超过一定亮度阈限的纹理进行模糊。泛光效果的强度很大程度上是由被模糊过滤器的范围和强度所决定。

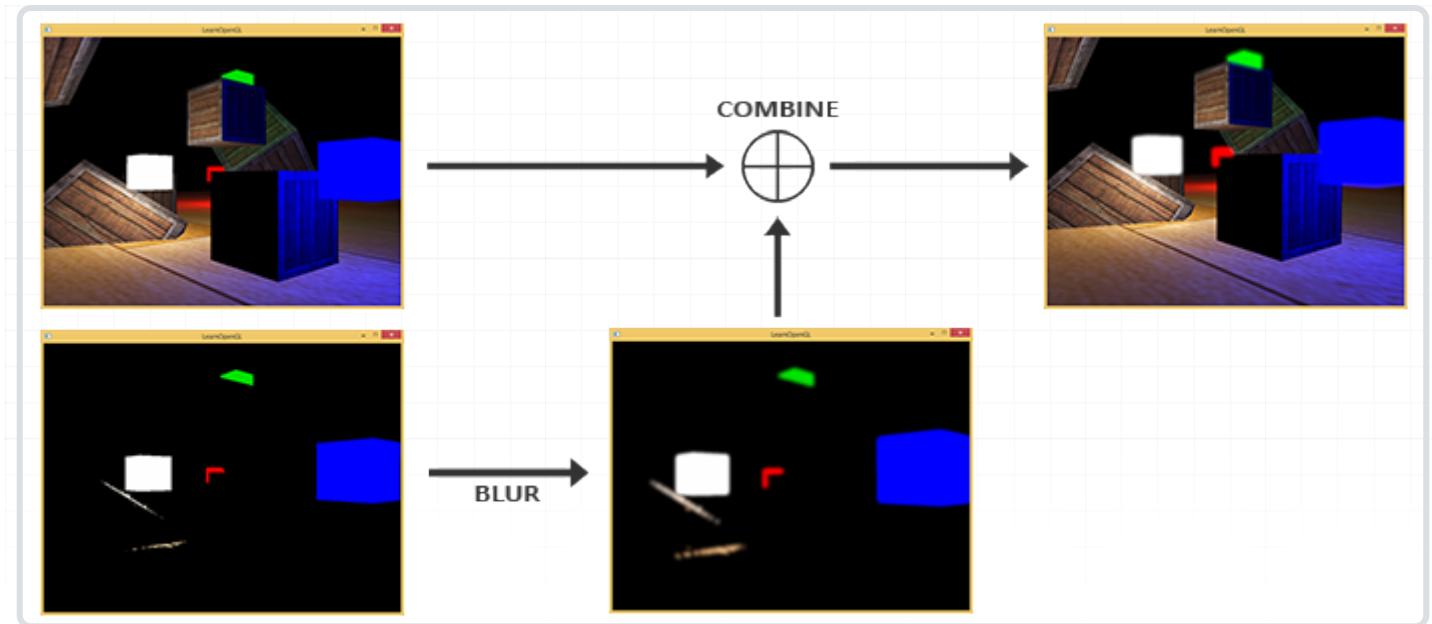


最终的被模糊化的纹理就是我们用来获得发出光晕效果的东西。这个已模糊的纹理要添加到原来的HDR场景纹理之上。因为模糊过滤器的应用明亮区域发出光晕，所以明亮区域在长和宽上都有所扩展。



泛光本身并不是个复杂的技术，但很难获得正确的效果。它的品质很大程度上取决于所用的模糊过滤器的质量和类型。简单地改改模糊过滤器就会极大的改变泛光效果的品质。

下面这几步就是泛光后处理特效的过程，它总结了实现泛光所需的步骤。



首先我们需要根据一定的阈限提取所有明亮的颜色。我们先来做这件事。

提取亮色

第一步我们要从渲染出来的场景中提取两张图片。我们可以渲染场景两次，每次使用一个不同的着色器渲染到不同的帧缓冲中，但我们可以使用一个叫做MRT（Multiple Render Targets，多渲染目标）的小技巧，这样我们就能指定多个像素着色器输出；有了它我们还能够在单独渲染处理中提取头两个图片。在像素着色器的输出前，我们指定一个布局location标识符，这样我们便可控制一个像素着色器写入到哪个颜色缓冲：

```
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;
```

只有我们真的具有多个地方可写的时候这才能工作。使用多个像素着色器输出的必要条件是，有多个颜色缓冲附加到了当前绑定的帧缓冲对象上。你可能从帧缓冲教程那里回忆起，当把一个纹理链接到帧缓冲的颜色缓冲上时，我们可以指定一个颜色附件。直到现在，我们一直使用着GL_COLOR_ATTACHMENT0，但通过使用GL_COLOR_ATTACHMENT1，我们可以得到一个附加了两个颜色缓冲的帧缓冲对象：

```
// Set up floating point framebuffer to render scene to
GLuint hdrFBO;
 glGenFramebuffers(1, &hdrFBO);
 glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
 GLuint colorBuffers[2];
 glGenTextures(2, colorBuffers);
 for (GLuint i = 0; i < 2; i++)
 {
    glBindTexture(GL_TEXTURE_2D, colorBuffers[i]);
    glTexImage2D(
        GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL
    );
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    // attach texture to framebuffer
    glFramebufferTexture2D(
        GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + i, GL_TEXTURE_2D, colorBuffers[i], 0
    );
 }
```

我们需要显式告知OpenGL我们正在通过glDrawBuffers渲染到多个颜色缓冲，否则OpenGL只会渲染到帧缓冲的第一个颜色附件，而忽略所有其他的。我们可以通过传递多个颜色附件的枚举来做这件事，我们以下面的操作进行渲染：

```
GLuint attachments[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, attachments);
```

当渲染到这个帧缓冲的时候，一个着色器使用一个布局location修饰符，fragment就会写入对应的颜色缓冲。这很棒，因为这样省去了我们为提取明亮区域的额外渲染步骤，因为我们现在可以直接从将被渲染的fragment提取出它们：

```
#version 330 core
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;

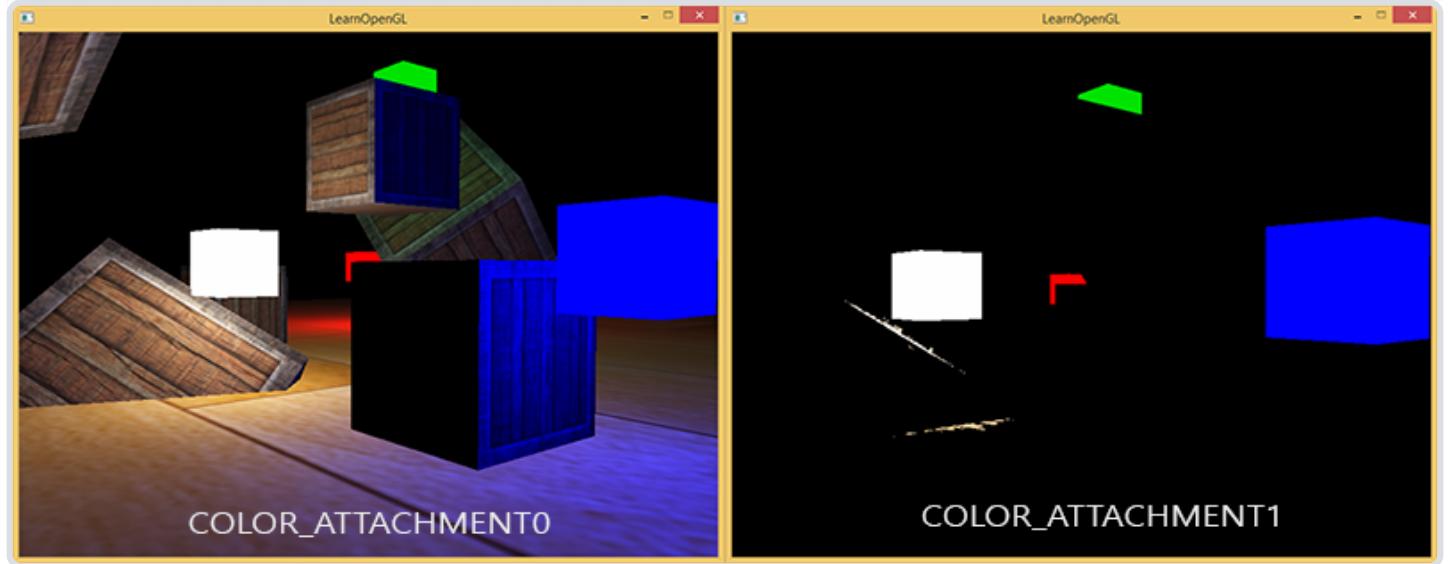
[...]

void main()
{
    [...] // first do normal lighting calculations and output results
    FragColor = vec4(lighting, 1.0f);
    // Check whether fragment output is higher than threshold, if so output as brightness color
    float brightness = dot(FragColor.rgb, vec3(0.2126, 0.7152, 0.0722));
    if(brightness > 1.0)
        BrightColor = vec4(FragColor.rgb, 1.0);
}
```

这里我们先正常计算光照，将其传递给第一个像素着色器的输出变量FragColor。然后我们使用当前储存在FragColor的东西来决定它的亮度是否超过了一定阈限。我们通过恰当地将其转为灰度的方式计算一个fragment的亮度，如果它超过了一定阈限，我们就把颜色输出到第二个颜色缓冲，那里保存着所有亮部；渲染发光的立方体也是一样的。

这也说明了为什么泛光在HDR基础上能够运行得很好。因为HDR中，我们可以将颜色值指定超过1.0这个默认的范围，我们能够得到对一个图像中的亮度的更好的控制权。没有HDR我们必须将阈限设置为小于1.0的数，虽然可行，但是亮部很容易变得很多，这就导致光晕效果过重。

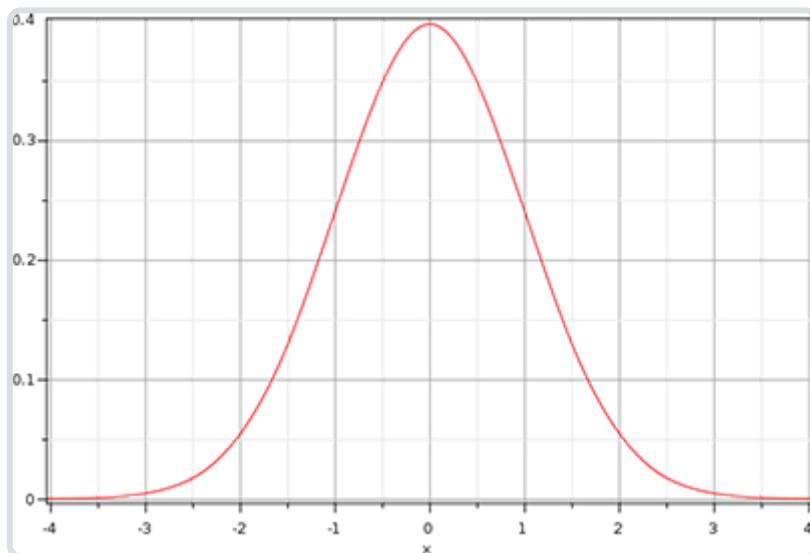
有了两个颜色缓冲，我们就有了一个正常场景的图像和一个提取出的亮区的图像；这些都在一个渲染步骤中完成。



有了一个提取出的亮区图像，我们现在就要把这个图像进行模糊处理。我们可以使用帧缓冲教程后处理部分的那个简单的盒子过滤器，但不过我们最好还是使用一个更高级的更漂亮的模糊过滤器：高斯模糊(Gaussian blur)。

高斯模糊

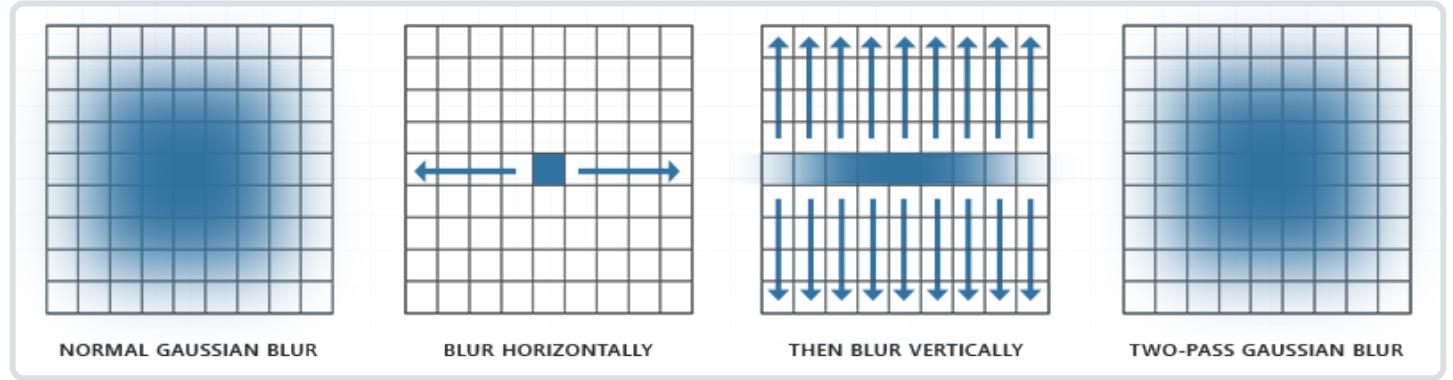
在后处理教程那里，我们采用的模糊是一个图像中所有周围像素的均值，它的确为我们提供了一个简易实现的模糊，但是效果并不好。高斯模糊基于高斯曲线，高斯曲线通常被描述为一个钟形曲线，中间的值达到最大化，随着距离的增加，两边的值不断减少。高斯曲线在数学上有不同的形式，但是通常是这样的形状：



高斯曲线在它的中间处的面积最大，使用它的值作为权重使得近处的样本拥有最大的优先权。比如，如果我们从fragment的32×32的四方形区域采样，这个权重随着和fragment的距离变大逐渐减小；通常这会得到更好更真实的模糊效果，这种模糊叫做高斯模糊。

要实现高斯模糊过滤我们需要一个二维四方形作为权重，从这个二维高斯曲线方程中去获取它。然而这个过程有个问题，就是很快会消耗极大的性能。以一个 32×32 的模糊kernel为例，我们必须对每个fragment从一个纹理中采样1024次！

幸运的是，高斯方程有个非常巧妙的特性，它允许我们把二维方程分解为两个更小的方程：一个描述水平权重，另一个描述垂直权重。我们首先用水平权重在整个纹理上进行水平模糊，然后在经改变的纹理上进行垂直模糊。利用这个特性，结果是一样的，但是可以节省难以置信的性能，因为我们现在只需做 $32+32$ 次采样，不再是1024了！这叫做两步高斯模糊。



这意味着我们如果对一个图像进行模糊处理，至少需要两步，最好使用帧缓冲对象做这件事。具体来说，我们将实现像乒乓球一样的帧缓冲来实现高斯模糊。它的意思是，有一对儿帧缓冲，我们把另一个帧缓冲的颜色缓冲放进当前的帧缓冲的颜色缓冲中，使用不同的着色效果渲染指定的次数。基本上就是不断地切换帧缓冲和纹理去绘制。这样我们先在场景纹理的第一个缓冲中进行模糊，然后在把第一个帧缓冲的颜色缓冲放进第二个帧缓冲进行模糊，接着，将第二个帧缓冲的颜色缓冲放进第一个，循环往复。

在我们研究帧缓冲之前，先讨论高斯模糊的像素着色器：

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D image;

uniform bool horizontal;

uniform float weight[5] = float[] (0.227027, 0.1945946, 0.1216216, 0.054054, 0.016216);

void main()
{
    vec2 tex_offset = 1.0 / textureSize(image, 0); // gets size of single texel
    vec3 result = texture(image, TexCoords).rgb * weight[0]; // current fragment's contribution
    if(horizontal)
    {
        for(int i = 1; i < 5; ++i)
        {
            result += texture(image, TexCoords + vec2(tex_offset.x * i, 0.0)).rgb * weight[i];
            result += texture(image, TexCoords - vec2(tex_offset.x * i, 0.0)).rgb * weight[i];
        }
    }
    else
    {
        for(int i = 1; i < 5; ++i)
        {
            result += texture(image, TexCoords + vec2(0.0, tex_offset.y * i)).rgb * weight[i];
            result += texture(image, TexCoords - vec2(0.0, tex_offset.y * i)).rgb * weight[i];
        }
    }
    FragColor = vec4(result, 1.0);
}
```

这里我们使用一个比较小的高斯权重做例子，每次我们用它来指定当前fragment的水平或垂直样本的特定权重。你会发现我们基本上是将模糊过滤器根据我们在uniform变量horizontal设置的值分割为一个水平和一个垂直部分。通过用1.0除以纹理的大小（从textureSize得到一个vec2）得到一个纹理像素的实际大小，以此作为偏移距离的根据。

我们为图像的模糊处理创建两个基本的帧缓冲，每个只有一个颜色缓冲纹理：

```

GLuint pingpongFB0[2];
GLuint pingpongBuffer[2];
 glGenFramebuffers(2, pingpongFB0);
 glGenTextures(2, pingpongBuffer);
 for (GLuint i = 0; i < 2; i++)
 {
    glBindFramebuffer(GL_FRAMEBUFFER, pingpongFB0[i]);
    glBindTexture(GL_TEXTURE_2D, pingpongBuffer[i]);
    glTexImage2D(
        GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL
    );
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glFramebufferTexture2D(
        GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, pingpongBuffer[i], 0
    );
 }
}

```

得到一个HDR纹理后，我们用提取出来的亮区纹理填充一个帧缓冲，然后对其模糊处理10次（5次垂直5次水平）：

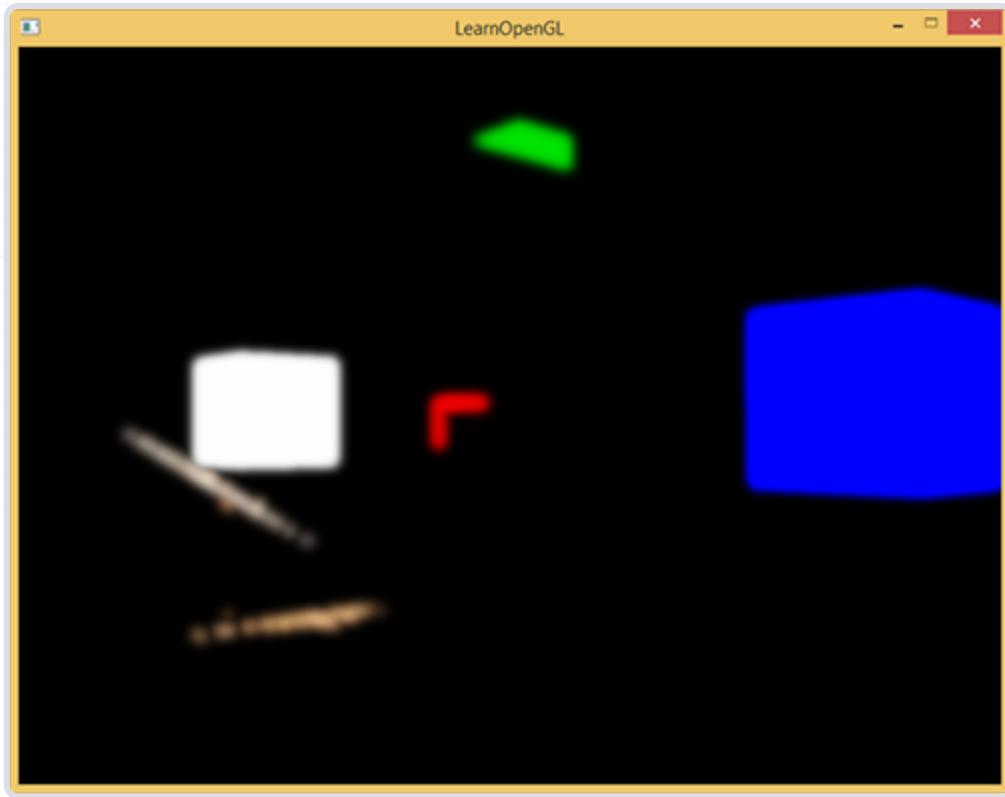
```

GLboolean horizontal = true, first_iteration = true;
GLuint amount = 10;
shaderBlur.Use();
for (GLuint i = 0; i < amount; i++)
{
    glBindFramebuffer(GL_FRAMEBUFFER, pingpongFB0[horizontal]);
    glUniform1i(glGetUniformLocation(shaderBlur.Program, "horizontal"), horizontal);
    glBindTexture(
        GL_TEXTURE_2D, first_iteration ? colorBuffers[1] : pingpongBuffers[!horizontal]
    );
    RenderQuad();
    horizontal = !horizontal;
    if (first_iteration)
        first_iteration = false;
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

每次循环我们根据我们打算渲染的是水平还是垂直来绑定两个缓冲其中之一，而将另一个绑定为纹理进行模糊。第一次迭代，因为两个颜色缓冲都是空的所以我们随意绑定一个去进行模糊处理。重复这个步骤10次，亮区图像就进行一个重复5次的高斯模糊了。这样我们可以对任意图像进行任意次模糊处理；高斯模糊循环次数越多，模糊的强度越大。

通过对提取亮区纹理进行5次模糊，我们就得到了一个正确的模糊的场景亮区图像。



泛光的最后一步是把模糊处理的图像和场景原来的HDR纹理进行结合。

把两个纹理混合

有了场景的HDR纹理和模糊处理的亮区纹理，我们只需把它们结合起来就能实现泛光或称光晕效果了。最终的像素着色器（大部分和HDR教程用的差不多）要把两个纹理混合：

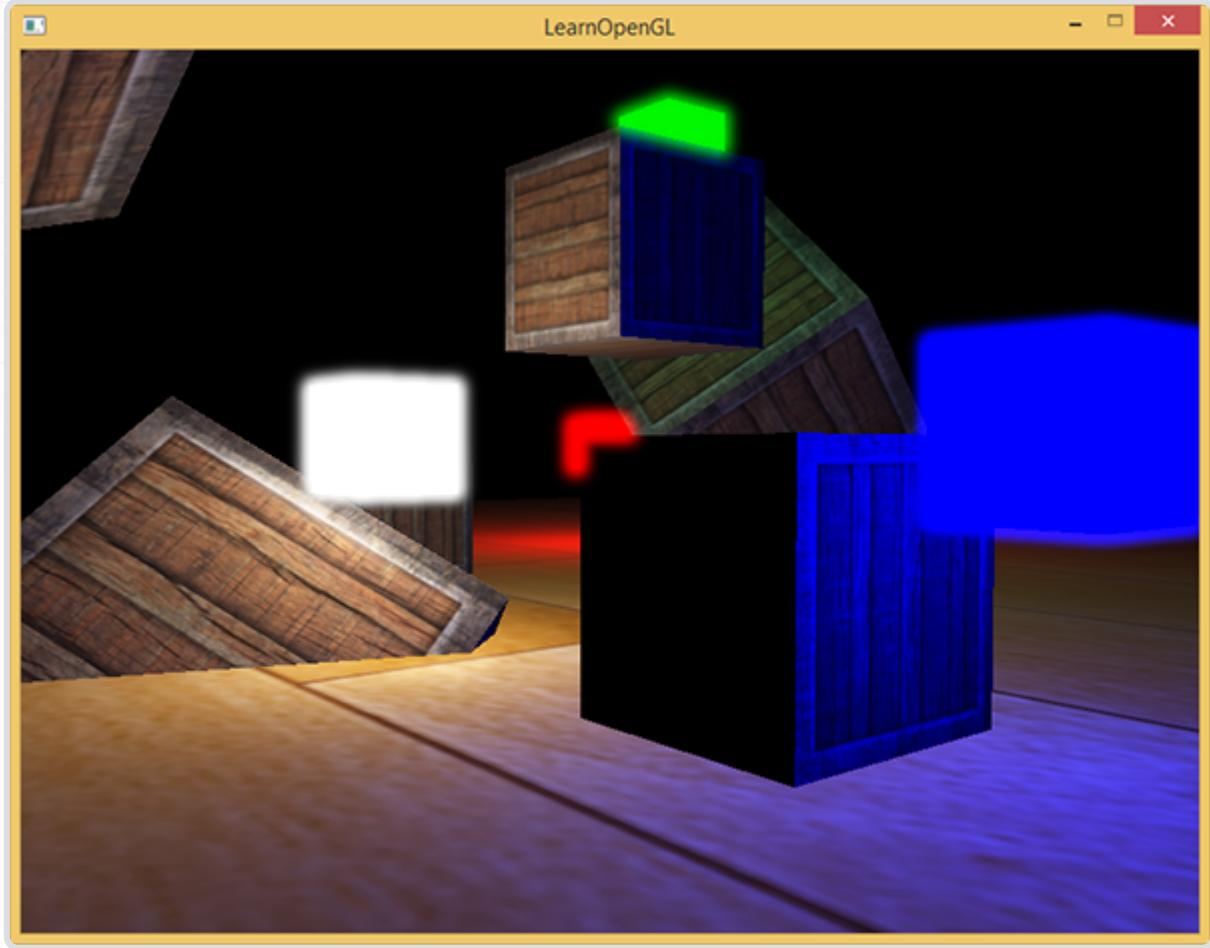
```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D scene;
uniform sampler2D bloomBlur;
uniform float exposure;

void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(scene, TexCoords).rgb;
    vec3 bloomColor = texture(bloomBlur, TexCoords).rgb;
    hdrColor += bloomColor; // additive blending
    // tone mapping
    vec3 result = vec3(1.0) - exp(-hdrColor * exposure);
    // also gamma correct while we're at it
    result = pow(result, vec3(1.0 / gamma));
    FragColor = vec4(result, 1.0f);
}
```

要注意的是我们要在应用色调映射之前添加泛光效果。这样添加的亮区的泛光，也会柔和转换为LDR，光照效果相对会更好。

把两个纹理结合以后，场景亮区便有了合适的光晕特效：



有颜色的立方体看起来仿佛更亮，它向外发射光芒，的确是一个更好的视觉效果。这个场景比较简单，所以泛光效果不算十分令人瞩目，但在更好的场景中合理配置之后效果会有巨大的不同。你可以在这里找到这个简单的例子的源码，以及模糊的顶点和像素着色器、立方体的像素着色器、后处理的顶点和像素着色器。

这个教程我们只是用了一个相对简单的高斯模糊过滤器，它在每个方向上只有5个样本。通过沿着更大的半径或重复更多次数的模糊，进行采样我们就可以提升模糊的效果。因为模糊的质量与泛光效果的质量正相关，提升模糊效果就能够提升泛光效果。有些提升将模糊过滤器与不同大小的模糊kernel或采用多个高斯曲线来选择性地结合权重结合起来使用。来自Kalogirou和EpicGames的附加资源讨论了如何通过提升高斯模糊来显著提升泛光效果。

附加资源

- [Efficient Gaussian Blur with linear sampling](#): 非常详细地描述了高斯模糊，以及如何使用OpenGL的双线性纹理采样提升性能。
- [Bloom Post Process Effect](#): 来自Epic Games关于通过对权重的多个高斯曲线结合来提升泛光效果的文章。
- [How to do good bloom for HDR rendering](#): Kalogirou的文章描述了如何使用更好的高斯模糊算法来提升泛光效果。

2.6.8 延迟着色法

原文 [Deferred Shading](#)

作者 JoeyDeVries

翻译 Krasjet

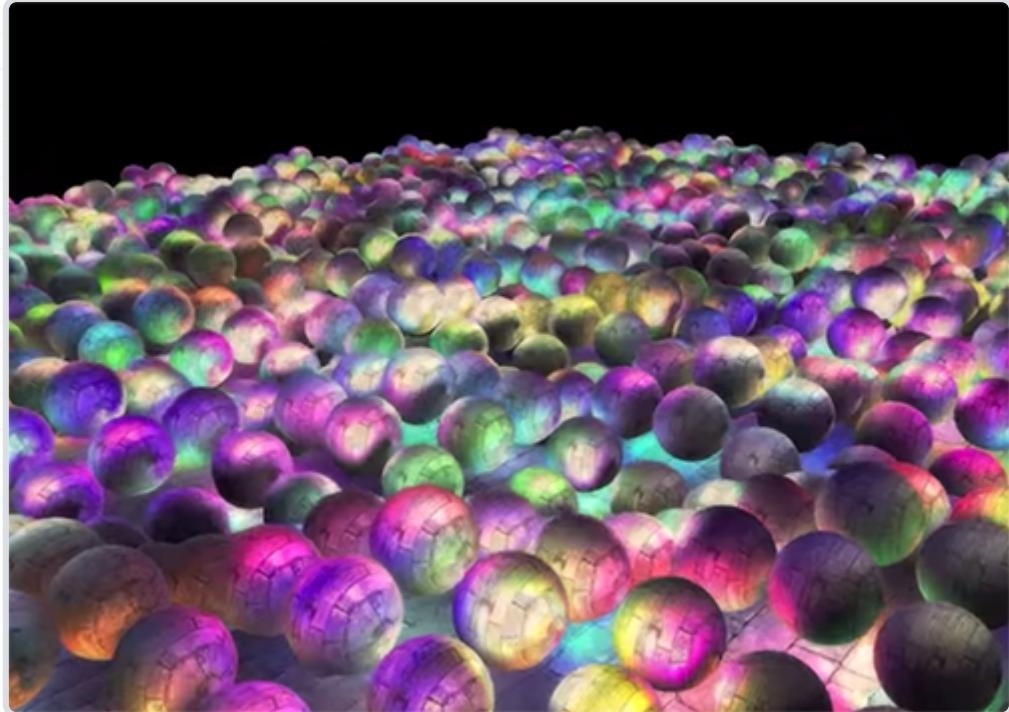
校对 [KenLee](#)

Note

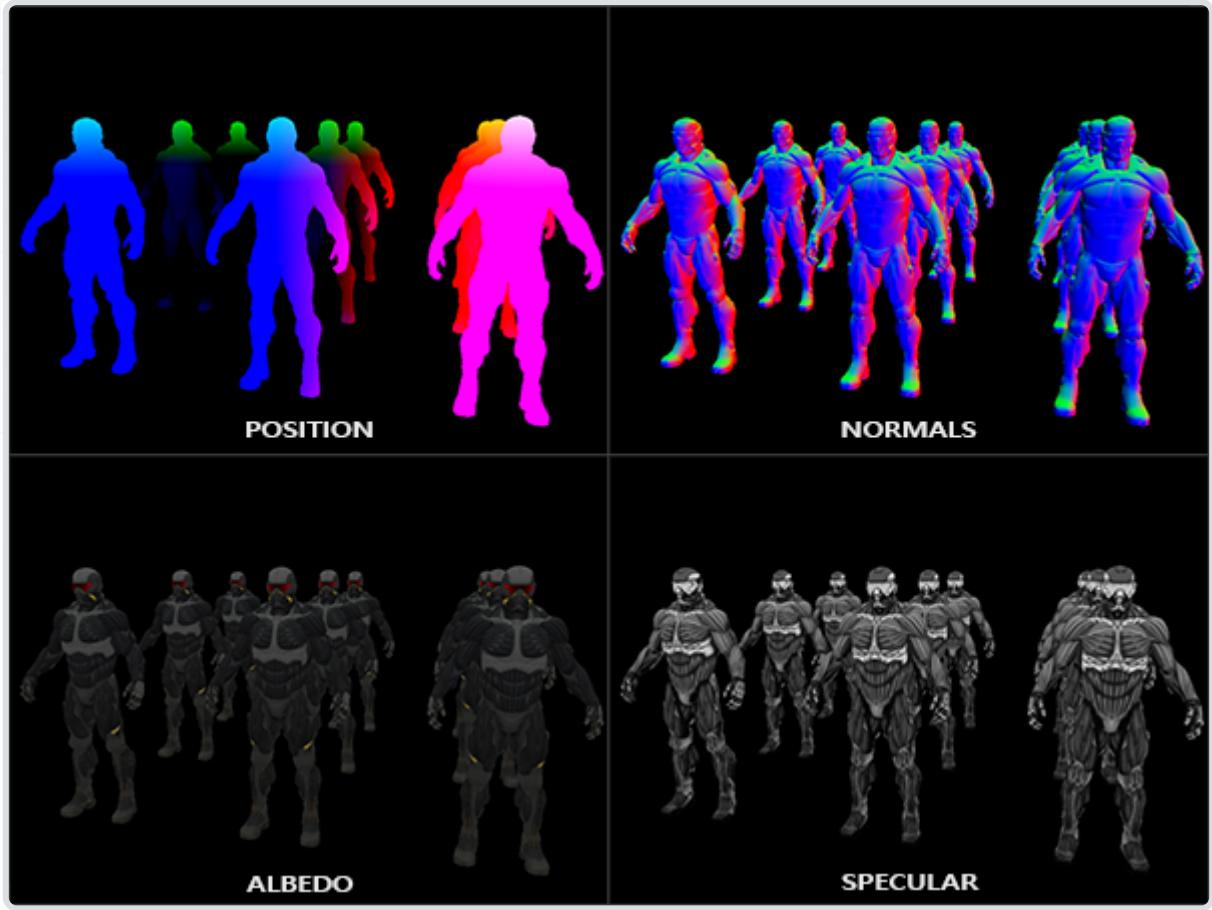
本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

我们现在一直使用的光照方式叫做正向渲染(Forward Rendering)或者正向着色法(Forward Shading)，它是我们渲染物体的一种非常直接的方式，在场景中我们根据所有光源照亮一个物体，之后再渲染下一个物体，以此类推。它非常容易理解，也很容易实现，但是同时它对程序性能的影响也很大，因为对于每一个需要渲染的物体，程序都要对每一个光源每一个需要渲染的片段进行迭代，这是非常多的！因为大部分片段着色器的输出都会被之后的输出覆盖，正向渲染还会在场景中因为高深的复杂度(多个物体重合在一个像素上)浪费大量的片段着色器运行时间。

延迟着色法(Deferred Shading)，或者说是延迟渲染(Deferred Rendering)，为了解决上述问题而诞生了，它大幅度地改变了我们渲染物体的方式。这给我们优化拥有大量光源的场景提供了很多的选择，因为它能够在渲染上百甚至上千光源的同时还能够保持能让人接受的帧率。下面这张图片包含了一共1874个点光源，它是使用延迟着色法来完成的，而这对于正向渲染几乎是不可能的(图片来源：Hannes Nevalainen)。

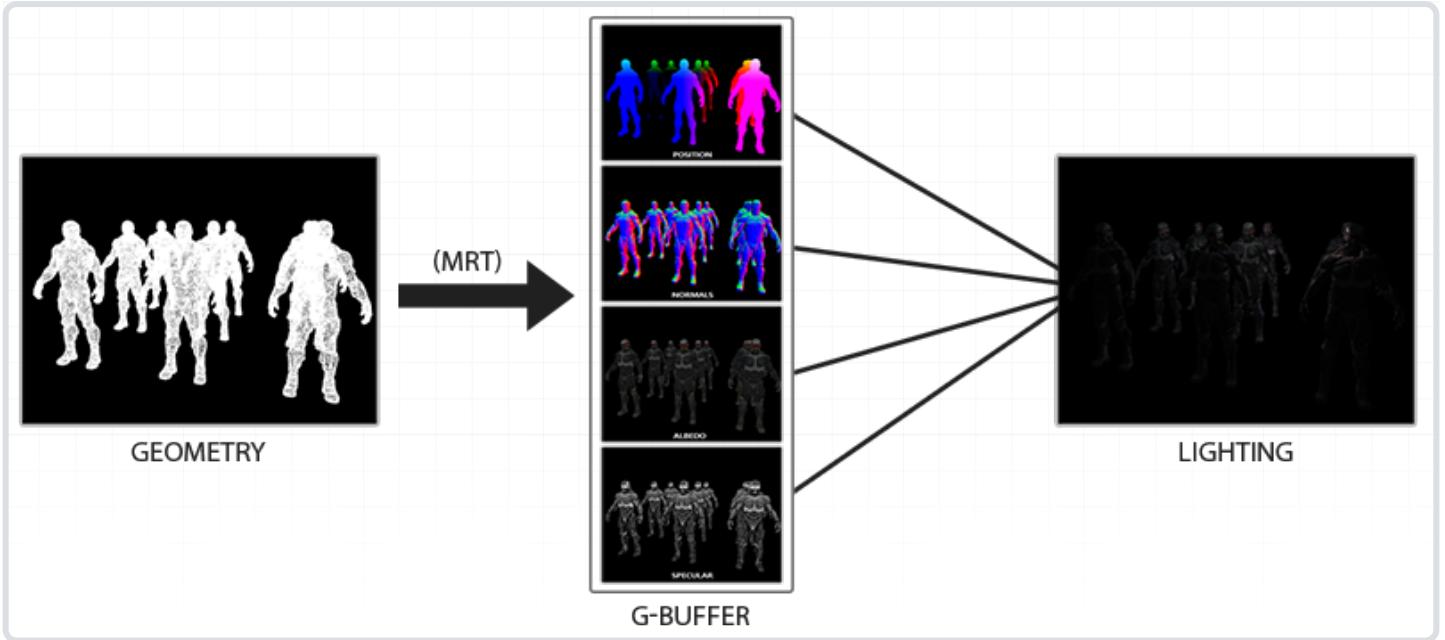


延迟着色法基于我们延迟(Defer)或推迟(Postpone)大部分计算量非常大的渲染(像是光照)到后期进行处理的想法。它包含两个处理阶段(Pass)：在第一个几何处理阶段(Geometry Pass)中，我们先渲染场景一次，之后获取对象的各种几何信息，并储存在一系列叫做G缓冲(G-buffer)的纹理中；想想位置向量(Position Vector)、颜色向量(Color Vector)、法向量(Normal Vector)和/或镜面值(Specular Value)。场景中这些储存在G缓冲中的几何信息将会在之后用来做(更复杂的)光照计算。下面是一帧中G缓冲的内容：



我们会在第二个光照处理阶段(Lighting Pass)中使用G缓冲内的纹理数据。在光照处理阶段中，我们渲染一个屏幕大小的方形，并使用G缓冲中的几何数据对每一个片段计算场景的光照；在每个像素中我们都会对G缓冲进行迭代。我们对于渲染过程进行解耦，将它高级的片段处理挪到后期进行，而不是直接将每个对象从顶点着色器带到片段着色器。光照计算过程还是和我们以前一样，但是现在我们需要从对应的G缓冲而不是顶点着色器(和一些uniform变量)那里获取输入变量了。

下面这幅图片很好地展示了延迟着色法的整个过程：



这种渲染方法一个很大的好处就是能保证在G缓冲中的片段和在屏幕上呈现的像素所包含的片段信息是一样的，因为深度测试已经最终将这里的片段信息作为最顶层的片段。这样保证了对于在光照处理阶段中处理的每一个像素都只处理一次，所以我们能够省下很多无用的渲染调用。除此之外，延迟渲染还允许我们做更多的优化，从而渲染更多的光源。

当然这种方法也带来几个缺陷，由于G缓冲要求我们在纹理颜色缓冲中存储相对比较大的场景数据，这会消耗比较多的显存，尤其是类似位置向量之类的需要高精度的场景数据。另外一个缺点就是他不支持混色(因为我们只有最前面的片段信息)，因此也不能使用MSAA了。针对这几个问题我们可以做一些变通来克服这些缺点，这些我们留会在教程的最后讨论。

在几何处理阶段中填充G缓冲非常高效，因为我们直接储存像素位置，颜色或者是法线等对象信息到帧缓冲中，而这几乎不会消耗处理时间。在此基础上使用多渲染目标(Multiple Render Targets, MRT)技术，我们甚至可以在一个渲染处理之内完成这所有的工作。

G缓冲

G缓冲(G-buffer)是对所有用来储存光照相关的数据，并在最后的光照处理阶段中使用的所有纹理的总称。趁此机会，让我们顺便复习一下在正向渲染中照亮一个片段所需要的所有数据：

- 一个3D位置向量来计算(插值)片段位置变量供 `lightDir` 和 `viewDir` 使用
- 一个RGB漫反射颜色向量，也就是反照率(Albedo)
- 一个3D法向量来判断平面的斜率
- 一个镜面强度(Specular Intensity)浮点值
- 所有光源的位置和颜色向量
- 玩家或者观察者的位置向量

有了这些(逐片段)变量的处置权，我们就能够计算我们很熟悉的(布林-)冯氏光照(Blinn-Phong Lighting)了。光源的位置，颜色，和玩家的观察位置可以通过uniform变量来设置，但是其它变量对于每个对象的片段都是不同的。如果我们能以某种方式传输完全相同的数据到最终的延迟光照处理阶段中，我们就能计算与之前相同的光照效果了，尽管我们只是在渲染一个2D方形的片段。

OpenGL并没有限制我们能在纹理中能存储的东西，所以现在你应该清楚在一个或多个屏幕大小的纹理中储存所有逐片段数据并在之后光照处理阶段中使用的可行性了。因为G缓冲纹理将会和光照处理阶段中的2D方形一样大，我们会获得和正向渲染设置完全一样的片段数据，但在光照处理阶段这里是一对一映射。

整个过程在伪代码中会是这样的：

```
while(...) // 游戏循环
{
    // 1. 几何处理阶段：渲染所有的几何/颜色数据到G缓冲
    glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    gBufferShader.Use();
    for(Object obj : Objects)
    {
        ConfigureShaderTransformsAndUniforms();
        obj.Draw();
    }
    // 2. 光照处理阶段：使用G缓冲计算场景的光照
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    lightingPassShader.Use();
    BindAllGBufferTextures();
    SetLightingUniforms();
    RenderQuad();
}
```

对于每一个片段我们需要储存的数据有：一个位置向量、一个法向量，一个颜色向量，一个镜面强度值。所以我们在几何处理阶段中需要渲染场景中所有的对象并储存这些数据分量到G缓冲中。我们可以再次使用多渲染目标(**Multiple Render Targets**)来在一个渲染处理之内渲染多个颜色缓冲，在之前的[泛光教程](#)中我们也简单地提及了它。

对于几何渲染处理阶段，我们首先需要初始化一个帧缓冲对象，我们很直观的称它为 `gBuffer`，它包含了多个颜色缓冲和一个单独的深度渲染缓冲对象(Depth Renderbuffer Object)。对于位置和法向量的纹理，我们希望使用高精度的纹理(每分量16或32位的浮点数)，而对于反照率和镜面值，使用默认的纹理(每分量8位浮点数)就够了。

```

GLuint gBuffer;
 glGenFramebuffers(1, &gBuffer);
 glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
 GLuint gPosition, gNormal, gColorSpec;

// - 位置颜色缓冲
 glGenTextures(1, &gPosition);
 glBindTexture(GL_TEXTURE_2D, gPosition);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, gPosition, 0

// - 法线颜色缓冲
 glGenTextures(1, &gNormal);
 glBindTexture(GL_TEXTURE_2D, gNormal);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, gNormal, 0);

// - 颜色 + 镜面颜色缓冲
 glGenTextures(1, &gAlbedoSpec);
 glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, gAlbedoSpec, 0);

// - 告诉OpenGL我们将要使用(帧缓冲的)哪种颜色附件来进行渲染
 GLuint attachments[3] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2 };
 glDrawBuffers(3, attachments);

// 之后同样添加渲染缓冲对象(Render Buffer Object)为深度缓冲(Depth Buffer)，并检查完整性
 [...]

```

由于我们使用了多渲染目标，我们需要显式告诉OpenGL我们需要使用 `glDrawBuffers` 渲染的是和 `GBuffer` 关联的那个颜色缓冲。同样需要注意的是，我们使用RGB纹理来储存位置和法线的数据，因为每个对象只有三个分量；但是我们将颜色和镜面强度数据合并到一起，存储到一个单独的RGBA纹理里面，这样我们就不需要声明一个额外的颜色缓冲纹理了。随着你的延迟渲染管线变得越来越复杂，需要更多的数据的时候，你就会很快发现新的方式来组合数据到一个单独的纹理当中。

接下来我们需要渲染它们到G缓冲中。假设每个对象都有漫反射，一个法线和一个镜面强度纹理，我们会想使用一些像下面这个片段着色器的东西来渲染它们到G缓冲中去。

```
#version 330 core
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;

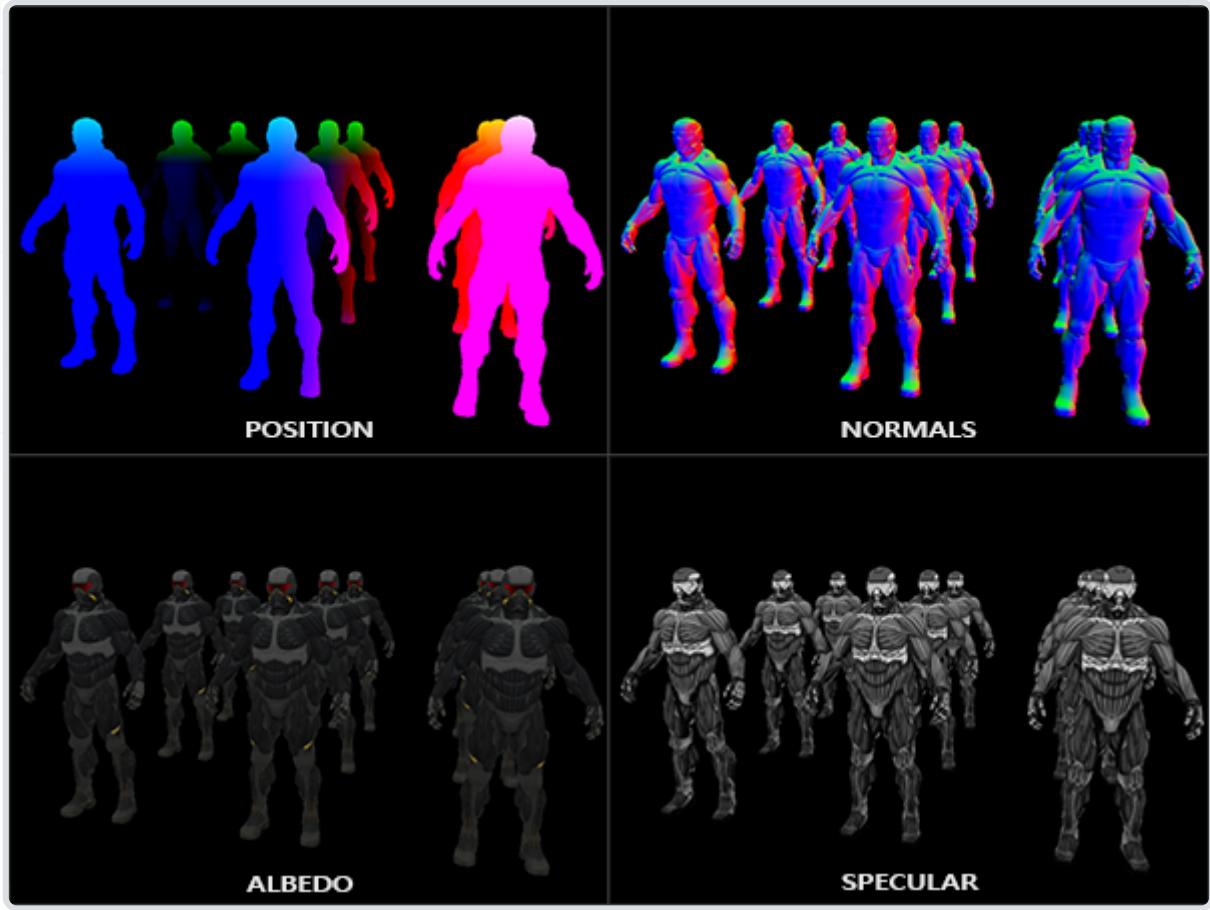
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_specular1;

void main()
{
    // 存储第一个G缓冲纹理中的片段位置向量
    gPosition = FragPos;
    // 同样存储对每个逐片段法线到G缓冲中
    gNormal = normalize(Normal);
    // 和漫反射对每个逐片段颜色
    gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;
    // 存储镜面强度到gAlbedoSpec的alpha分量
    gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
}
```

因为我们使用了多渲染目标，这个布局指示符(Layout Specifier)告诉了OpenGL我们需要渲染到当前的活跃帧缓冲中的哪一个颜色缓冲。注意我们并没有储存镜面强度到一个单独的颜色缓冲纹理中，因为我们可以储存它单独的浮点值到其它颜色缓冲纹理的alpha分量中。

请记住，因为有光照计算，所以保证所有变量在一个坐标空间当中至关重要。在这里我们在世界空间中存储(并计算)所有的变量。

如果我们现在想要渲染一大堆纳米装战士对象到 `gBuffer` 帧缓冲中，并通过一个一个分别投影它的颜色缓冲到铺屏四边形中尝试将它们显示出来，我们会看到向下面这样的东西：



尝试想象世界空间位置和法向量都是正确的。比如说，指向右侧的法向量将会被更多地对齐到红色上，从场景原点指向右侧的位置矢量也同样是这样。一旦你对G缓冲中的内容满意了，我们就该进入到下一步：光照处理阶段了。

延迟光照处理阶段

现在我们已经有了一大堆的片段数据储存在G缓冲中供我们处置，我们可以选择通过一个像素一个像素地遍历各个G缓冲纹理，并将储存在它们里面的内容作为光照算法的输入，来完全计算场景最终的光照颜色。由于所有的G缓冲纹理都代表的是最终变换的片段值，我们只需要对每一个像素执行一次昂贵的光照运算就行了。这使得延迟光照非常高效，特别是在需要调用大量重型片段着色器的复杂场景中。

对于这个光照处理阶段，我们将会渲染一个2D全屏的方形(有一点像后期处理效果)并且在每个像素上运行一个昂贵的光片段着色器。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shaderLightingPass.Use();
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, gPosition);
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2);
 glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
// 同样发送光照相关的uniform
SendAllLightUniformsToShader(shaderLightingPass);
glUniform3fv(glGetUniformLocation(shaderLightingPass.Program, "viewPos"), 1, &camera.Position[0]);
RenderQuad();
```

我们在渲染之前绑定了G缓冲中所有相关的纹理，并且发送光照相关的uniform变量到着色器中。

光照处理阶段的片段着色器和我们之前一直在用的光照教程着色器是非常相似的，除了我们添加了一个新的方法，从而使我们能够获取光照的输入变量，当然这些变量我们会从G缓冲中直接采样。

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

struct Light {
    vec3 Position;
    vec3 Color;
};
const int NR_LIGHTS = 32;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;

void main()
{
    // 从G缓冲中获取数据
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Albedo = texture(gAlbedoSpec, TexCoords).rgb;
    float Specular = texture(gAlbedoSpec, TexCoords).a;

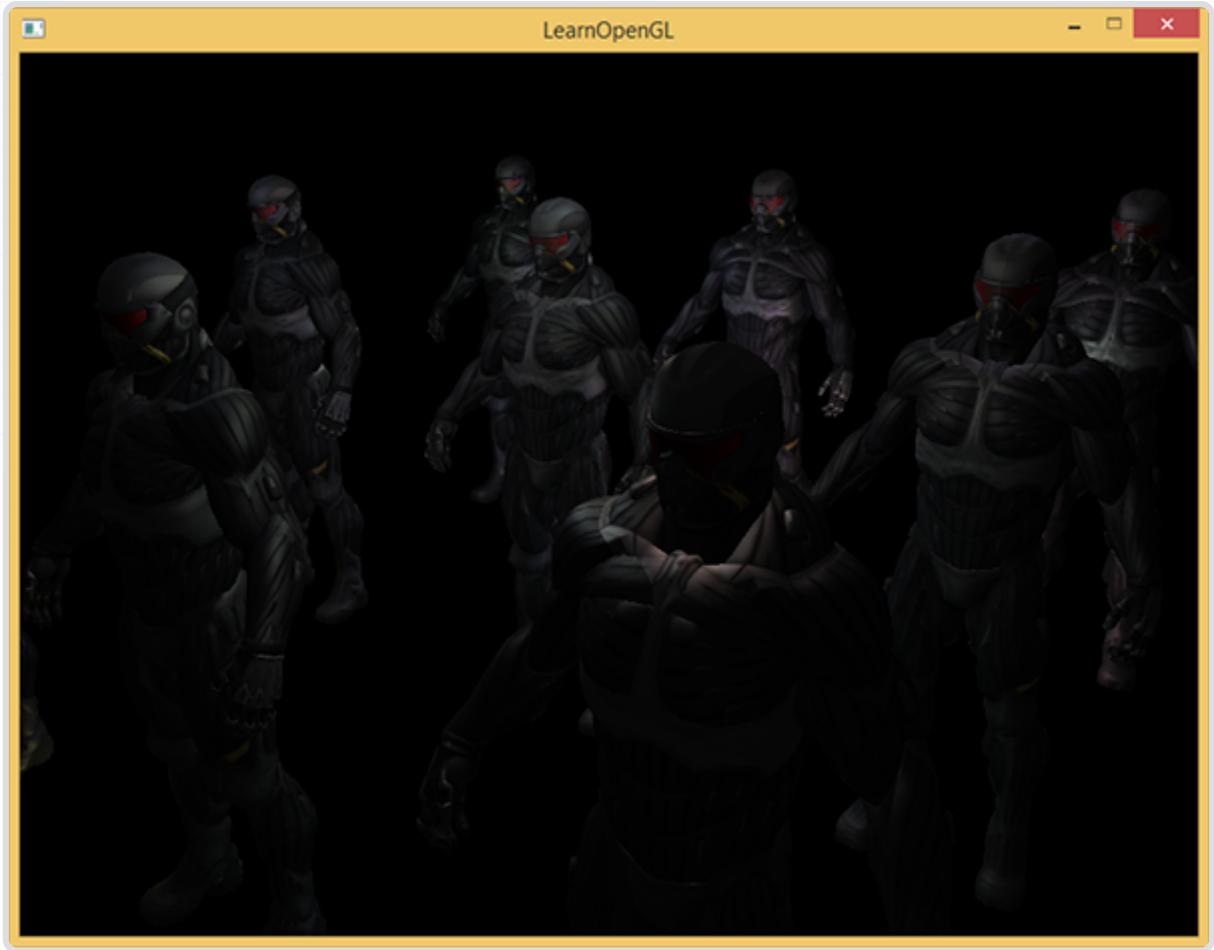
    // 然后和往常一样地计算光照
    vec3 lighting = Albedo * 0.1; // 硬编码环境光照分量
    vec3 viewDir = normalize(viewPos - FragPos);
    for(int i = 0; i < NR_LIGHTS; ++i)
    {
        // 漫反射
        vec3 lightDir = normalize(lights[i].Position - FragPos);
        vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Albedo * lights[i].Color;
        lighting += diffuse;
    }

    FragColor = vec4(lighting, 1.0);
}
```

光照处理阶段着色器接受三个uniform纹理，代表G缓冲，它们包含了我们在几何处理阶段储存的所有数据。如果我们现在再使用当前片段的纹理坐标采样这些数据，我们将会获得和之前完全一样的片段值，这就像我们在直接渲染几何体。在片段着色器的一开始，我们通过一个简单的纹理查找从G缓冲纹理中获取了光照相关的变量。注意我们从 `gAlbedoSpec` 纹理中同时获取了 `Albedo` 颜色和 `Specular` 强度。

因为我们现在已经有了必要的逐片段变量(和相关的uniform变量)来计算布林-冯氏光照(Blinn-Phong Lighting)，我们不需要对光照代码做任何修改了。我们在延迟着色法中唯一需要改的就是获取光照输入变量的方法。

运行一个包含32个小光源的简单Demo会是像这样子的：



你可以在以下位置找到Demo的[完整源代码](#)，和几何渲染阶段的[顶点](#)和[片段](#)着色器，还有光照渲染阶段的[顶点](#)和[片段](#)着色器。

延迟着色法的其中一个缺点就是它不能进行[混合](#)(Blending)，因为G缓冲中所有的数据都是从一个单独的片段中来的，而混合需要对多个片段的组合进行操作。延迟着色法另外一个缺点就是它迫使你对大部分场景的光照使用相同的光照算法，你可以通过包含更多关于材质的数据到G缓冲中来减轻这一缺点。

为了克服这些缺点(特别是混合)，我们通常分割我们的渲染器为两个部分：一个是延迟渲染的部分，另一个是专门为了混合或者其他不适合延迟渲染管线的着色器效果而设计的正向渲染的部分。为了展示这是如何工作的，我们将会使用正向渲染器渲染光源为一个小立方体，因为光照立方体会需要一个特殊的着色器(会输出一个光照颜色)。

结合延迟渲染与正向渲染

现在我们想要渲染每一个光源为一个3D立方体，并放置在光源的位置上随着延迟渲染器一起发出光源的颜色。很明显，我们需要做的第一件事就是在延迟渲染方形之上正向渲染所有的光源，它会在延迟渲染管线的最后进行。所以我们只需要像正常情况下渲染立方体，只是会在我完成延迟渲染操作之后进行。代码会像这样：

```
// 延迟渲染光照渲染阶段
[...]
RenderQuad();

// 现在像正常情况一样正向渲染所有光立方体
shaderLightBox.Use();
glUniformMatrix4fv(locProjection, 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(locView, 1, GL_FALSE, glm::value_ptr(view));
for (GLuint i = 0; i < lightPositions.size(); i++)
{
    model = glm::mat4();
    model = glm::translate(model, lightPositions[i]);
    model = glm::scale(model, glm::vec3(0.25f));
    glUniformMatrix4fv(locModel, 1, GL_FALSE, glm::value_ptr(model));
    glUniform3fv(locLightcolor, 1, &lightColors[i][0]);
    RenderCube();
}
```

然而，这些渲染出来的立方体并没有考虑到我们储存的延迟渲染器的几何深度(Depth)信息，并且结果是它被渲染在之前渲染过的物体之上，这并不是我们想要的结果。



我们需要做的就是首先复制出在几何渲染阶段中储存的深度信息，并输出到默认的帧缓冲的深度缓冲，然后我们才渲染光立方体。这样之后只有当它在之前渲染过的几何体上方的时候，光立方体的片段才会被渲染出来。我们可以使用 `glBlitFramebuffer` 复制一个帧缓冲的内容到另一个帧缓冲中，这个函数我们也在[抗锯齿](#)的教程中使用过，用来还原多重采样的帧缓冲。`glBlitFramebuffer` 这个函数允许我们复制一个用户定义的帧缓冲区域到另一个用户定义的帧缓冲区域。

我们储存所有延迟渲染阶段中所有物体的深度信息在 `gBuffer` 这个FBO中。如果我们仅仅是简单复制它的深度缓冲内容到默认帧缓冲的深度缓冲中，那么光立方体就会像是场景中所有的几何体都是正向渲染出来的一样渲染出来。就像在[抗锯齿](#)教程中介绍的那样，我们需要指定一个帧缓冲为读帧缓冲(Read Framebuffer)，并且类似地指定一个帧缓冲为写帧缓冲(Write Framebuffer)：

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, gBuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0); // 写入到默认帧缓冲
glBlitFramebuffer(
    0, 0, SCR_WIDTH, SCR_HEIGHT, 0, 0, SCR_WIDTH, SCR_HEIGHT, GL_DEPTH_BUFFER_BIT, GL_NEAREST
);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 现在像之前一样渲染光立方体
[...]
```

在这里我们复制整个读帧缓冲的深度缓冲信息到默认帧缓冲的深度缓冲，对于颜色缓冲和模板缓冲我们也可以这样处理。现在如果我们接下来再渲染光立方体，场景里的几何体将会看起来很真实了，而不只是简单地粘贴立方体到2D方形之上：



你可以在[这里](#)找到Demo的源代码，还有光立方体的[顶点](#)和[片段](#)着色器。

有了这种方法，我们就能够轻易地结合延迟着色法和正向着色法了。这真是太棒了，我们现在可以应用混合或者渲染需要特殊着色器效果的物体了，这在延迟渲染中是不可能做到的。

更多的光源

延迟渲染一直被称赞的原因就是它能够渲染大量的光源而不消耗大量的性能。然而，延迟渲染本身并不能支持非常大量的光源，因为我们仍然必须要对场景中每一个光源计算每一个片段的光照分量。真正让大量光源成为可能的是我们能够对延迟渲染管线引用的一个非常棒的优化：光体积(**Light Volumes**)

通常情况下，当我们渲染一个复杂光照场景下的片段着色器时，我们会计算场景中每一个光源的贡献，不管它们离这个片段有多远。很大部分的光源根本就不会到达这个片段，所以为什么我们还要浪费这么多光照运算呢？

隐藏在光体积背后的想法就是计算光源的半径，或是体积，也就是光能够到达片段的范围。由于大部分光源都使用了某种形式的衰减(Attenuation)，我们可以用它来计算光源能够到达的最大路程，或者说是半径。我们接下来只需要对那些在一个或多个光体积内的片段进行繁重的光照运算就行了。这可以给我们省下来很可观的计算量，因为我们现在只在需要的情况下计算光照。

这个方法的难点基本就是找出一个光源光体积的大小，或者是半径。

计算一个光源的体积或半径

为了获取一个光源的体积半径，我们需要解一个对于一个我们认为是黑暗(Dark)的亮度(Brightness)的衰减方程，它可以是0.0，或者是更亮一点的但仍被认为黑暗的值，像是0.03。为了展示我们如何计算光源的体积半径，我们将会使用一个在[投光物](#)这节中引入的一个更加复杂，但非常灵活的衰减方程：

我们现在想要在等于0的前提下解这个方程，也就是说光在该距离完全是黑暗的。然而这个方程永远不会真正等于0.0，所以它没有解。所以，我们不会求表达式等于0.0时候的解，相反我们会求当亮度值靠近于0.0的解，这时候它还是能被看做是黑暗的。在这个教程的演示场景中，我们选择作为一个合适的光照值；除以256是因为默认的8-bit帧缓冲可以每个分量显示这么多强度值(Intensity)。

我们使用的衰减方程在它的可视范围内基本都是黑暗的，所以如果我们想要限制它为一个比更加黑暗的亮度，光体积就会变得太大从而变得低效。只要是用户不能在光体积边缘看到一个突兀的截断，这个参数就没事了。当然它还是依赖于场景的类型，一个高的亮度阀值会产生更小的光体积，从而获得更高的效率，然而它同样会产生一个很容易发现的副作用，那就是光会在光体积边界看起来突然断掉。

我们要求的衰减方程会是这样：

在这里，是光源最亮的颜色分量。我们之所以使用光源最亮的颜色分量是因为解光源最亮的强度值方程最好地反映了理想光体积半径。

从这里我们继续解方程：

最后的方程形成了的形式，我们可以用求根公式来解这个二次方程：

它给我们了一个通用公式从而允许我们计算的值，即光源的光体积半径，只要我们提供了一个常量，线性和二次项参数：

```
GLfloat constant = 1.0;
GLfloat linear    = 0.7;
GLfloat quadratic = 1.8;
GLfloat lightMax  = std::fmaxf(std::fmaxf(lightColor.r, lightColor.g), lightColor.b);
GLfloat radius    =
(-linear + std::sqrtf(linear * linear - 4 * quadratic * (constant - (256.0 / 5.0) * lightMax)))
 / (2 * quadratic);
```

它会返回一个大概在1.0到5.0范围内的半径值，它取决于光的最大强度。

对于场景中每一个光源，我们都计算它的半径，并仅在片段在光源的体积内部时才计算该光源的光照。下面是更新过的光照处理阶段片段着色器，它考虑到了计算出来的光体积。注意这种方法仅仅用作教学目的，在实际场景中是不可行的，我们会在后面讨论它：

```
struct Light {
    [...]
    float Radius;
};

void main()
{
    [...]
    for(int i = 0; i < NR_LIGHTS; ++i)
    {
        // 计算光源和该片段间距离
        float distance = length(lights[i].Position - FragPos);
        if(distance < lights[i].Radius)
        {
            // 执行大开销光照
            [...]
        }
    }
}
```

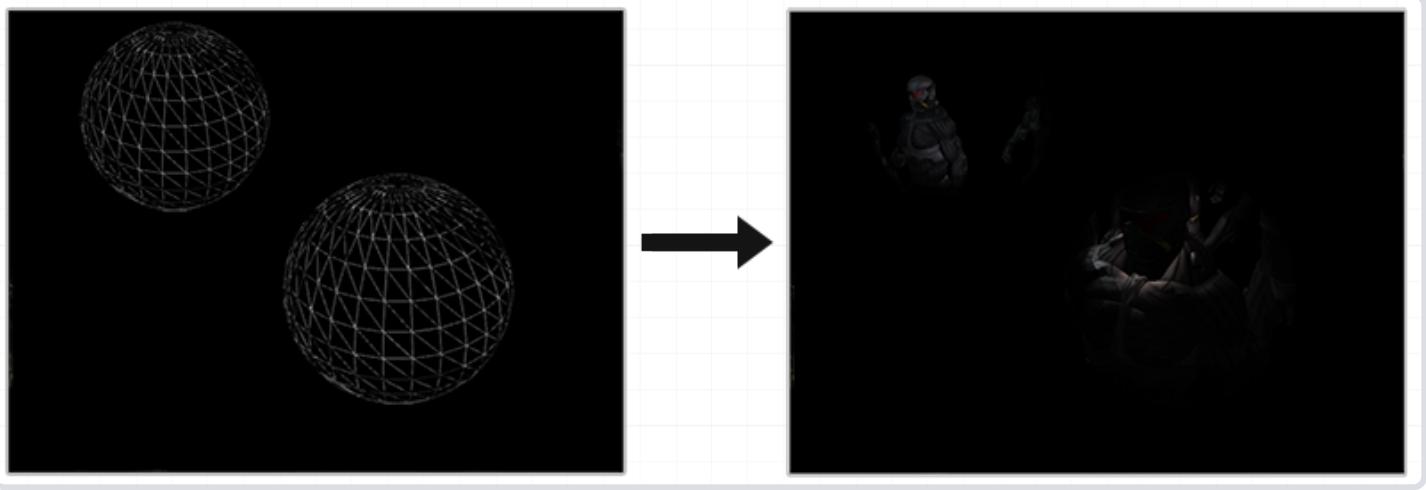
这次的结果和之前一模一样，但是这次物体只对所在光体积的光源计算光照。

你可以在[这里](#)找到Demo最终的源码，并且还有更新的光照渲染阶段的[片段着色器](#)

真正使用光体积

上面那个片段着色器在实际情况下不能真正地工作，并且它只演示了我们可以不知怎样能使用光体积减少光照运算。然而事实上，你的GPU和GLSL并不擅长优化循环和分支。这一缺陷的原因是GPU中着色器的运行是高度并行的，大部分的架构要求对于一个大的线程集合，GPU需要对它运行完全一样的着色器代码从而获得高效率。这通常意味着一个着色器运行时总是执行一个if语句所有的分支从而保证着色器运行都是一样的，这使得我们之前的半径检测优化完全变得无用，我们仍然在对所有光源计算光照！

使用光体积更好的方法是渲染一个实际的球体，并根据光体积的半径缩放。这些球的中心放置在光源的位置，由于它是根据光体积半径缩放的，这个球体正好覆盖了光的可视体积。这就是我们的技巧：我们使用大体相同的延迟片段着色器来渲染球体。因为球体产生了完全匹配于受影响像素的着色器调用，我们只渲染了受影响的像素而跳过其它的像素。下面这幅图展示了这一技巧：



它被应用在场景中每个光源上，并且所得的片段相加混合在一起。这个结果和之前场景是一样的，但这一次只渲染对于光源相关的片段。它有效地减少了从 $nr_objects * nr_lights$ 到 $nr_objects + nr_lights$ 的计算量，这使得多光源场景的渲染变得无比高效。这正是为什么延迟渲染非常适合渲染很大数量光源。

然而这个方法仍然有一个问题：面剔除(Face Culling)需要被启用(否则我们会渲染一个光效果两次)，并且在它启用的时候用户可能进入一个光源的光体积，然而这样之后这个体积就不再被渲染了(由于背面剔除)，这会使得光源的影响消失。这个问题可以通过一个模板缓冲技巧来解决。

渲染光体积确实会带来沉重的性能负担，虽然它通常比普通的延迟渲染更快，这仍然不是最好的优化。另外两个基于延迟渲染的更流行(并且更高效)的拓展叫做延迟光照(Deferred Lighting)和切片式延迟着色(Tile-based Deferred Shading)。这些方法会很大程度上提高大量光源渲染的效率，并且也能允许一个相对高效的多重采样抗锯齿(MSAA)。然而受制于这篇教程的长度，我将会在之后的教程中介绍这些优化。

延迟渲染 vs 正向渲染

仅仅是延迟着色法它本身(没有光体积)已经是一个很大的优化了，每个像素仅仅运行一个单独的片段着色器，然而对于正向渲染，我们通常会对一个像素运行多次片段着色器。当然，延迟渲染确实带来一些缺点：大内存开销，没有MSAA和混合(仍需要正向渲染的配合)。

当你有一个很小的场景并且没有很多的光源时候，延迟渲染并不一定会更快一点，甚至有些时候由于开销超过了它的优点还会更慢。然而在一个更复杂的场景中，延迟渲染会快速变成一个重要的优化，特别是有了更先进的优化拓展的时候。

最后我仍然想指出，基本上所有能通过正向渲染完成的效果能够同样在延迟渲染场景中实现，这通常需要一些小的翻译步骤。举个例子，如果我们想要在延迟渲染器中使用法线贴图(Normal Mapping)，我们需要改变几何渲染阶段着色器来输出一个世界空间法线(World-space Normal)，它从法线贴图中提取出来(使用一个TBN矩阵)而不是表面法线，光照渲染阶段中的光照运算一点都不需要变。如果你想要让视差贴图工作，首先你需要在采样一个物体的漫反射，镜面，和法线纹理之前首先置换几何渲染阶段中的纹理坐标。一旦你了解了延迟渲染背后的理念，变得有创造力并不是什么难事。

附加资源

- [Tutorial 35: Deferred Shading - Part 1](#) : OGLDev的一个分成三部分的延迟着色法教程。在Part 2和3中介绍了渲染光体积
- [Deferred Rendering for Current and Future Rendering Pipelines](#) : Andrew Lauritzen的幻灯片，讨论了高级切片式延迟着色法和延迟光照

2.6.9 SSAO

原文 [SSAO](#)

作者 JoeyDeVries

翻译 Krasjet

校对 未校对

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

我们已经在前面的基础教程中简单介绍到了这部分内容：环境光照(Ambient Lighting)。环境光照是我们加入场景总体光照中的一个固定光照常量，它被用来模拟光的散射(Scattering)。在现实中，光线会以任意方向散射，它的强度是会一直改变的，所以间接被照到的那部分场景也应该有变化的强度，而不是一成不变的环境光。其中一种间接光照的模拟叫做环境光遮蔽(Ambient Occlusion)，它的原理是通过将褶皱、孔洞和非常靠近的墙面变暗的方法近似模拟出间接光照。这些区域很大程度上是被周围的几何体遮蔽的，光线会很难流失，所以这些地方看起来会更暗一些。站起来看一看你房间的拐角或者是褶皱，是不是这些地方会看起来有一点暗？

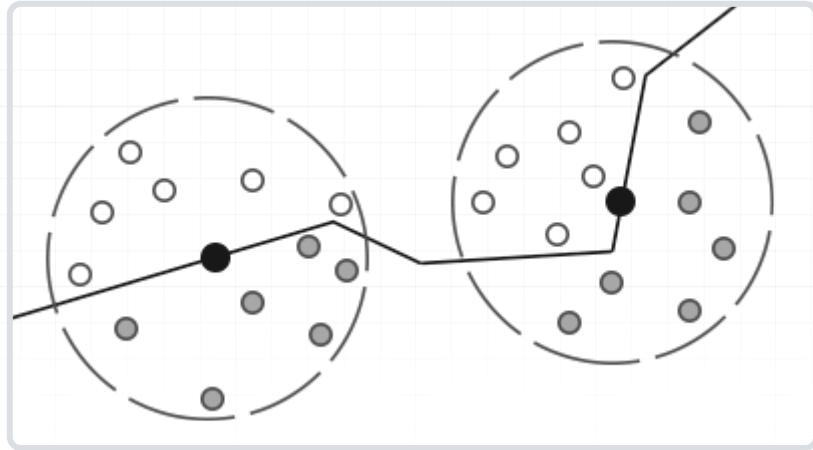
下面这幅图展示了在使用和不使用SSAO时场景的不同。特别注意对比褶皱部分，你会发现(环境)光被遮蔽了许多：



尽管这不是一个非常明显的效果，启用SSAO的图像确实给我们更真实的感觉，这些小的遮蔽细节给整个场景带来了更强的深度感。

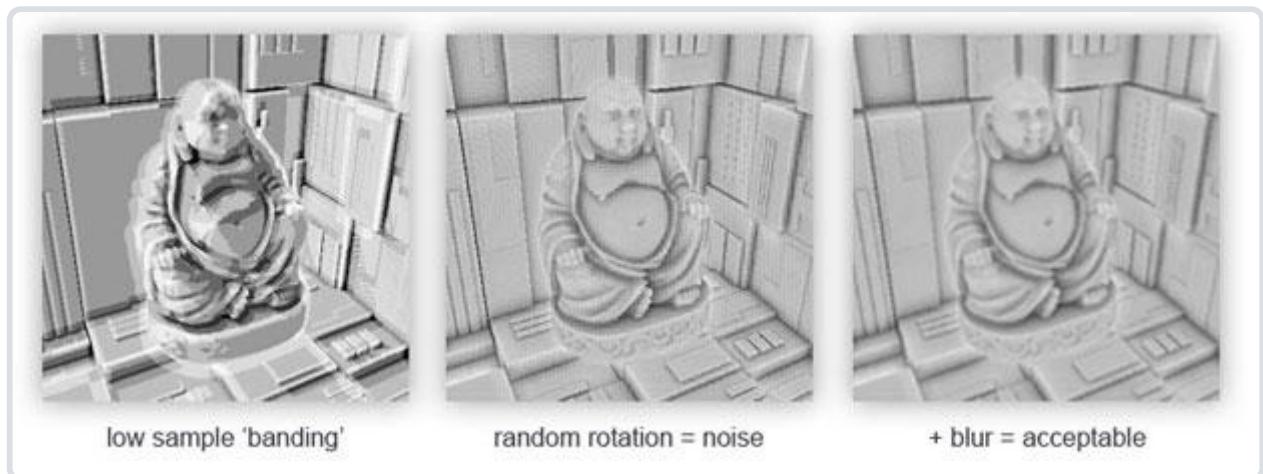
环境光遮蔽这一技术会带来很大的性能开销，因为它还需要考虑周围的几何体。我们可以对空间中每一点发射大量光线来确定其遮蔽量，但是这在实时运算中会很快变成大问题。在2007年，Crytek公司发布了一款叫做屏幕空间环境光遮蔽(Screen-Space Ambient Occlusion, SSAO)的技术，并用在了他们的看家作孤岛危机上。这一技术使用了屏幕空间场景的深度而不是真实的几何体数据来确定遮蔽量。这一做法相对于真正的环境光遮蔽不但速度快，而且还能获得很好的效果，使得它成为近似实时环境光遮蔽的标准。

SSAO背后的原理很简单：对于铺屏四边形(Screen-filled Quad)上的每一个片段，我们都会根据周边深度值计算一个遮蔽因子(Occlusion Factor)。这个遮蔽因子之后会被用来减少或者抵消片段的环境光照分量。遮蔽因子是通过采集片段周围球型核心(Kernel)的多个深度样本，并和当前片段深度值对比而得到的。高于片段深度值样本的个数就是我们想要的遮蔽因子。



上图中在几何体内灰色的深度样本都是高于片段深度值的，他们会增加遮蔽因子；几何体内样本个数越多，片段获得的环境光照也就越少。

很明显，渲染效果的质量和精度与我们采样的样本数量有直接关系。如果样本数量太低，渲染的精度会急剧减少，我们会得到一种叫做波纹(Banding)的效果；如果它太高了，反而会影响性能。我们可以通过引入随机性到采样核心(Sample Kernel)的采样中从而减少样本的数目。通过随机旋转采样核心，我们能在有限样本数量中得到高质量的结果。然而这仍然会有一定的麻烦，因为随机性引入了一个很明显的噪声图案，我们将需要通过模糊结果来修复这一问题。下面这幅图片([John Chapman](#)的佛像)展示了波纹效果还有随机性造成的效果：

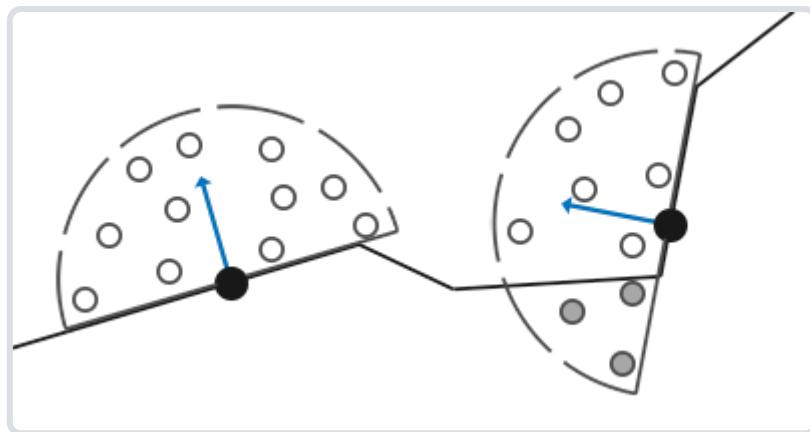


你可以看到，尽管我们在低样本数的情况下得到了很明显的波纹效果，引入随机性之后这些波纹效果就完全消失了。

Crytek公司开发的SSAO技术会产生一种特殊的视觉风格。因为使用的采样核心是一个球体，它导致平整的墙面也会显得灰蒙蒙的，因为核心中一半的样本都会在墙这个几何体上。下面这幅图展示了孤岛危机的SSAO，它清晰地展示了这种灰蒙蒙的感觉：



由于这个原因，我们将不会使用球体的采样核心，而使用一个沿着表面法向量的半球体采样核心。



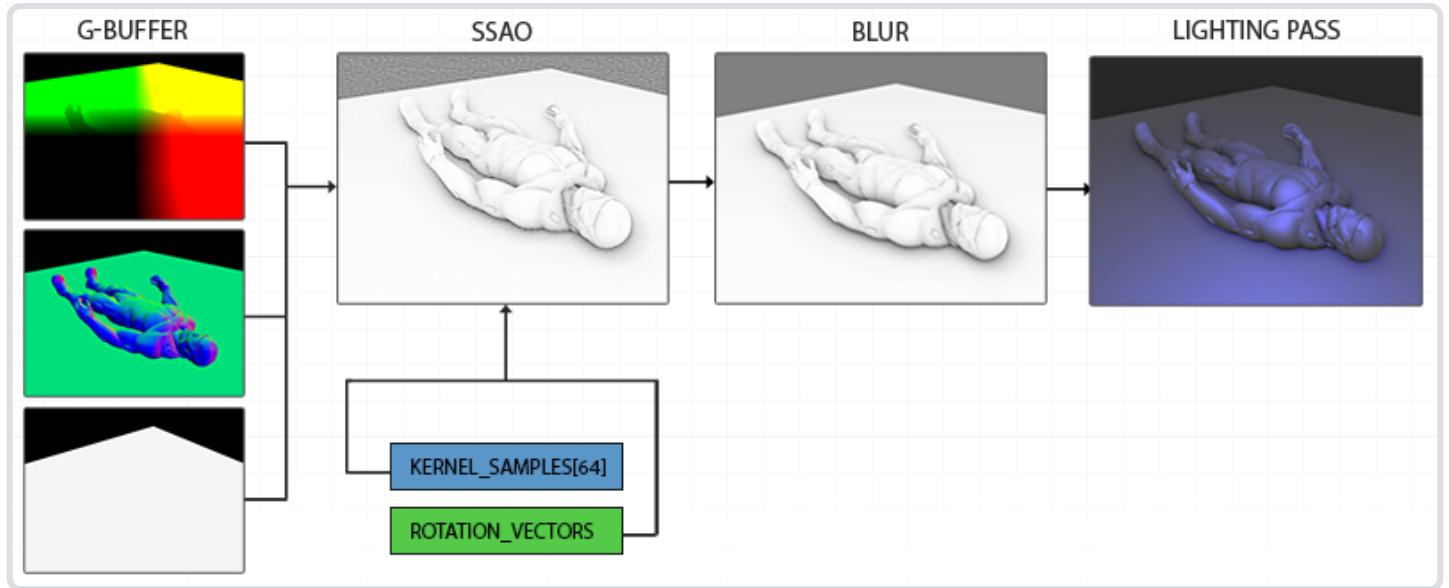
通过在法向半球体(Normal-oriented Hemisphere)周围采样，我们将不会考虑到片段底部的几何体。它消除了环境光遮蔽灰蒙蒙的感觉，从而产生更真实的结果。这个SSAO教程将会基于法向半球法和John Chapman出色的[SSAO教程](#)。

样本缓冲

SSAO需要获取几何体的信息，因为我们需要一些方式来确定一个片段的遮蔽因子。对于每一个片段，我们将需要这些数据：

- 逐片段位置向量
- 逐片段的法线向量
- 逐片段的反射颜色
- 采样核心
- 用来旋转采样核心的随机旋转矢量

通过使用一个逐片段观察空间位置，我们可以将一个采样半球核心对准片段的观察空间表面法线。对于每一个核心样本我们会采样线性深度纹理来比较结果。采样核心会根据旋转矢量稍微偏转一点；我们所获得的遮蔽因子将会之后用来限制最终的环境光照分量。



由于SSAO是一种屏幕空间技巧，我们对铺屏2D四边形上每一个片段计算这一效果；也就是说我们没有场景中几何体的信息。我们能做的只是渲染几何体数据到屏幕空间纹理中，我们之后再会将此数据发送到SSAO着色器中，之后我们就能访问到这些几何体数据了。如果你看了前面一篇教程，你会发现这和延迟渲染很相似。这也就是说SSAO和延迟渲染能完美地兼容，因为我们已经存位置和法线向量到G缓冲中了。

在这个教程中，我们将会在一个简化版本的延迟渲染器([延迟着色法](#)教程中的基础上实现SSAO，所以如果你不知道什么是延迟着色法，请先读完那篇教程。

由于我们已经有了逐片段位置和法线数据(G缓冲中)，我们只需要更新一下几何着色器，让它包含片段的线性深度就行了。回忆我们在深度测试那一节学过的知识，我们可以从 `gl_FragCoord.z` 中提取线性深度：

```
#version 330 core
layout (location = 0) out vec4 gPositionDepth;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;

const float NEAR = 0.1; // 投影矩阵的近平面
const float FAR = 50.0f; // 投影矩阵的远平面
float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // 回到NDC
    return (2.0 * NEAR * FAR) / (FAR + NEAR - z * (FAR - NEAR));
}

void main()
{
    // 储存片段的位置矢量到第一个G缓冲纹理
    gPositionDepth.xyz = FragPos;
    // 储存线性深度到gPositionDepth的alpha分量
    gPositionDepth.a = LinearizeDepth(gl_FragCoord.z);
    // 储存法线信息到G缓冲
    gNormal = normalize(Normal);
    // 和漫反射颜色
    gAlbedoSpec.rgb = vec3(0.95);
}
```

提取出来的线性深度是在观察空间中的，所以之后的运算也是在观察空间中。确保G缓冲中的位置和法线都在观察空间中(乘上观察矩阵也一样)。观察空间线性深度值之后会被保存在 `gPositionDepth` 颜色缓冲的alpha分量中，省得我们再声明一个新的颜色缓冲纹理。

通过一些小技巧来通过深度值重构实际位置值是可能的，Matt Pettineo在他的[博客](#)里提到了这一技巧。这一技巧需要在着色器里进行一些计算，但是省了我们在G缓冲中存储位置数据，从而省了很多内存。为了示例的简单，我们将不会使用这些优化技巧，你可以自行探究。

`gPositionDepth` 颜色缓冲纹理被设置成了下面这样：

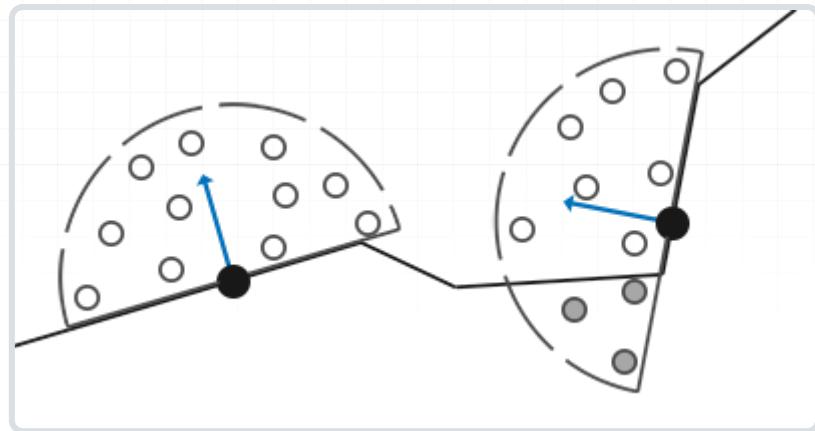
```
 glGenTextures(1, &gPositionDepth);
 glBindTexture(GL_TEXTURE_2D, gPositionDepth);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

这给我们了一个线性深度纹理，我们可以用它来对每一个核心样本获取深度值。注意我们把线性深度值存储为了浮点数据；这样从0.1到50.0范围深度值都不会被限制在[0.0, 1.0]之间了。如果你不用浮点值存储这些深度数据，确保你首先将值除以 `FAR` 来标准化它们，再存储到 `gPositionDepth` 纹理中，并在以后的着色器中用相似的方法重建它们。同样需要注意的是 `GL_CLAMP_TO_EDGE` 的纹理封装方法。这保证了我们不会不小心采样到在屏幕空间中纹理默认坐标区域之外的深度值。

接下来我们需要真正的半球采样核心和一些方法来随机旋转它。

法向半球

我们需要沿着表面法线方向生成大量的样本。就像我们在这个教程的开始介绍的那样，我们想要生成形成半球形的样本。由于对每个表面法线方向生成采样核心非常困难，也不合实际，我们将在[切线空间](#)(Tangent Space)内生成采样核心，法向量将指向正z方向。



假设我们有一个单位半球，我们可以获得一个拥有最大64样本值的采样核心：

```
std::uniform_real_distribution<GLfloat> randomFloats(0.0, 1.0); // 随机浮点数，范围0.0 - 1.0
std::default_random_engine generator;
std::vector<glm::vec3> ssaoKernel;
for (GLuint i = 0; i < 64; ++i)
{
    glm::vec3 sample(
        randomFloats(generator) * 2.0 - 1.0,
        randomFloats(generator) * 2.0 - 1.0,
        randomFloats(generator));
    sample = glm::normalize(sample);
    sample *= randomFloats(generator);
    GLfloat scale = GLfloat(i) / 64.0;
    ssaoKernel.push_back(sample);
}
```

我们在切线空间中以-1.0到1.0为范围变换x和y方向，并以0.0和1.0为范围变换样本的z方向(如果以-1.0到1.0为范围，取样核心就变成球型了)。由于采样核心将会沿着表面法线对齐，所得的样本矢量将会在半球里。

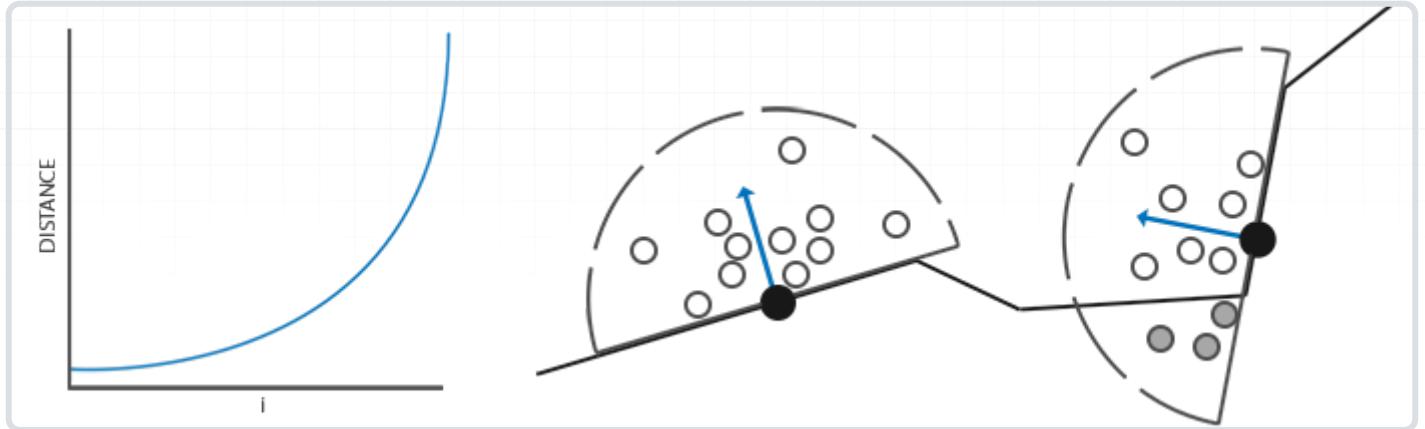
目前，所有的样本都是平均分布在采样核心里的，但是我们更愿意将更多的注意放在靠近真正片段的遮蔽上，也就是将核心样本靠近原点分布。我们可以用一个加速插值函数实现它：

```
... [接上函数]
scale = lerp(0.1f, 1.0f, scale * scale);
sample *= scale;
ssaoKernel.push_back(sample);
}
```

`lerp` 被定义为：

```
GLfloat lerp(GLfloat a, GLfloat b, GLfloat f)
{
    return a + f * (b - a);
}
```

这就给了我们一个大部分样本靠近原点的核心分布。



每个核心样本将会被用来偏移观察空间片段位置从而采样周围的几何体。我们在教程开始的时候看到，如果没有变化采样核心，我们将需要大量的样本来获得真实的结果。通过引入一个随机的转动到采样核心中，我们可以很大程度上减少这一数量。

随机核心转动

通过引入一些随机性到采样核心上，我们可以大大减少获得不错结果所需的样本数量。我们可以对场景中每一个片段创建一个随机旋转向量，但这会很快将内存耗尽。所以，更好的方法是创建一个小的随机旋转向量纹理平铺在屏幕上。

我们创建一个 4×4 朝向切线空间平面法线的随机旋转向量数组：

```
std::vector<glm::vec3> ssaoNoise;
for (GLuint i = 0; i < 16; i++)
{
    glm::vec3 noise(
        randomFloats(generator) * 2.0 - 1.0,
        randomFloats(generator) * 2.0 - 1.0,
        0.0f);
    ssaoNoise.push_back(noise);
}
```

由于采样核心是沿着正z方向在切线空间内旋转，我们设定z分量为0.0，从而围绕z轴旋转。

我们接下来创建一个包含随机旋转向量的4x4纹理；记得设定它的封装方法为 `GL_REPEAT`，从而保证它合适地平铺在屏幕上。

```
GLuint noiseTexture;
glGenTextures(1, &noiseTexture);
 glBindTexture(GL_TEXTURE_2D, noiseTexture);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, 4, 4, 0, GL_RGB, GL_FLOAT, &ssaoNoise[0]);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

现在我们有了所有的相关输入数据，接下来我们需要实现SSAO。

SSAO着色器

SSAO着色器在2D的铺屏四边形上运行，它对于每一个生成的片段计算遮蔽值(为了在最终的光照着色器中使用)。由于我们需要存储SSAO阶段的结果，我们还需要在创建一个帧缓冲对象：

```
GLuint ssaoFBO;
 glGenFramebuffers(1, &ssaoFBO);
 glBindFramebuffer(GL_FRAMEBUFFER, ssaoFBO);
 GLuint ssaoColorBuffer;

 glGenTextures(1, &ssaoColorBuffer);
 glBindTexture(GL_TEXTURE_2D, ssaoColorBuffer);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, ssaoColorBuffer, 0);
```

由于环境遮蔽的结果是一个灰度值，我们将只需要纹理的红色分量，所以我们将颜色缓冲的内部格式设置为 `GL_RED`。

渲染SSAO完整的过程会像这样：

```
// 几何处理阶段：渲染到G缓冲中
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
[...]
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 使用G缓冲渲染SSAO纹理
glBindFramebuffer(GL_FRAMEBUFFER, ssaoFBO);
    glClear(GL_COLOR_BUFFER_BIT);
    shaderSSAO.Use();
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, gPositionDepth);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, gNormal);
    glActiveTexture(GL_TEXTURE2);
    glBindTexture(GL_TEXTURE_2D, noiseTexture);
    SendKernelSamplesToShader();
    glUniformMatrix4fv(projLocation, 1, GL_FALSE, glm::value_ptr(projection));
    RenderQuad();
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 光照处理阶段：渲染场景光照
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shaderLightingPass.Use();
[...]
glActiveTexture(GL_TEXTURE3);
glBindTexture(GL_TEXTURE_2D, ssaoColorBuffer);
[...]
RenderQuad();
```

`shaderSSAO` 这个着色器将对应G缓冲纹理(包括线性深度), 噪声纹理和法向半球核心样本作为输入参数:

```
#version 330 core
out float FragColor;
in vec2 TexCoords;

uniform sampler2D gPositionDepth;
uniform sampler2D gNormal;
uniform sampler2D texNoise;

uniform vec3 samples[64];
uniform mat4 projection;

// 屏幕的平铺噪声纹理会根据屏幕分辨率除以噪声大小的值来决定
const vec2 noiseScale = vec2(800.0/4.0, 600.0/4.0); // 屏幕 = 800x600

void main()
{
    [...]
}
```

注意我们这里有一个 `noiseScale` 的变量。我们想要将噪声纹理平铺(Tile)在屏幕上，但是由于 `TexCoords` 的取值在0.0和1.0之间，`texNoise` 纹理将不会平铺。所以我们将通过屏幕分辨率除以噪声纹理大小的方式计算 `TexCoords` 的缩放大小，并在之后提取相关输入向量的时候使用。

```
vec3 fragPos = texture(gPositionDepth, TexCoords).xyz;
vec3 normal = texture(gNormal, TexCoords).rgb;
vec3 randomVec = texture(texNoise, TexCoords * noiseScale).xyz;
```

由于我们将 `texNoise` 的平铺参数设置为 `GL_REPEAT`，随机的值将会在全屏不断重复。加上 `fragPos` 和 `normal` 向量，我们就有足够的数据来创建一个TBN矩阵，将向量从切线空间变换到观察空间。

```
vec3 tangent = normalize(randomVec - normal * dot(randomVec, normal));
vec3 bitangent = cross(normal, tangent);
mat3 TBN = mat3(tangent, bitangent, normal);
```

通过使用一个叫做Gramm-Schmidt处理(Gramm-Schmidt Process)的过程，我们创建了一个正交基(Orthogonal Basis)，每一次它都会根据 `randomVec` 的值稍微倾斜。注意因为我们使用了一个随机向量来构造切线向量，我们没必要有一个恰好沿着几何体表面的TBN矩阵，也就是不需要逐顶点切线(和双切)向量。

接下来我们对每个核心样本进行迭代，将样本从切线空间变换到观察空间，将它们加到当前像素位置上，并将片段位置深度与储存在原始深度缓冲中的样本深度进行比较。我们来一步步讨论它：

```
float occlusion = 0.0;
for(int i = 0; i < kernelSize; ++i)
{
    // 获取样本位置
    vec3 sample = TBN * samples[i]; // 切线->观察空间
    sample = fragPos + sample * radius;

    [...]
}
```

这里的 `kernelSize` 和 `radius` 变量都可以用来调整效果；在这里我们分别保持他们的默认值为 `64` 和 `1.0`。对于每一次迭代我们首先变换各自样本到观察空间。之后我们会加观察空间核心偏移样本到观察空间片段位置上；最后再用 `radius` 乘上偏移样本来增加(或减少)SSAO的有效取样半径。

接下来我们变换 `sample` 到屏幕空间，从而我们可以就像正在直接渲染它的位置到屏幕上一样取样 `sample` 的(线性)深度值。由于这个向量目前在观察空间，我们将首先使用 `projection` 矩阵uniform变换它到裁剪空间。

```
vec4 offset = vec4(sample, 1.0);
offset = projection * offset; // 观察->裁剪空间
offset.xyz /= offset.w; // 透视线划分
offset.xyz = offset.xyz * 0.5 + 0.5; // 变换到0.0 - 1.0的值域
```

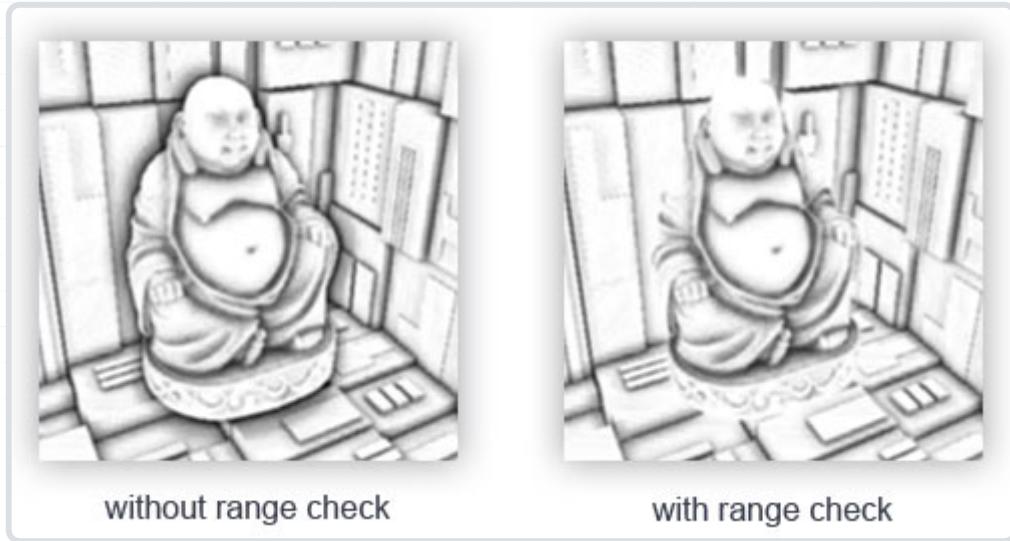
在变量被变换到裁剪空间之后，我们用 `xyz` 分量除以 `w` 分量进行透视线划分。结果所得的标准化设备坐标之后变换到[0.0, 1.0]范围以便我们使用它们去取样深度纹理：

```
float sampleDepth = -texture(gPositionDepth, offset.xy).w;
```

我们使用 `offset` 向量的 `x` 和 `y` 分量采样线性深度纹理从而获取样本位置从观察者视角的深度值(第一个不被遮蔽的可见片段)。我们接下来检查样本的当前深度值是否大于存储的深度值，如果是的，添加到最终的贡献因子上。

```
occlusion += (sampleDepth >= sample.z ? 1.0 : 0.0);
```

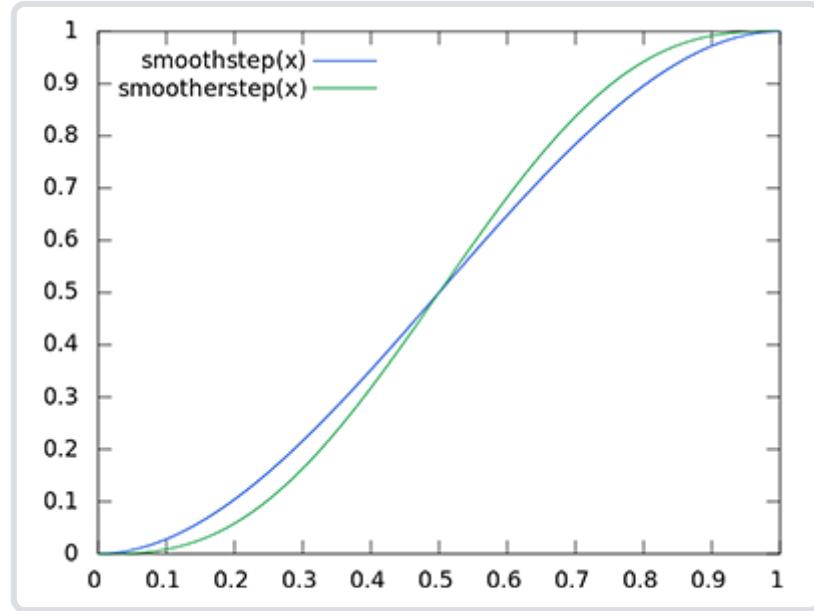
这并没有完全结束，因为仍然还有一个小问题需要考虑。当检测一个靠近表面边缘的片段时，它将会考虑测试表面之下的表面的深度值；这些值将会(不正确地)影响遮蔽因子。我们可以通过引入一个范围检测从而解决这个问题，正如下图所示([John Chapman](#)的佛像)：



我们引入一个范围测试从而保证我们只当被测深度值在取样半径内时影响遮蔽因子。将代码最后一行换成：

```
float rangeCheck = smoothstep(0.0, 1.0, radius / abs(fragPos.z - sampleDepth));
occlusion += (sampleDepth >= sample.z ? 1.0 : 0.0) * rangeCheck;
```

这里我们使用了GLSL的 `smoothstep` 函数，它非常光滑地在第一和第二个参数范围内插值了第三个参数。如果深度差因此最终取值在 `radius` 之间，它们的值将会光滑地根据下面这个曲线插值在0.0和1.0之间：



如果我们使用一个在深度值在 `radius` 之外就突然移除遮蔽贡献的硬界限范围检测(Hard Cut-off Range Check)，我们将会在范围检测应用的地方看见一个明显的(很难看的)边缘。

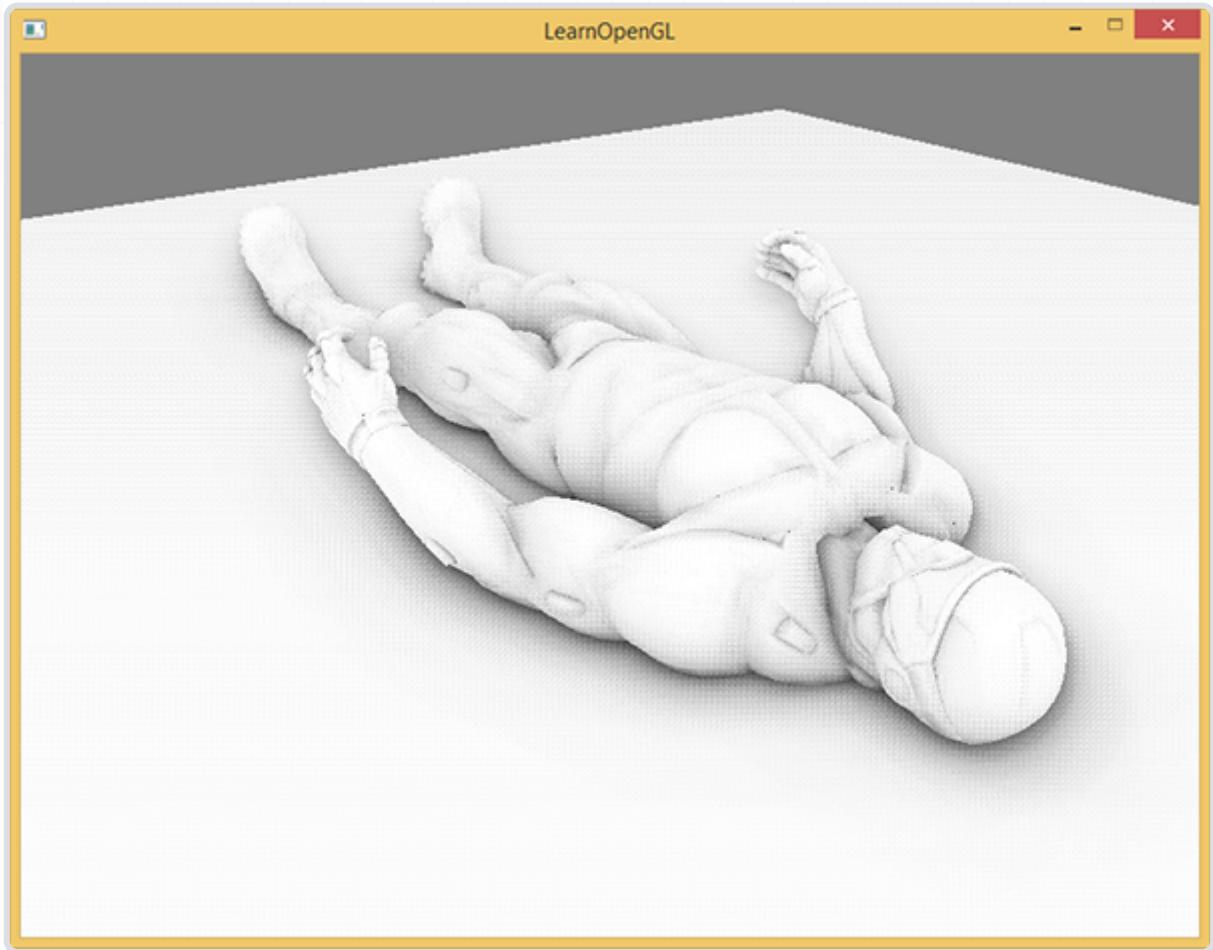
最后一步，我们需要将遮蔽贡献根据核心的大小标准化，并输出结果。注意我们用1.0减去了遮蔽因子，以便直接使用遮蔽因子去缩放环境光照分量。

```
}
```

```
occlusion = 1.0 - (occlusion / kernelSize);
```

```
FragColor = occlusion;
```

下面这幅图展示了我们最喜欢的纳米装模型正在打盹的场景，环境遮蔽着色器产生了以下的纹理：



可见，环境遮蔽产生了非常强烈的深度感。仅仅通过环境遮蔽纹理我们就已经能清晰地看见模型一定躺在地板上而不是浮在空中。

现在的效果仍然看起来不是很完美，由于重复的噪声纹理再图中清晰可见。为了创建一个光滑的环境遮蔽结果，我们需要模糊环境遮蔽纹理。

环境遮蔽模糊

在SSAO阶段和光照阶段之间，我们想要进行模糊SSAO纹理的处理，所以我们又创建了一个帧缓冲对象来储存模糊结果。

```
GLuint ssaoBlurFB0, ssaoColorBufferBlur;
 glGenFramebuffers(1, &ssaoBlurFB0);
 glBindFramebuffer(GL_FRAMEBUFFER, ssaoBlurFB0);
 glGenTextures(1, &ssaoColorBufferBlur);
 glBindTexture(GL_TEXTURE_2D, ssaoColorBufferBlur);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, ssaoColorBufferBlur, 0);
```

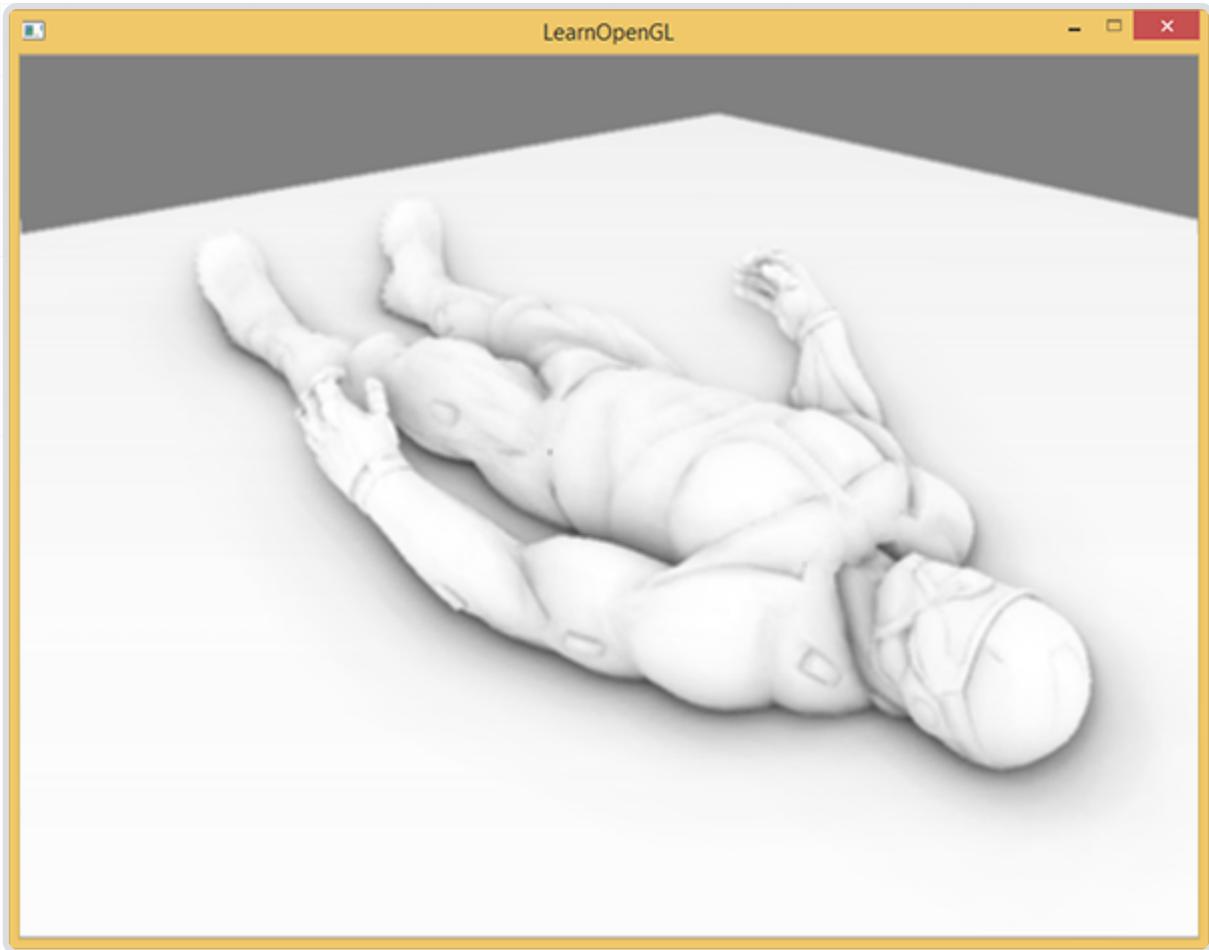
由于平铺的随机向量纹理保持了一致的随机性，我们可以使用这一性质来创建一个简单的模糊着色器：

```
#version 330 core
in vec2 TexCoords;
out float fragColor;

uniform sampler2D ssaoInput;

void main() {
    vec2 texelSize = 1.0 / vec2(textureSize(ssaoInput, 0));
    float result = 0.0;
    for (int x = -2; x < 2; ++x)
    {
        for (int y = -2; y < 2; ++y)
        {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(ssaoInput, TexCoords + offset).r;
        }
    }
    fragColor = result / (4.0 * 4.0);
}
```

这里我们遍历了周围在-2.0和2.0之间的SSAO纹理单元(Texel)，采样与噪声纹理维度相同数量的SSAO纹理。我们通过使用返回 `vec2` 纹理维度的 `textureSize`，根据纹理单元的真实大小偏移了每一个纹理坐标。我们平均所得的结果，获得一个简单但是有效的模糊效果：



这就完成了，一个包含逐片段环境遮蔽数据的纹理；在光照处理阶段中可以直接使用。

应用环境遮蔽

应用遮蔽因子到光照方程中极其简单：我们要做的只是将逐片段环境遮蔽因子乘到光照环境分量上。如果我们使用上个教程中的Blinn-Phong延迟光照着色器并做出一点修改，我们将会得到下面这个片段着色器：

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D gPositionDepth;
uniform sampler2D gNormal;
uniform sampler2D gAlbedo;
uniform sampler2D ssao;

struct Light {
    vec3 Position;
    vec3 Color;

    float Linear;
    float Quadratic;
    float Radius;
};

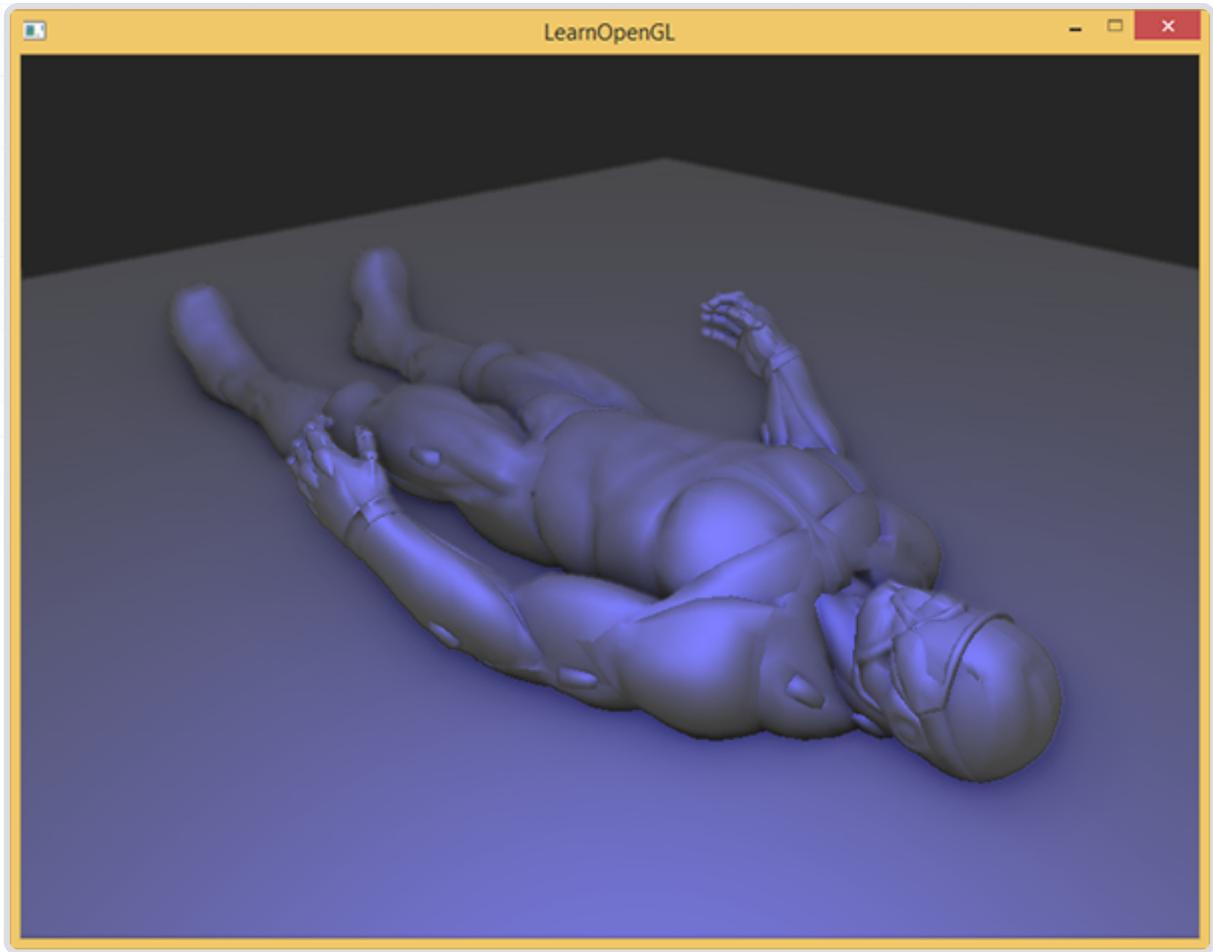
uniform Light light;

void main()
{
    // 从G缓冲中提取数据
    vec3 FragPos = texture(gPositionDepth, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Diffuse = texture(gAlbedo, TexCoords).rgb;
    float AmbientOcclusion = texture(ssao, TexCoords).r;

    // Blinn-Phong (观察空间中)
    vec3 ambient = vec3(0.3 * AmbientOcclusion); // 这里我们加上遮蔽因子
    vec3 lighting = ambient;
    vec3 viewDir = normalize(-FragPos); // Viewpos 为 (0.0, 0), 在观察空间中
    // 漫反射
    vec3 lightDir = normalize(light.Position - FragPos);
    vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * light.Color;
    // 镜面
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(Normal, halfwayDir), 0.0), 8.0);
    vec3 specular = light.Color * spec;
    // 衰减
    float dist = length(light.Position - FragPos);
    float attenuation = 1.0 / (1.0 + light.Linear * dist + light.Quadratic * dist * dist);
    diffuse *= attenuation;
    specular *= attenuation;
    lighting += diffuse + specular;

    FragColor = vec4(lighting, 1.0);
}
```

(除了将其改到观察空间)对比于之前的光照实现，唯一的真正改动就是场景环境分量与 `AmbientOcclusion` 值的乘法。通过在场景中加入一个淡蓝色的点光源，我们将会得到下面这个结果：



你可以在[这里](#)找到完整的源代码，和以下着色器：

- 几何：[顶点](#), [片段](#)
- SSAO：[顶点](#), [片段](#)
- 模糊：[顶点](#), [片段](#)
- 光照：[顶点](#), [片段](#)

屏幕空间环境遮蔽是一个可高度自定义的效果，它的效果很大程度上依赖于我们根据场景类型调整它的参数。对所有类型的场景并不存在什么完美的参数组合方式。一些场景只在小半径情况下工作，又有些场景会需要更大的半径和更大的样本数量才能看起来更真实。当前这个演示用了64个样本，属于比较多的了，你可以调调更小的核心大小从而获得更好的结果。

一些你可以调整(比如说通过uniform)的参数：核心大小，半径和/或噪声核心的大小。你也可以提升最终的遮蔽值到一个用户定义的幂从而增加它的强度：

```
occlusion = 1.0 - (occlusion / kernelSize);
FragColor = pow(occlusion, power);
```

多试试不同的场景和不同的参数，来欣赏SSAO的可定制性。

尽管SSAO是一个很微小的效果，可能甚至不是很容易注意到，它在很大程度上增加了合适光照场景的真实性，它也绝对是一个在你工具箱中必备的技术。

附加资源

- [SSAO教程](#) : John Chapman优秀的SSAO教程；本教程很大一部分代码和技巧都是基于他的文章
- [了解你的SSAO效果](#)：关于提高SSAO特定效果的一篇很棒的文章
- [深度值重构SSAO](#) : OGLDev的一篇在SSAO之上的拓展教程，它讨论了通过仅仅深度值重构位置矢量，节省了存储开销巨大的位置矢量到G缓冲的过程

2.7 PBR

2.7.1 理论

原文 [Theory](#)

作者 JoeyDeVries

翻译 [J.moons](#)

校对 Krasjet (初校)

Note

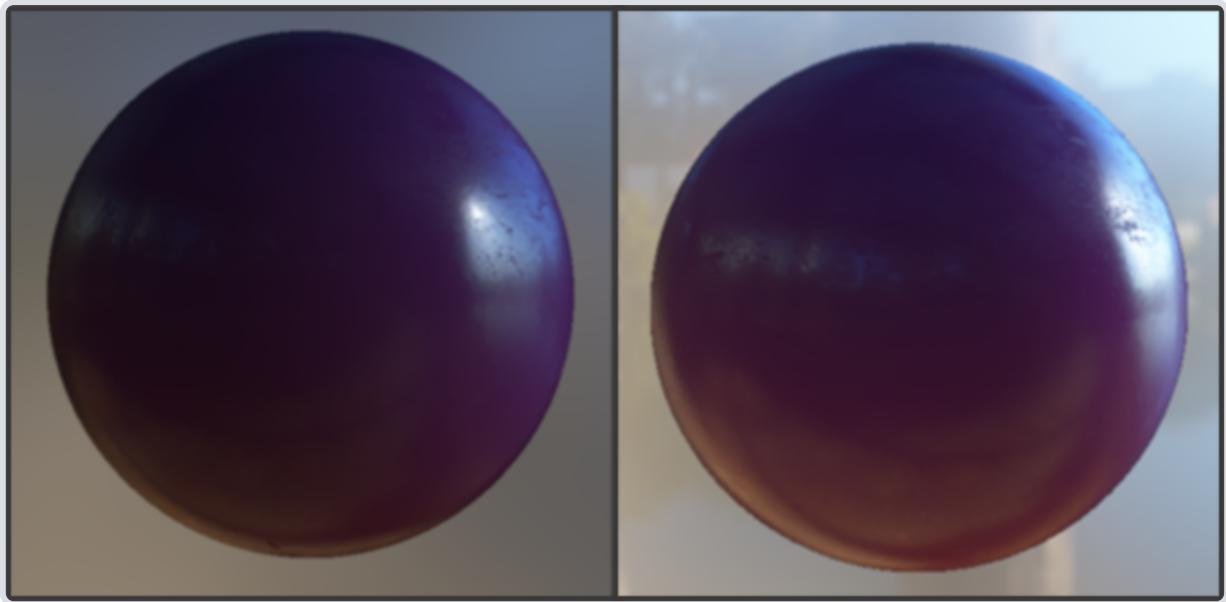
本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

PBR，或者用更通俗一些的称呼是指**基于物理的渲染**(Physically Based Rendering)，它指的是一些在不同程度上都基于与现实世界的物理原理更相符的基本理论所构成的渲染技术的集合。正因为基于物理的渲染目的便是为了使用一种更符合物理学规律的方式来模拟光线，因此这种渲染方式与我们原来的Phong或者Blinn-Phong光照算法相比总体上看起来要更真实一些。除了看起来更好些以外，由于它与物理性质非常接近，因此我们（尤其是美术师们）可以直接以物理参数为依据来编写表面材质，而不必依靠粗劣的修改与调整来让光照效果看上去正常。使用基于物理参数的方法来编写材质还有一个更大的好处，就是不论光照条件如何，这些材质看上去都会是正确的，而在非PBR的渲染管线当中有些东西就不会那么真实了。

虽然如此，基于物理的渲染仍然只是对基于物理原理的现实世界的一种近似，这也就是为什么它被称为基于物理的着色(Physically based Shading)而非物理着色(Physical Shading)的原因。判断一种PBR光照模型是否是基于物理的，必须满足以下三个条件（不用担心，我们很快就会了解它们的）：

1. 基于微平面(Microfacet)的表面模型。
2. 能量守恒。
3. 应用基于物理的BRDF。

在这次的PBR系列教程之中，我们将会把重点放在最先由迪士尼(Disney)提出探讨并被Epic Games首先应用于实时渲染的PBR方案。他们基于**金属质地工作流**(Metallic Workflow)的方案有非常完备的文献记录，广泛应用于各种流行的引擎之中并且有着非常令人惊叹的视觉效果。完成这次的教程之后我们将会制作出类似于这样的一些东西：



注意这次的理论教程是一个三部曲系列（目前还在创作中）的一部分，所以这部分教程的内容有可能随着其他两部还没完成的教程的进展而发生变化。同样，这部教程的源代码也并不完整，所以上面的图片是取自一个私人项目之中，而非本教程的代码所生成的。不过在完成之后这个图片看上去应该会好很多。

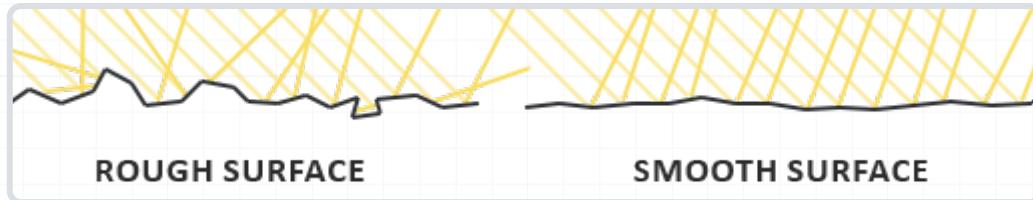
请注意这个系列的教程所探讨的内容属于相当高端的领域，因此要求读者对OpenGL和着色器光照有较好的理解。读者将会需要这些相关的知识：[帧缓冲](#)，[立方体贴图](#)，[Gamma校正](#)，[HDR](#)和[法线贴图](#)。我们还会深入探讨一些高等数学的内容，我会尽我所能将相关的概念阐述清楚。

微平面模型

所有的PBR技术都基于微平面理论。这项理论认为，达到微观尺度之后任何平面都可以用被称为**微平面**(Microfacets)的细小镜面来进行描绘。根据平面粗糙程度的不同，这些细小镜面的取向排列可以相当不一致：

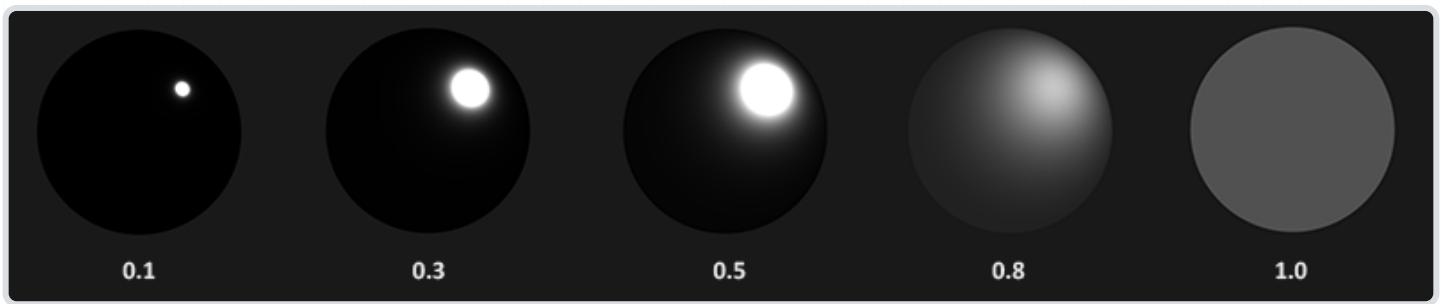


产生的效果就是：一个平面越是粗糙，这个平面上的微平面的排列就越混乱。这些微小镜面这样无序取向排列的影响就是，当我们特指镜面光/镜面反射时，入射光线更趋向于向完全不同的方向**发散**(Scatter)开来，进而产生出分布范围更广泛的镜面反射。而与之相反的是，对于一个光滑的平面，光线大体上会更趋向于向同一个方向反射，造成更小更锐利的反射：



在微观尺度下，没有任何平面是完全光滑的。然而由于这些微平面已经微小到无法逐像素的继续对其进行区分，因此我们只有假设一个**粗糙度**(Roughness)参数，然后用统计学的方法来概略的估算微平面的粗糙程度。我们可以基于一个平面的粗糙度来计算出某个向量的方向与微平面平均取向方向一致的概率。这个向量便是位于光线向量和视线向量之间的**中间向量**(Halfway Vector)。我们曾经在之前的[高级光照](#)教程中谈到过中间向量，它的计算方法如下：

微平面的取向方向与中间向量的方向越是一致，镜面反射的效果就越是强烈越是锐利。然后再加上一个介于0到1之间的粗糙度参数，这样我们就能概略的估算微平面的取向情况了：



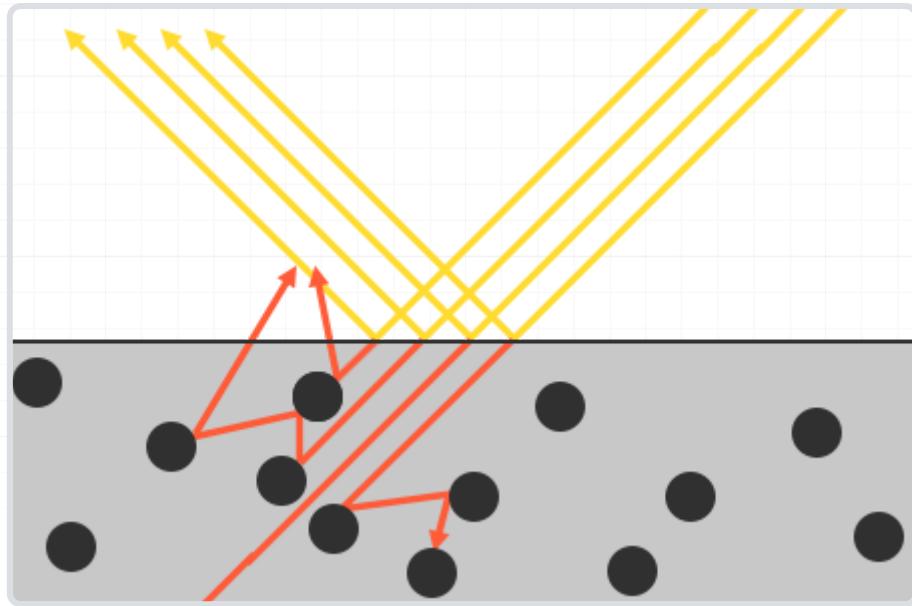
我们可以看到，较高的粗糙度值显示出来的镜面反射的轮廓要更大一些。与之相反地，较小的粗糙值显示出的镜面反射轮廓则更小更锐利。

能量守恒

微平面近似法使用了这样一种形式的**能量守恒**(Energy Conservation)：出射光线的能量永远不能超过入射光线的能量（发光面除外）。如图示我们可以看到，随着粗糙度的上升镜面反射区域的会增加，但是镜面反射的亮度却会下降。如果不管反射轮廓的大小而让每个像素的镜面反射强度(Specular Intensity)都一样的话，那么粗糙的平面就会放射出过多的能量，而这样就违背了能量守恒定律。这也就是为什么正如我们看到的一样，光滑平面的镜面反射更强烈而粗糙平面的反射更昏暗。

为了遵守能量守恒定律，我们需要对漫反射光和镜面反射光之间做出明确的区分。当一束光线碰撞到一个表面的时候，它就会分离成一个**折射**部分和一个**反射**部分。反射部分就是会直接反射开来而不会进入平面的那部分光线，这就是我们所说的镜面光照。而折射部分就是余下的会进入表面并被吸收的那部分光线，这也就是我们所说的漫反射光照。

这里还有一些细节需要处理，因为当光线接触到一个表面的时候折射光是不会立即就被吸收的。通过物理学我们可以得知，光线实际上可以被认为是一束没有耗尽就不停向前运动的能量，而光束是通过碰撞的方式来消耗能量。每一种材料都是由无数微小的粒子所组成，这些粒子都能如下图所示一样与光线发生碰撞。这些粒子在每次的碰撞中都可以吸收光线所携带的一部分或者是全部的能量而后转变成为热量。



一般来说，并非所有能量都会被全部吸收，而光线也会继续沿着（基本上）随机的方向发散，然后再和其他的粒子碰撞直至能量完全耗尽或者再次离开这个表面。而光线脱离物体表面后将会协同构成该表面的（漫反射）颜色。不过在基于物理的渲染之中我们进行了简化，假设对平面上的每一点所有的折射光都会被完全吸收而不会散开。而有一些被称为次表面散射(Subsurface Scattering)技术的着色器技术将这个问题考虑了进去，它们显著的提升了一些诸如皮肤，大理石或者蜡质这样材质的视觉效果，不过伴随而来的则是性能下降代价。

对于金属(Metallic)表面，当讨论到反射与折射的时候还有一个细节需要注意。金属表面对光的反应与非金属材料（也被称为介电质(Dielectrics)材料）表面相比是不同的。它们遵从的反射与折射原理是相同的，但是所有的折射光都会被直接吸收而不会散开，只留下反射光或者说镜面反射光。亦即是说，金属表面不会显示出漫反射颜色。由于金属与电介质之间存在这样明显的区别，因此它们两者在PBR渲染管线中被区别处理，而我们将在文章的后面进一步详细探讨这个问题。

反射光与折射光之间的这个区别使我们得到了另一条关于能量守恒的经验结论：反射光与折射光它们二者之间是互斥的关系。无论何种光线，其被材质表面所反射的能量将无法再被材质吸收。因此，诸如折射光这样的余下的进入表面之中的能量正好就是我们计算完反射之后余下的能量。

我们按照能量守恒的关系，首先计算镜面反射部分，它的值等于入射光线被反射的能量所占的百分比。然后折射光部分就可以直接由镜面反射部分计算得出：

```
float ks = calculateSpecularComponent(...); // 反射/镜面 部分
float kd = 1.0 - ks;                      // 折射/漫反射 部分
```

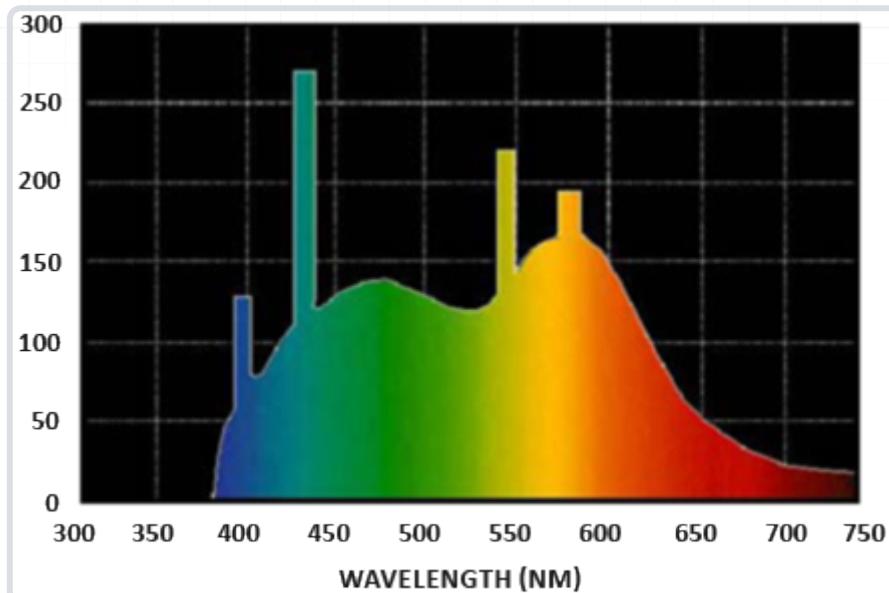
这样我们就能在遵守能量守恒定律的前提下知道入射光线的反射部分与折射部分所占的总量了。按照这种方法折射/漫反射与反射/镜面反射所占的份额都不会超过1.0，如此就能保证它们的能量总和永远不会超过入射光线的能量。而这些都是我们在前面的光照教程中没有考虑的问题。

反射率方程

在这里我们引入了一种被称为[渲染方程](#)(Render Equation)的东西。它是某些聪明绝顶人所构想出来的一个精妙的方程式，是如今我们所拥有的用来模拟光的视觉效果最好的模型。基于物理的渲染所坚定的遵循的是一种被称为[反射率方程](#)(The Reflectance Equation)的渲染方程的特化版本。要正确的理解PBR很重要的一点就是要首先透彻的理解反射率方程：

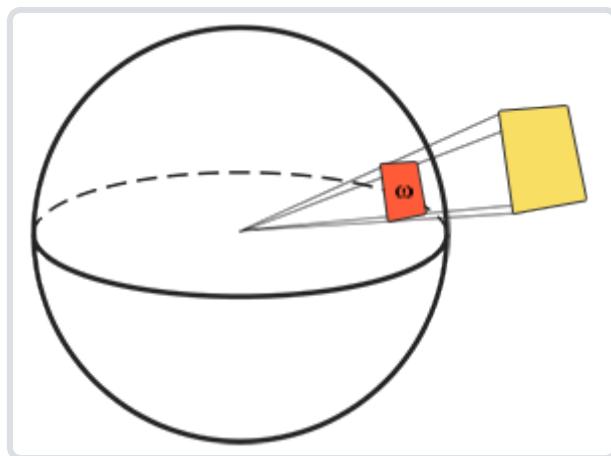
反射率方程一开始可能会显得有些吓人，不过随着我们慢慢对其进行剖析，读者最终会逐渐理解它的。要正确的理解这个方程式，我们必须稍微涉足一些**辐射度量学**(Radiometry)的内容。辐射度量学是一种用来度量电磁场辐射（包括可见光）的手段。有很多种辐射度量(radiometric quantities)可以用来测量曲面或者某个方向上的光，但是我们将只会讨论其中和反射率方程有关的一种。它被称为**辐射率**(Radiance)，在这里用来表示。辐射率被用来量化单一方向上发射来的光线的大小或者强度。由于辐射率是由许多物理变量集合而成的，一开始理解起来可能有些困难，因此我们首先关注一下这些物理量：

辐射通量：辐射通量表示的是一个光源所输出的能量，以瓦特为单位。光是由多种不同波长的能量所集合而成的，而每种波长则与一种特定的（可见的）颜色相关。因此一个光源所放射出来的能量可以被视作这个光源包含的所有各种波长的一个函数。波长介于390nm到700nm（纳米）的光被认为是处于可见光光谱中，也就是说它们是人眼可见的波长。在下面你可以看到一幅图片，里面展示了日光中不同波长的光所具有的能量：



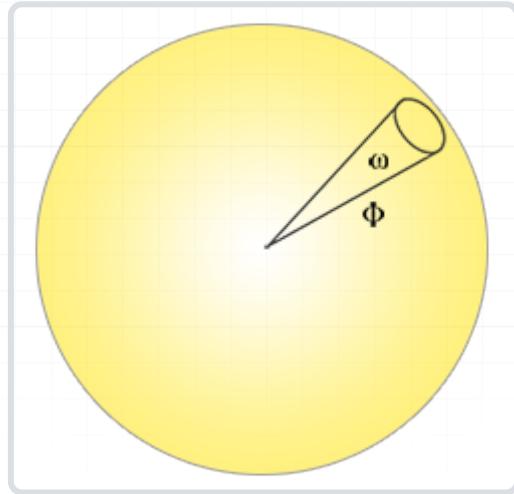
辐射通量将会计算这个由不同波长构成的函数的总面积。直接将这种对不同波长的计量作为参数输入计算机图形有一些不切实际，因此我们通常不直接使用波长的强度而是使用三原色编码，也就是**RGB**（或者按通常的称呼：光色）来作为辐射通量表示的简化。这套编码确实会带来一些信息上的损失，但是这对于视觉效果上的影响基本可以忽略。

立体角：立体角用表示，它可以为我们描述投射到单位球体上的一个截面的大小或者面积。投射到这个单位球体上的截面的面积就被称为**立体角**(Solid Angle)，你可以把立体角想象成为一个带有体积的方向：



可以把自己想象成为一个站在单位球面的中心的观察者，向着投影的方向看。这个投影轮廓的大小就是立体角。

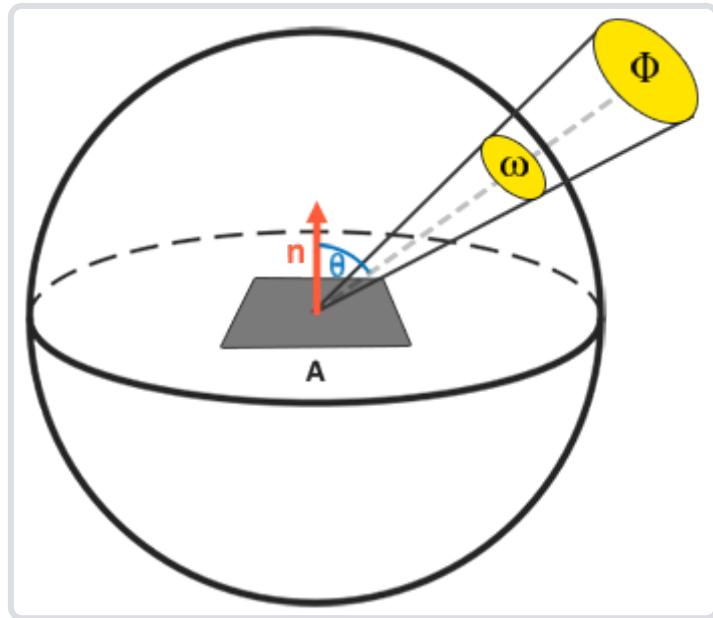
辐射强度：辐射强度(Radiant Intensity)表示的是在单位球面上，一个光源向每单位立体角所投送的辐射通量。举例来说，假设一个全向光源向所有方向均匀的辐射能量，辐射强度就能帮我们计算出它在一个单位面积（立体角）内的能量大小：



计算辐射强度的公式如下所示：

其中表示辐射通量除以立体角。

在理解了辐射通量，辐射强度与立体角的概念之后，我们终于可以开始讨论辐射率的方程式了。这个方程表示的是，一个拥有辐射强度的光源在单位面积，单位立体角上的辐射出的总能量：



辐射率是辐射度量学上表示一个区域平面上光线总量的物理量，它受到入射(Incident)（或者来射）光线与平面法线间的夹角的余弦值的影响：当直接辐射到平面上的程度越低时，光线就越弱，而当光线完全垂直于平面时强度最高。这和我们在前面的基础光照教程中对于漫反射光照的概念相似，其中就直接对应于光线的方向向量和平面法向量的点积：

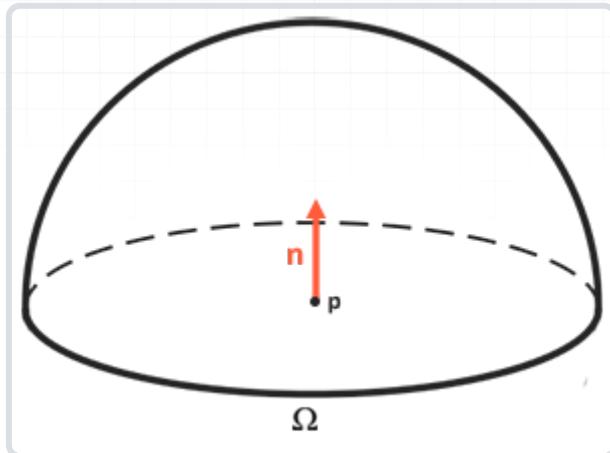
```
float cosTheta = dot(lightDir, N);
```

辐射率方程很有用，因为它把大部分我们感兴趣的物理量都包含了进去。如果我们把立体角和面积看作是无穷小的，那么我们就能用辐射率来表示单束光线穿过空间中的一个点的通量。这就使我们可以计算得出作用于单个（片段）点上的单束光线的辐射率，我们实际上把立体角转变为方向向量然后把面转换为点。这样我们就能直接在我们的着色器中使用辐射率来计算单束光线对每个片段的作用了。

事实上，当涉及到辐射率时，我们通常关心的是所有投射到点上的光线的总和，而这个和就称为辐射照度或者辐照度(Irradiance)。在理解了辐射率和辐照度的概念之后，让我们再回过头来看看反射率方程：

我们知道在渲染方程中代表通过某个无限小的立体角在某个点上的辐射率，而立体角可以视作是入射方向向量。注意我们利用光线和平面间的入射角的余弦值来计算能量，亦即从辐射率公式转化至反射率公式时的。用表示观察方向，也就是出射方向，反射率公式计算了点在方向上被反射出来的辐射率的总和。或者换句话说：表示了从方向上观察，光线投射到点上反射出来的辐照度。

基于反射率公式是围绕所有入射辐射率的总和，也就是辐照度来计算的，所以我们需要计算的就不只是单一的一个方向上的入射光，而是一个以点为球心的半球领域内所有方向上的入射光。一个半球领域(Hemisphere)可以描述为以平面法线为轴所环绕的半个球体：



为了计算某些面积的值，或者像是在半球领域的问题中计算某一个体积的时候我们会需要用到一种称为积分(Integral)的数学手段，也就是反射率公式中的符号，它的运算包含了半球领域内所有入射方向上的。积分运算的值等于一个函数曲线的面积，它的计算结果要么是解析解要么就是数值解。由于渲染方程和反射率方程都没有解析解，我们将会用离散的方法来求得这个积分的数值解。这个问题就转化为，在半球领域中按一定的步长将反射率方程分散求解，然后再按照步长大小将所得到的结果平均化。这种方法被称为黎曼和(Riemann sum)，我们可以用下面的代码粗略的演示一下：

```
int steps = 100;
float sum = 0.0f;
vec3 P    = ...;
vec3 Wo   = ...;
vec3 N    = ...;
float dW  = 1.0f / steps;
for(int i = 0; i < steps; ++i)
{
    vec3 Wi = getNextIncomingLightDir(i);
    sum += Fr(p, Wi, Wo) * L(p, Wi) * dot(N, Wi) * dW;
}
```

通过利用 `dW` 来对所有离散部分进行缩放，其和最后就等于积分函数的总面积或者总体积。这个用来对每个离散步长进行缩放的 `dW` 可以认为就是反射率方程中的 。在数学上，用来计算积分的 表示的是一个连续的符号，而我们使用的 `dW` 在代码中和它并没有直接的联系（因为它代表的是黎曼和中的离散步长），这样说是为了可以帮助你理解。请牢记，使用离散步长得到的是函数总面积的一个近似值。细心的读者可能已经注意到了，我们可以通过增加离散部分的数量来提高黎曼和的准确度(Accuracy)。

反射率方程概括了在半球领域内，碰撞到了点上的所有入射方向上的光线的辐射率，并受到的约束，然后返回观察方向上反射光的。正如我们所熟悉的那样，入射光辐射率可以由光源处获得，此外还可以利用一个环境贴图来测算所有入射方向上的辐射率，我们将在未来的IBL教程中讨论这个方法。

现在唯一剩下的未知符号就是了，它被称为BRDF，或者双向反射分布函数(Bidirectional Reflective Distribution Function)，它的作用是基于表面材质属性来对入射辐射率进行缩放或者加权。

BRDF

BRDF，或者说双向反射分布函数，它接受入射（光）方向，出射（观察）方向，平面法线以及一个用来表示微平面粗糙程度的参数作为函数的输入参数。BRDF可以近似的求出每束光线对一个给定了材质属性的平面上最终反射出来的光线所作出的贡献程度。举例来说，如果一个平面拥有完全光滑的表面（比如镜面），那么对于所有的入射光线（除了一束以外）而言BRDF函数都会返回0.0，只有一束与出射光线拥有相同（被反射）角度的光线会得到1.0这个返回值。

BRDF基于我们之前所探讨过的微平面理论来近似的求得材质的反射与折射属性。对于一个BRDF，为了实现物理学上的可信度，它必须遵守能量守恒定律，也就是说反射光线的总和永远不能超过入射光线的总量。严格上来说，同样采用和作为输入参数的 Blinn-Phong 光照模型也被认为是一个BRDF。然而由于Blinn-Phong模型并没有遵循能量守恒定律，因此它不被认为是基于物理的渲染。现在已经有很好几种BRDF都能近似的得出物体表面对于光的反应，但是几乎所有实时渲染管线使用的都是一种被称为Cook-Torrance BRDF模型。

Cook-Torrance BRDF兼有漫反射和镜面反射两个部分：

这里的是早先提到过的入射光线中被折射部分的能量所占的比率，而是被反射部分的比率。BRDF的左侧表示的是漫反射部分，这里用来表示。它被称为Lambertian漫反射，这和我们之前在漫反射着色中使用的常数因子类似，用如下的公式来表示：

表示表面颜色（回想一下漫反射表面纹理）。除以是为了对漫反射光进行标准化，因为前面含有BRDF的积分方程是受影响的（我们会在IBL的教程中探讨这个问题的）。

你也许会感到好奇，这个Lambertian漫反射和我们之前经常使用的漫反射到底有什么关系：之前我们是用表面法向量与光照方向向量进行点乘，然后再将结果与平面颜色相乘得到漫反射参数。点乘依然还在，但是却不在BRDF之内，而是转变成为了积分末公式末尾处的。

目前存在着许多不同类型的模型来实现BRDF的漫反射部分，大多看上去都相当真实，但是相应的运算开销也非常的昂贵。不过按照Epic公司给出的结论，Lambertian漫反射模型已经足够应付大多数实时渲染的用途了。

BRDF的镜面反射部分要稍微更高级一些，它的形式如下所示：

Cook-Torrance BRDF的镜面反射部分包含三个函数，此外分母部分还有一个标准化因子。字母D，F与G分别代表着一种类型的函数，各个函数分别用来近似的计算出表面反射特性的一个特定部分。三个函数分别为法线分布函数(Normal Distribution Function)，菲涅尔方程(Fresnel Rquation)和几何函数(Geometry Function)：

- 法线分布函数：估算在受到表面粗糙度的影响下，取向方向与中间向量一致的微平面的数量。这是用来估算微平面的主要函数。
- 几何函数：描述了微平面自成阴影的属性。当一个平面相对比较粗糙的时候，平面表面上的微平面有可能挡住其他的微平面从而减少表面所反射的光线。
- 菲涅尔方程：菲涅尔方程描述的是在不同的表面角下表面所反射的光线所占的比率。

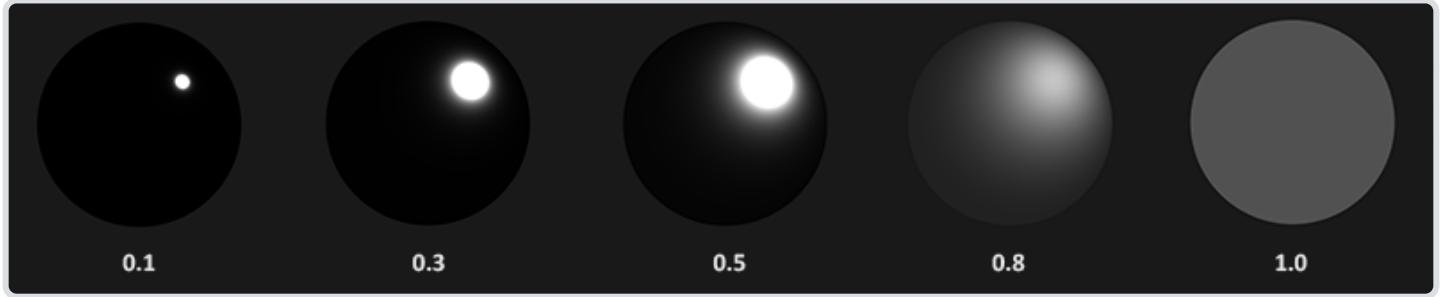
以上的每一种函数都是用来估算相应的物理参数的，而且你会发现用来实现相应物理机制的每种函数都有不止一种形式。它们有的非常真实，有的则性能高效。你可以按照自己的需求任意选择自己想要的函数的实现方法。英佩游戏公司的Brian Karis对于这些函数的多种近似实现方式进行了大量的研究。我们将会采用Epic Games在Unreal Engine 4中所使用的函数，其中D使用Trowbridge-Reitz GGX，F使用Fresnel-Schlick近似(Fresnel-Schlick Approximation)，而G使用Smith's Schlick-GGX。

法线分布函数

法线分布函数, 或者说镜面分布, 从统计学上近似的表示了与某些(中间)向量取向一致的微平面的比率。举例来说, 假设给定向量, 如果我们的微平面中有35%与向量取向一致, 则法线分布函数或者说NDF将会返回0.35。目前有很多种NDF都可以从统计学上来估算微平面的总体取向度, 只要给定一些粗糙度的参数以及一个我们马上将会要用到的参数Trowbridge-Reitz GGX:

在这里表示用来与平面上微平面做比较用的中间向量, 而表示表面粗糙度。

如果我们把当成是不同粗糙度参数下, 平面法向量和光线方向向量之间的中间向量的话, 我们可以得到如下图示的效果:



当粗糙度很低(也就是说表面很光滑)的时候, 与中间向量取向一致的微平面会高度集中在一个很小的半径范围内。由于这种集中性, NDF最终会生成一个非常明亮的斑点。但是当表面比较粗糙的时候, 微平面的取向方向会更加的随机。你将会发现与向量取向一致的微平面分布在一个大得多的半径范围内, 但是同时较低的集中性也会让我们的最终效果显得更加灰暗。

使用GLSL代码编写的Trowbridge-Reitz GGX法线分布函数是下面这个样子的:

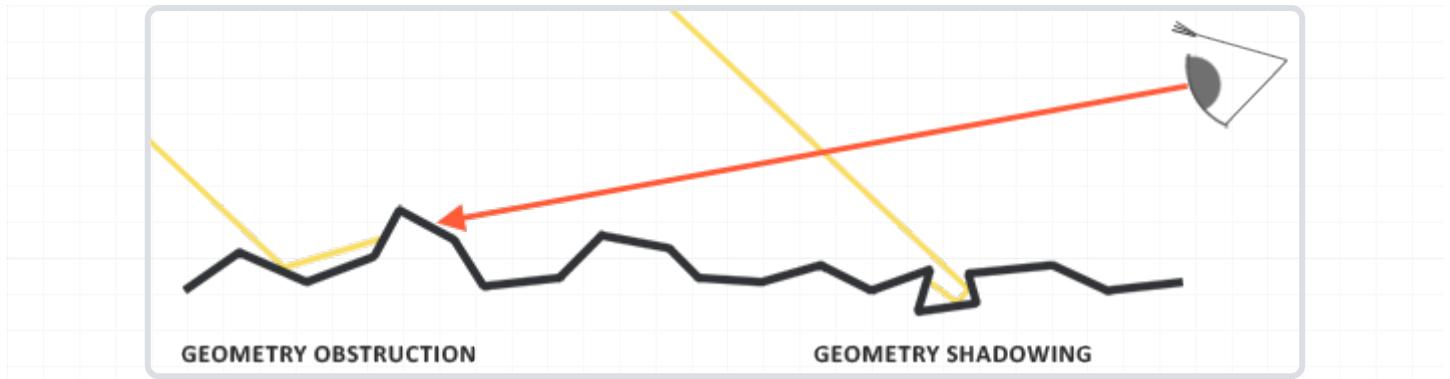
```
float D_GGX_TR(vec3 N, vec3 H, float a)
{
    float a2      = a*a;
    float NdotH  = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

    float nom    = a2;
    float denom  = (NdotH2 * (a2 - 1.0) + 1.0);
    denom       = PI * denom * denom;

    return nom / denom;
}
```

几何函数

几何函数从统计学上近似的求得了微平面间相互遮蔽的比率, 这种相互遮蔽会损耗光线的能量。



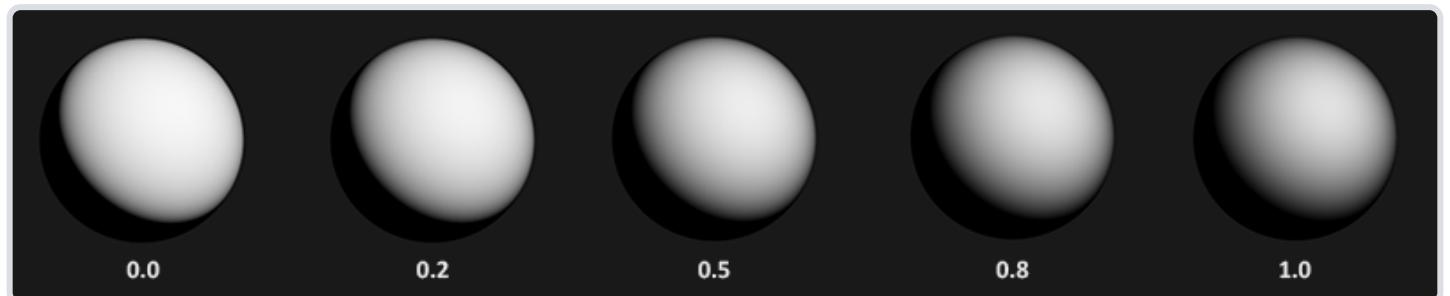
与NDF类似，几何函数采用一个材料的粗糙度参数作为输入参数，粗糙度较高的表面其微平面间相互遮蔽的概率就越高。我们将要使用的几何函数是GGX与Schlick-Beckmann近似的结合体，因此又称为Schlick-GGX：

这里的是基于几何函数是针对直接光照还是针对IBL光照的重映射(Remapping)：

注意，根据你的引擎把粗糙度转化为的方式不同，得到的值也有可能不同。在接下来的教程中，我们将会广泛的讨论这个重映射是如何起作用的。

为了有效的估算几何部分，需要将观察方向（几何遮蔽(Geometry Obstruction)）和光线方向向量（几何阴影(Geometry Shadowing)）都考虑进去。我们可以使用[史密斯法](#)(Smith's method)来把两者都纳入其中：

使用史密斯法与Schlick-GGX作为可以得到如下所示不同粗糙度的视觉效果：



几何函数是一个值域为[0.0, 1.0]的乘数，其中白色或者说1.0表示没有微平面阴影，而黑色或者说0.0则表示微平面彻底被遮蔽。

使用GLSL编写的几何函数代码如下：

```
float GeometrySchlickGGX(float NdotV, float k)
{
    float nom = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float k)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx1 = GeometrySchlickGGX(NdotV, k);
    float ggx2 = GeometrySchlickGGX(NdotL, k);

    return ggx1 * ggx2;
}
```

菲涅尔方程

菲涅尔（发音为Freh-nel）方程描述的是被反射的光线对比光线被折射的部分所占的比率，这个比率会随着我们观察的角度不同而不同。当光线碰撞到一个表面的时候，菲涅尔方程会根据观察角度告诉我们被反射的光线所占的百分比。利用这个反射比率和能量守恒原则，我们可以直接得出光线被折射的部分以及光线剩余的能量。

当垂直观察的时候，任何物体或者材质表面都有一个**基础反射率**(Base Reflectivity)，但是如果以一定的角度往平面上看的时候所有反光都会变得明显起来。你可以自己尝试一下，用垂直的视角观察你自己的木制/金属桌面，此时一定只有最基本的反射性。但是如果你从近乎90度（译注：应该是指和法线的夹角）的角度观察的话反光就会变得明显的多。如果从理想的90度视角观察，所有的平面理论上来说都能完全的反射光线。这种现象因**菲涅尔**而闻名，并体现在了菲涅尔方程之中。

菲涅尔方程是一个相当复杂的方程式，不过幸运的是菲涅尔方程可以用**Fresnel-Schlick**近似法求得近似解：

表示平面的基础反射率，它是利用所谓折射指数(Indices of Refraction)或者说IOR计算得出的。然后正如你可以从球体表面看到的那样，我们越是朝球面掠角的方向上看（此时视线和表面法线的夹角接近90度）菲涅尔现象就越明显，反光就越强：



菲涅尔方程还存在一些细微的问题。其中一个问题是在Fresnel-Schlick近似仅仅对电介质或者说非金属表面有定义。对于导体(Conductor)表面（金属），使用它们的折射指数计算基础折射率并不能得出正确的结果，这样我们就需要使用一种不同的菲涅尔方程来对导体表面进行计算。由于这样很不方便，所以我们预先计算出平面对于法向入射（）的反应（处于0度角，好像直接看向表面一样）然后基于相应观察角的Fresnel-Schlick近似对这个值进行插值，用这种方法来进行进一步的估算。这样我们就能对金属和非金属材质使用同一个公式了。

平面对于法向入射的响应或者说基础反射率可以在一些大型数据库中找到，比如[这个](#)。下面列举的这一些常见数值就是从Naty Hoffman的课程讲义中所得到的：

材料	\(F_0\)	(线性)	\(F_0\)	(sRGB)	颜色
水		(0.02, 0.02, 0.02)	(0.15, 0.15, 0.15)		
塑料/玻璃（低）		(0.03, 0.03, 0.03)	(0.21, 0.21, 0.21)		
塑料（高）		(0.05, 0.05, 0.05)	(0.24, 0.24, 0.24)		
玻璃（高）/红宝石	(0.08, 0.08, 0.08)	(0.31, 0.31, 0.31)			
钻石	(0.17, 0.17, 0.17)	(0.45, 0.45, 0.45)			
铁	(0.56, 0.57, 0.58)	(0.77, 0.78, 0.78)			
铜	(0.95, 0.64, 0.54)	(0.98, 0.82, 0.76)			
金	(1.00, 0.71, 0.29)	(1.00, 0.86, 0.57)			
铝	(0.91, 0.92, 0.92)	(0.96, 0.96, 0.97)			
银	(0.95, 0.93, 0.88)	(0.98, 0.97, 0.95)			

这里可以观察到的一个有趣的现象，所有电介质材质表面的基础反射率都不会高于0.17，这其实是例外而非普遍情况。导体材质表面的基础反射率起点更高一些并且（大多）在0.5和1.0之间变化。此外，对于导体或者金属表面而言基础反射率一般是带有色彩的，这也是为什么要用RGB三原色来表示的原因（法向入射的反射率可随波长不同而不同）。这种现象我们只能在金属表面观察的到。

金属表面这些和电介质表面相比所独有的特性引出了所谓的金属工作流的概念。也就是我们需要额外使用一个被称为金属度(Metalness)的参数来参与编写表面材质。金属度用来描述一个材质表面是金属还是非金属的。

理论上来说，一个表面的金属度应该是二元的：要么是金属要么不是金属，不能两者皆是。但是，大多数的渲染管线都允许在0.0至1.0之间线性的调配金属度。这主要是由于材质纹理精度不足以描述一个拥有诸如细沙/沙状粒子/刮痕的金属表面。通过对这些小的类非金属粒子/刮痕调整金属度值，我们可以获得非常好看的效果。

通过预先计算电介质与导体的值，我们可以对两种类型的表面使用相同的Fresnel-Schlick近似，但是如果是金属表面的话就需要对基础反射率添加色彩。我们一般是按下面这个样子来实现的：

```
vec3 F0 = vec3(0.04);
F0      = mix(F0, surfaceColor.rgb, metalness);
```

我们为大多数电介质表面定义了一个近似的基础反射率。取最常见的电解质表面的平均值，这又是一个近似值。不过对于大多数电介质表面而言使用0.04作为基础反射率已经足够好了，而且可以在不需要输入额外表面参数的情况下得到物理可信的结果。然后，基于金属表面特性，我们要么使用电介质的基础反射率要么就使用来作为表面颜色。因为金属表面会吸收所有折射光线而没有漫反射，所以我们可以直接使用表面颜色纹理来作为它们的基础反射率。

Fresnel Schlick近似可以用代码表示为：

```
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}
```

其中 `cosTheta` 是表面法向量与观察方向的点乘的结果。

Cook-Torrance反射率方程

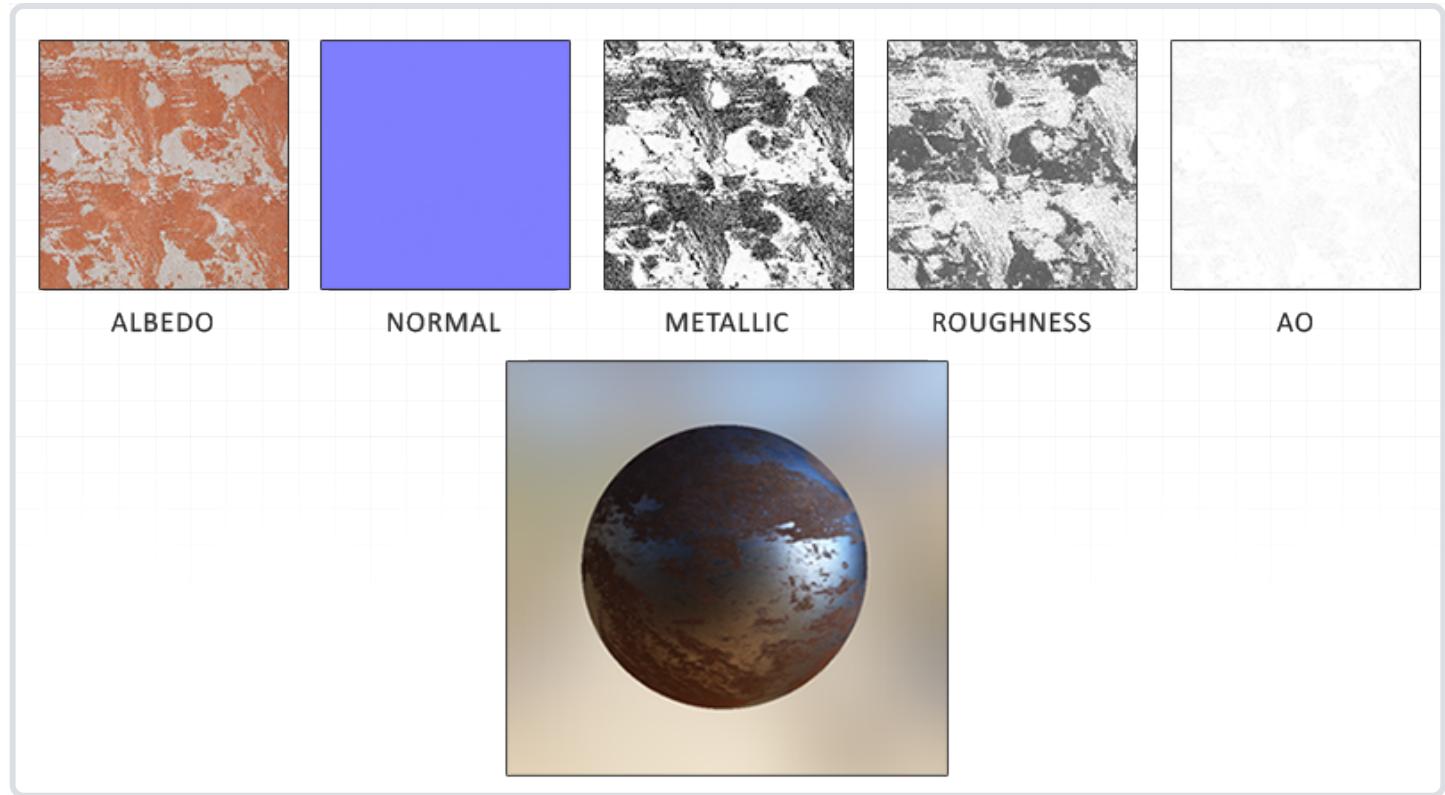
随着Cook-Torrance BRDF中所有元素都介绍完毕，我们现在可以将基于物理的BRDF纳入到最终的反射率方程当中去了：

这个方程现在完整的描述了一个基于物理的渲染模型，它现在可以认为就是我们一般意义上理解的基于物理的渲染也就是PBR。如果你还没有能完全理解我们将如何把所有这些数学运算结合到一起并融入到代码当中去的话也不必担心。在下一个教程当中，我们将探索如何实现反射率方程来在我们渲染的光照当中获得更加物理可信的结果，而所有这些零零星星的碎片将会慢慢组合到一起来。

编写PBR材质

在了解了PBR后面的数学模型之后，最后我们将通过说明美术师一般是如何编写一个我们可以直接输入PBR的平面物理属性的来结束这部分的讨论。PBR渲染管线所需要的每一个表面参数都可以用纹理来定义或者建模。使用纹理可以让我们逐个片段的来控制每个表面上特定的点对于光线是如何响应的：不论那个点是金属的，粗糙或者平滑，也不论表面对于不同波长的光会有如何的反应。

在下面你可以看到在一个PBR渲染管线当中经常会碰到的纹理列表，还有将它们输入PBR渲染器所能得到的相应的视觉输出：



反照率：反照率(Albedo)纹理为每一个金属的纹素(Texel)（纹理像素）指定表面颜色或者基础反射率。这和我们之前使用过的漫反射纹理相当类似，不同的是所有光照信息都是由一个纹理中提取的。漫反射纹理的图像当中常常包含一些细小的阴影或者深色的裂纹，而反照率纹理中是不会有这些东西的。它应该只包含表面的颜色（或者折射吸收系数）。

法线：法线贴图纹理和我们之前在[法线贴图](#)教程中所使用的贴图是完全一样的。法线贴图使我们可以逐片段的指定独特的法线，来为表面制造出起伏不平的假象。

金属度：金属(Metallic)贴图逐个纹素的指定该纹素是不是金属质地的。根据PBR引擎设置的不同，美术师们既可以将金属度编写为灰度值又可以编写为1或0这样的二元值。

粗糙度：粗糙度(Roughness)贴图可以以纹素为单位指定某个表面有多粗糙。采样得来的粗糙度数值会影响一个表面的微平面统计学上的取向度。一个比较粗糙的表面会得到更宽阔更模糊的镜面反射（高光），而一个比较光滑的表面则会得到集中而清晰的镜面反射。某些PBR引擎预设采用的是对某些美术师来说更加直观的光滑度(Smoothness)贴图而非粗糙度贴图，不过这些数值在采样之时就马上用 $(1 - \text{光滑度})$ 转换成了粗糙度。

AO：环境光遮蔽(Ambient Occlusion)贴图或者说AO贴图为表面和周围潜在的几何图形指定了一个额外的阴影因子。比如如果我们有一个砖块表面，反照率纹理上的砖块裂缝部分应该没有任何阴影信息。然而AO贴图则会把那些光线较难逃逸出来的暗色边缘指定出来。在光照的结尾阶段引入环境遮蔽可以明显的提升你场景的视觉效果。网格/表面的环境遮蔽贴图要么通过手动生成，要么由3D建模软件自动生成。

美术师们可以在纹素级别设置或调整这些基于物理的输入值，还可以以现实世界材料的表面物理性质来建立他们的材质数据。这是PBR渲染管线最大的优势之一，因为不论环境或者光照的设置如何改变这些表面的性质是不会改变的，这使得美术师们可以更便捷的获取物理可信的结果。在PBR渲染管线中编写的表面可以非常方便的在不同的PBR渲染引擎间共享使用，不论处于何种环境中它们看上去都会是正确的，因此看上去也会更自然。

延伸阅读

- [Background: Physics and Math of Shading by Naty Hoffmann](#): 由于在这一篇文章中要谈论的理论点太多，所以这里的理论知识都只是涉及到了皮毛。如果你希望了解更多关于光线背后的物理知识以及它们和PBR理论之间有什么关联的话，这才是你需要阅读的资源。
- [Real shading in Unreal Engine 4](#): 探讨了Epic Games在他们的Unreal 4引擎中所采用的PBR模型。我们这些教程中主要涉及的PBR系统就是基于他们的PBR模型。
- [Marmoset: PBR Theory](#): 主要针对美术师的PBR介绍，不过仍然是很好的读物。
- [Coding Labs: Physically based rendering](#): 介绍渲染方程以及它和PBR直接的关系。
- [Coding Labs: Physically Based Rendering - Cook–Torrance](#): 介绍了Cook-Torrance BRDF。
- [Wolfire Games - Physically based rendering](#): 介绍了PBR，由Lukas Orsvärn所著。

2.7.2 光照

原文 [Lighting](#)

作者 JoeyDeVries

翻译 [KenLee](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

译者注：

阅读本节请熟悉上一节提到的几个名词：

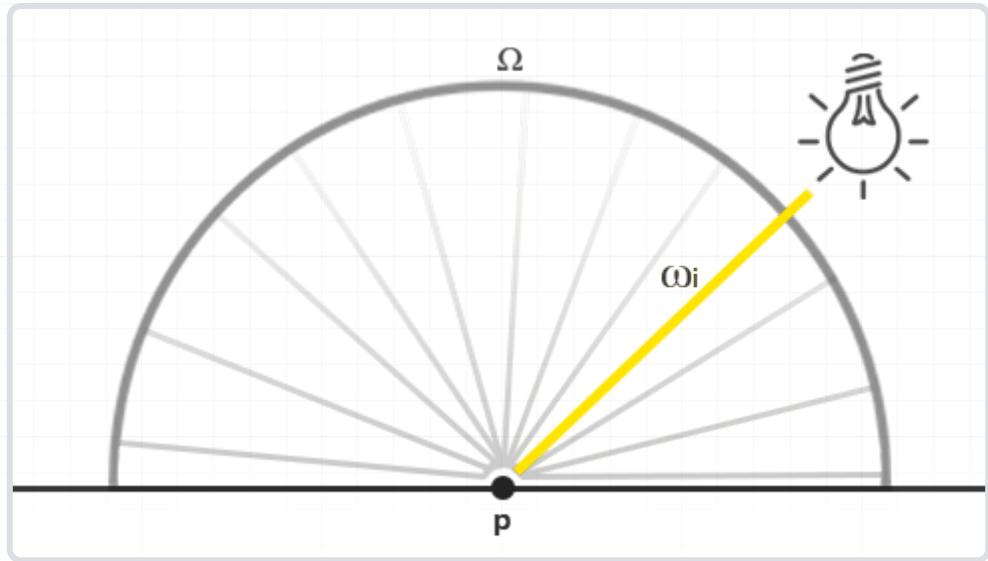
- 辐射通量(Radiant flux)
- 辐射率(Radiance)
- 辐照度(Irradiance)
- 辐射强度(Radiant Intensity)

在[上一个教程](#)中，我们讨论了一些PBR渲染的基础知识。在本章节中，我们将重点放在把以前讨论过的理论转化为实际的渲染器，这个渲染器将使用直接的（或解析的）光源：比如点光源，定向灯或聚光灯。

我们先来看看上一个章提到的反射方程的最终版：

我们大致上清楚这个反射方程在干什么，但我们仍然留有一些迷雾尚未揭开。比如说我们究竟将怎样表示场景上的辐照度(Irradiance)，辐射率(Radiance)？我们知道辐射率（在计算机图形领域中）表示在给定立体角的情况下光源的辐射通量(Radiant flux)或光源在角度下发送出来的光能。在我们的情况下，不妨假设立体角无限小，这样辐射度就表示光源在一条光线或单个方向向量上的辐射通量。

基于以上的知识，我们如何将其转化为以前的教程中积累的一些光照知识呢？那么想象一下，我们有一个点光源（一个光源在所有方向具有相同的亮度），它的辐射通量为用RGB表示为(23.47,21.31,20.79)。该光源的辐射强度(Radiant Intensity)等于其在所有出射光线的辐射通量。然而，当我们为一个表面上的特定的点着色时，在其半球领域的所有可能的入射方向上，只有一个入射方向向量直接来自于该点光源。假设我们在场景中只有一个光源，位于空间中的某一个点，因而对于点的其他可能的入射光线方向上的辐射率为0：



如果从一开始，我们就假设点光源不受光线衰减（光照强度会随着距离变暗）的影响，那么无论我们把光源放在哪，入射光线的辐射率总是一样的（除去入射角对辐射率的影响之外）。正是因为无论我们从哪个角度观察它，点光源总具有相同的辐射强度，我们可以有效地将其辐射强度建模为其辐射通量：一个常量向量 (23.47, 21.31, 20.79)。

然而，辐射率也需要将位置作为输入，正如所有现实的点光源都会受光线衰减影响一样，点光源的辐射强度应该根据点所在的位置和光源的位置以及他们之间的距离而做一些缩放。因此，根据原始的辐射方程，我们会根据表面法向量和入射角度来缩放光源的辐射强度。

在实现上来说：对于直接点光源的情况，辐射率函数先获取光源的颜色值，然后光源和某点的距离衰减，接着按照缩放，但是仅仅有一条入射角为的光线打在点上，这个同时也等于在点光源的方向向量。写成代码的话会是这样：

```
vec3 lightColor = vec3(23.47, 21.31, 20.79);
vec3 wi        = normalize(lightPos - fragPos);
float cosTheta = max(dot(N, Wi), 0.0);
float attenuation = calculateAttenuation(fragPos, lightPos);
float radiance   = lightColor * attenuation * cosTheta;
```

除了一些叫法上的差异以外，这段代码对你们来说应该很TM熟悉：这正是我们一直以来怎么计算(漫反射(diffuse))光照的！当涉及到直接照明(direct lighting)时，辐射率的计算方式和我们之前计算当只有一个光源照射在物体表面的时候非常相似。

请注意，这个假设是成立的条件是点光源体积无限小，相当于在空间中的一个点。如果我们认为该光源是具有体积的，它的辐射会在一个以上的入射光的方向不等于零。

对于其它类型的从单点发出来的光源我们类似地计算出辐射率。比如，定向光(directional light)拥有恒定的而不会有衰减因子；而一个聚光灯光源则没有恒定的辐射强度，其辐射强度是根据聚光灯的方向向量来缩放的。

这也让我们回到了对于表面的半球领域(hemisphere)的积分上。由于我们事先知道的所有贡献光源的位置，因此对物体表面上的一个点着色并不需要我们尝试去求解积分。我们可以直接拿光源的（已知的）数目，去计算它们的总辐照度，因为每个光源仅仅只有一个方向上的光线会影响物体表面的辐射率。这使得PBR对直接光源的计算相对简单，因为我们只需要有效地遍历所有有贡献的光源。而当我们后来把环境照明也考虑在内的[IBL](#)教程中，我们就必须采取积分去计算了，这是因为光线可能会在任何一个方向入射。

2.7.3 一个PBR表面模型

现在让我们开始写片段着色器来实现上述的PBR模型吧~ 首先我们需要把PBR相关的输入放进片段着色器。

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
in vec3 WorldPos;
in vec3 Normal;

uniform vec3 camPos;

uniform vec3 albedo;
uniform float metallic;
uniform float roughness;
uniform float ao;
```

我们把通用的顶点着色器的输出作为输入的一部分。另一部分输入则是物体表面模型的一些材质参数。

然后再片段着色器的开始部分我们做一下任何光照算法都需要做的计算:

```
void main()
{
    vec3 N = normalize(Normal);
    vec3 V = normalize(camPos - WorldPos);
    [...]
}
```

直接光照明

在本教程的例子中我们会采用总共4个点光源来直接表示场景的辐照度。为了满足反射率方程，我们循环遍历每一个光源，计算他们独立的辐射率然后求和，接着根据BRDF和光源的入射角来缩放该辐射率。我们可以把循环当作在对物体的半球领域对所以直接光源求积分。首先我们来计算一些可以预计的光照变量：

```
vec3 Lo = vec3(0.0);
for(int i = 0; i < 4; ++i)
{
    vec3 L = normalize(lightPositions[i] - WorldPos);
    vec3 H = normalize(V + L);

    float distance = length(lightPositions[i] - WorldPos);
    float attenuation = 1.0 / (distance * distance);
    vec3 radiance = lightColors[i] * attenuation;
    [...]
```

由于我们线性空间内计算光照（我们会在着色器的尾部进行Gamma校正），我们使用在物理上更为准确的平方倒数作为衰减。

相对于物理上正确来说，你可能仍然想使用常量，线性或者二次衰减方程（他们在物理上相对不准确），却可以为您提供提供在光的能量衰减更多的控制。

然后，对于每一个光源我们都想计算完整的 Cook-Torrance specular BRDF项：

首先我们想计算的是镜面反射和漫反射的系数，或者说发生表面反射和折射的光线的比值。我们从[上一个教程](#)知道可以使用菲涅尔方程计算：

```
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}
```

菲涅尔方程返回的是一个物体表面光线被反射的百分比，也就是我们反射方程中的参数。Fresnel-Schlick近似接受一个参数 `F0`，被称为 0° 入射角的反射(surface reflection at zero incidence)表示如果直接(垂直)观察表面的时候有多少光线会被反射。这个参数 `F0` 会因为材料不同而不同，而且会因为材质是金属而发生变色。在PBR金属流中我们简单地认为大多数的绝缘体在 `F0` 为0.04的时候看起来视觉上是正确的，我们同时会特别指定 `F0` 当我们遇到金属表面并且给定反射率的时候。因此代码上看起来会像是这样：

```
vec3 F0 = vec3(0.04);
F0      = mix(F0, albedo, metallic);
vec3 F  = fresnelSchlick(max(dot(H, V), 0.0), F0);
```

你可以看到，对于非金属材质来说 `F0` 永远保持0.04这个值，我们会根据表面的金属性来改变 `F0` 这个值，并且在原来的 `F0` 和反射率中插值计算 `F0`。

我们已经算出，剩下的项就是计算法线分布函数和几何遮蔽函数了。

因此一个直接PBR光照着色器中和的计算代码类似于：

```

float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a      = roughness*roughness;
    float a2     = a*a;
    float NdotH  = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

    float nom   = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;

    return nom / denom;
}

float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r*r) / 8.0;

    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}
float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2  = GeometrySchlickGGX(NdotV, roughness);
    float ggx1  = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}

```

这里比较重要的是和[上一个教程](#)不同的是，我们直接传了粗糙度(roughness)参数给上述的函数；通过这种方式，我们可以针对每一个不同的项对粗糙度做一些修改。根据迪士尼公司给出的观察以及后来被Epic Games公司采用的光照模型，光照在几何遮蔽函数和正太分布函数中采用粗糙度的平方会让光照看起来更加自然。

现在两个函数都给出了定义，在计算反射的循环中计算NDF和G项变得非常自然：

```

float NDF = DistributionGGX(N, H, roughness);
float G   = GeometrySmith(N, V, L, roughness);

```

这样我们就凑够了足够的项来计算Cook-Torrance BRDF：

```

vec3 nominator   = NDF * G * F;
float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001;
vec3 specular    = nominator / denominator;

```

注意我们在分母项中加了一个0.001为了避免出现除零错误。

现在我们终于可以计算每个光源在反射率方程中的贡献值了！因为菲涅尔方程直接给出了，我们可以使用 F 表示镜面反射在所有打在物体表面上的光线的贡献。从我们很容易计算折射的比值：

```
vec3 kS = F;
vec3 kD = vec3(1.0) - kS;

kD *= 1.0 - metallic;
```

我们可以看作表示光能中被反射的能量的比例，而剩下的光能会被折射，比值即为。更进一步来说，因为金属不会折射光线，因此不会有漫反射。所以如果表面是金属的，我们会把系数变为0。这样，我们终于集齐所有变量来计算我们出射光线的值：

```
const float PI = 3.14159265359;

float NdotL = max(dot(N, L), 0.0);
Lo += (kD * albedo / PI + specular) * radiance * NdotL;
}
```

最终的结果 Lo ，或者说是出射光线的辐射率，实际上是反射率方程的在半球领域的积分的结果。但是我们实际上不需要去求积，因为对于所有可能的入射光线方向我们知道只有4个方向的入射光线会影响片段(像素)的着色。因为这样，我们可以直接循环N次计算这些入射光线的方向(N 也就是场景中光源的数目)。

比较重要的是我们没有把 kS 乘进去我们的反射率方程中，这是因为我们已经在specular BRDF中乘了菲涅尔系数 F 了，因为 kS 等于 F ，因此我们不需要再乘一次。

剩下的工作就是加一个环境光照项给 Lo ，然后我们就拥有了片段的最后颜色：

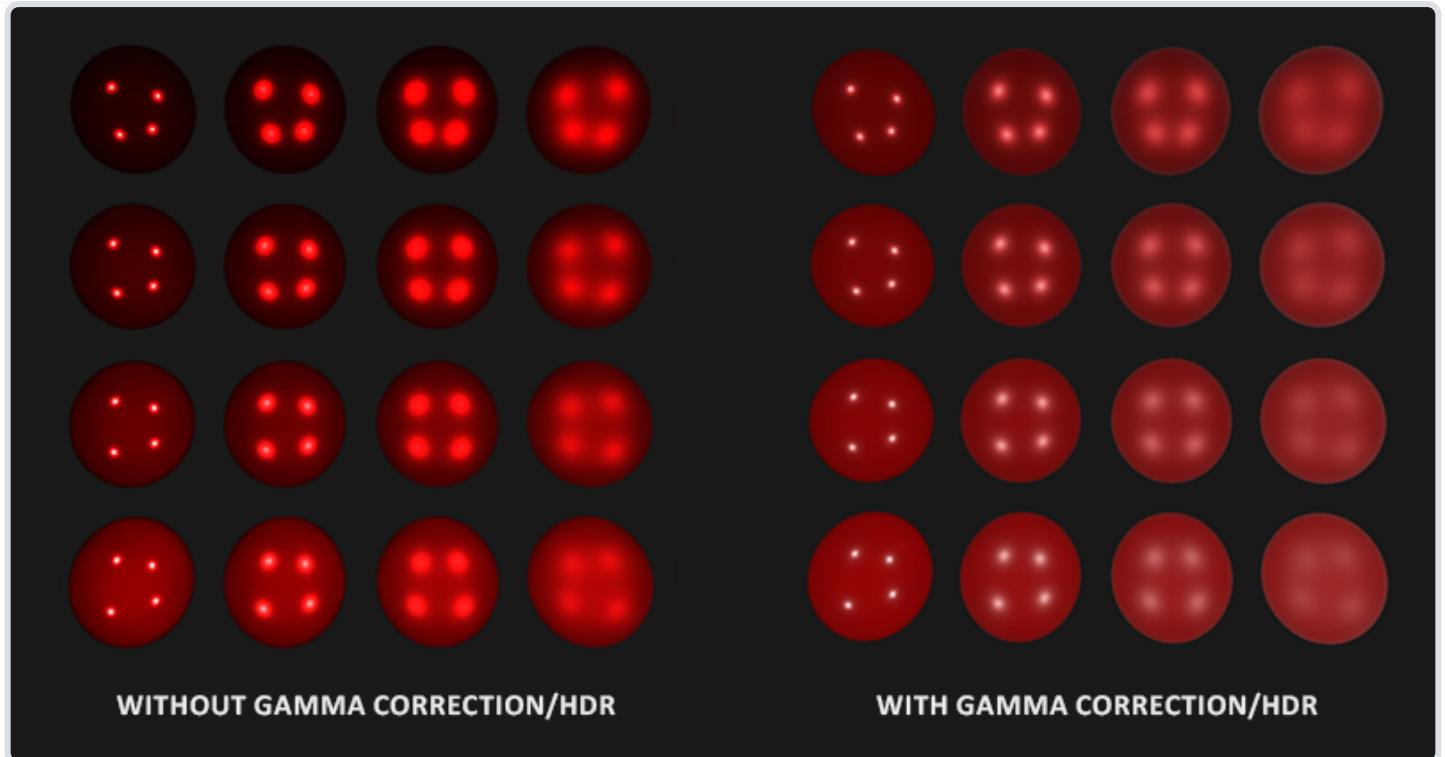
```
vec3 ambient = vec3(0.03) * albedo * ao;
vec3 color = ambient + Lo;
```

线性空间和HDR渲染

直到现在，我们假设的所有计算都在线性的颜色空间中进行的，因此我们需要在着色器最后做[伽马矫正](#)。在线性空间中计算光照是非常重要的，因为PBR要求所有输入都是线性的，如果不是这样，我们就会得到不正常的光照。另外，我们希望所有光照的输入都尽可能的接近他们在物理上的取值，这样他们的反射率或者说颜色值就会在色谱上有比较大的变化空间。 Lo 作为结果可能会变大得很快(超过1)，但是因为默认的LDR输入而取值被截断。所以在伽马矫正之前我们采用色调映射使 Lo 从LDR的值映射为HDR的值。

```
color = color / (color + vec3(1.0));
color = pow(color, vec3(1.0/2.2));
```

这里我们采用的色调映射方法为Reinhard 操作，使得我们在伽马矫正后可以保留尽可能多的辐照度变化。我们没有使用一个独立的帧缓冲或者采用后期处理，所以我们需要直接在每一步光照计算后采用色调映射和伽马矫正。



采用线性颜色空间和HDR在PBR渲染管线中非常重要。如果没有这些操作，几乎是不可能正确地捕获到因光照强度变化的细节，这最终会导致你的计算变得不正确，在视觉上看上去非常不自然。

完整的直接光照PBR着色器

现在剩下的事情就是把做好色调映射和伽马矫正的颜色值传给片段着色器的输出，然后我们就拥有了自己的直接光照PBR着色器。为了完整性，这里给出了完整的代码：

```

#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
in vec3 WorldPos;
in vec3 Normal;

// material parameters
uniform vec3 albedo;
uniform float metallic;
uniform float roughness;
uniform float ao;

// lights
uniform vec3 lightPositions[4];
uniform vec3 lightColors[4];

uniform vec3 camPos;

const float PI = 3.14159265359;

float DistributionGGX(vec3 N, vec3 H, float roughness);
float GeometrySchlickGGX(float NdotV, float roughness);
float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness);
vec3 fresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness);

void main()
{
    vec3 N = normalize(Normal);
    vec3 V = normalize(camPos - WorldPos);

    vec3 F0 = vec3(0.04);
    F0 = mix(F0, albedo, metallic);

    // reflectance equation
    vec3 Lo = vec3(0.0);
    for(int i = 0; i < 4; ++i)
    {
        // calculate per-light radiance
        vec3 L = normalize(lightPositions[i] - WorldPos);
        vec3 H = normalize(V + L);
        float distance = length(lightPositions[i] - WorldPos);
        float attenuation = 1.0 / (distance * distance);
        vec3 radiance = lightColors[i] * attenuation;

        // cook-torrance brdf
        float NDF = DistributionGGX(N, H, roughness);
        float G = GeometrySmith(N, V, L, roughness);
        vec3 F = fresnelSchlick(max(dot(H, V), 0.0), F0);

        vec3 kS = F;
        vec3 kD = vec3(1.0) - kS;
        kD *= 1.0 - metallic;

        vec3 nominator = NDF * G * F;
        float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001;
        vec3 specular = nominator / denominator;

        // add to outgoing radiance Lo
        float NdotL = max(dot(N, L), 0.0);

```

```

    Lo += (kD * albedo / PI + specular) * radiance * NdotL;
}

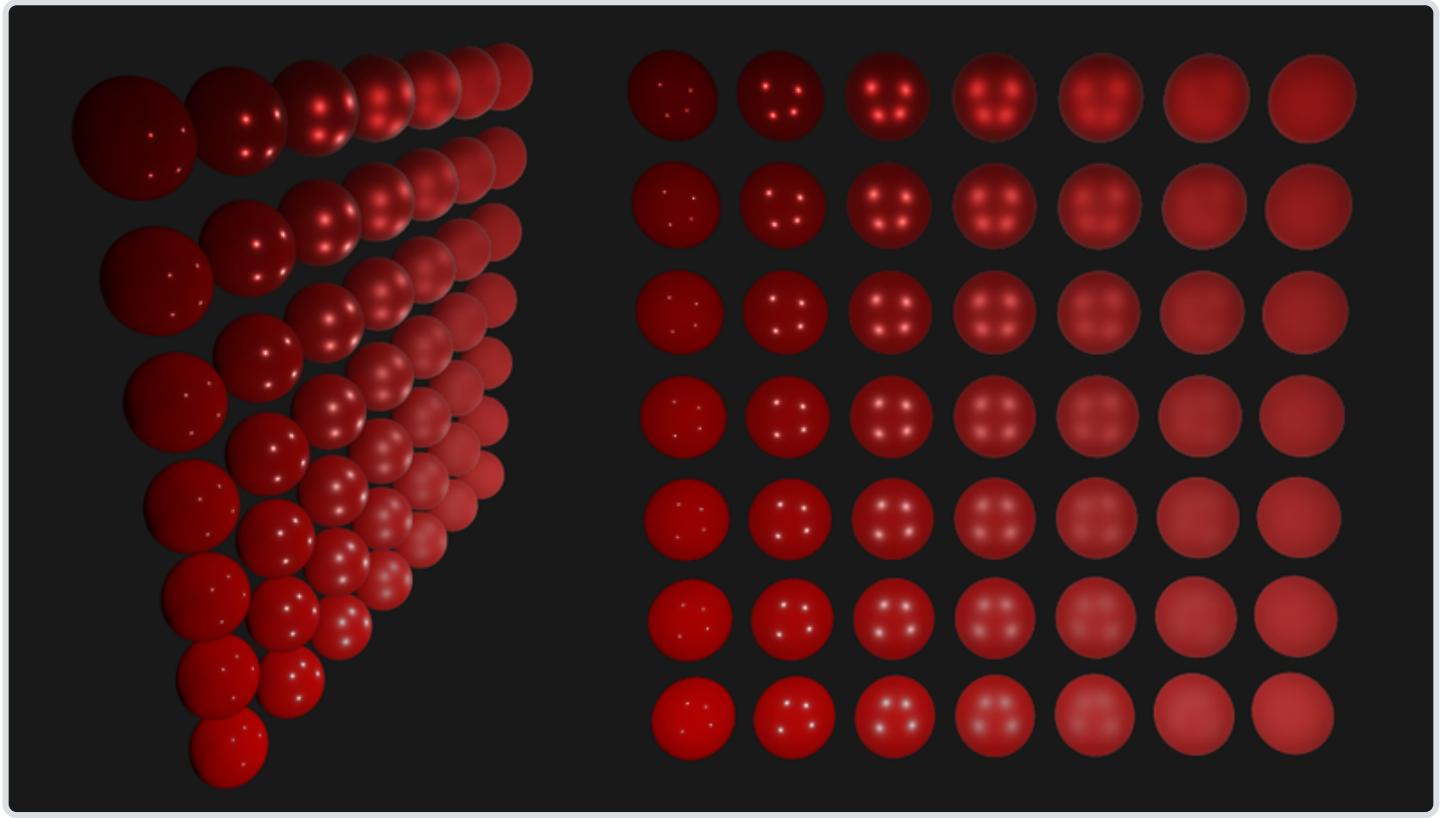
vec3 ambient = vec3(0.03) * albedo * ao;
vec3 color = ambient + Lo;

color = color / (color + vec3(1.0));
color = pow(color, vec3(1.0/2.2));

FragColor = vec4(color, 1.0);
}

```

希望经过上一个教程的[理论知识](#)以及学习过关于渲染方程的一些知识后，这个着色器看起来不会太可怕。如果我们采用这个着色器，加上4个点光源和一些球体，同时我们令这些球体的金属性(metallic)和粗糙度(roughness)沿垂直方向和水平方向分别变化，我们会得到这样的结果：



(上述图片)从下往上球体的金属性从0.0变到1.0， 从左到右球体的粗糙度从0.0变到1.0。你可以看到仅仅改变这两个值，显示的效果会发生巨大的改变！

你可以在[这里](#)找到整个demo的完整代码。

2.7.4 带贴图的PBR

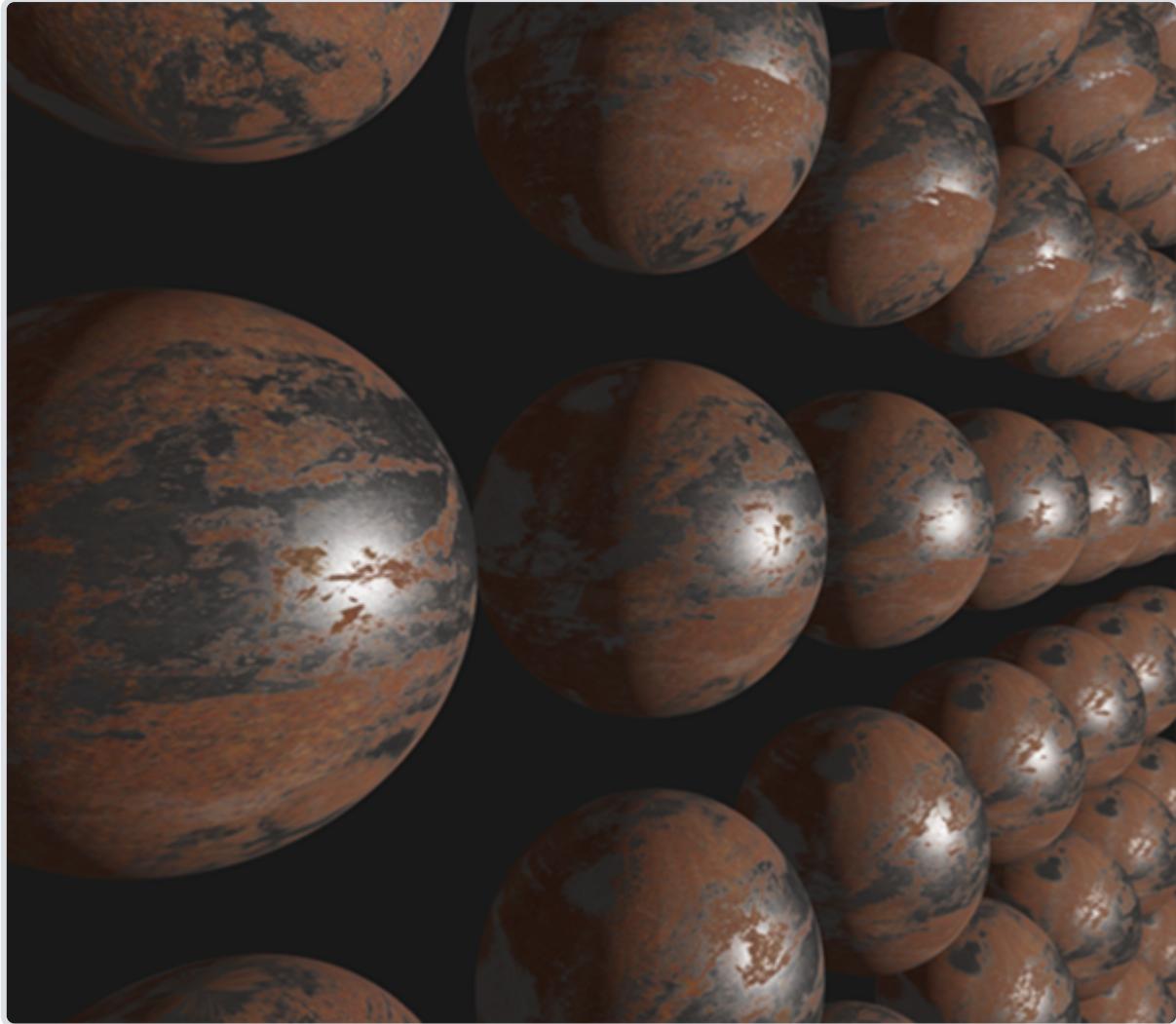
把我们系统扩展成可以接受纹理作为参数可以让我们对物体的材质有更多的自定义空间：

```
[...]
uniform sampler2D albedoMap;
uniform sampler2D normalMap;
uniform sampler2D metallicMap;
uniform sampler2D roughnessMap;
uniform sampler2D aoMap;

void main()
{
    vec3 albedo      = pow(texture(albedoMap, TexCoords).rgb, 2.2);
    vec3 normal      = getNormalFromNormalMap();
    float metallic   = texture(metallicMap, TexCoords).r;
    float roughness  = texture(roughnessMap, TexCoords).r;
    float ao          = texture(aoMap, TexCoords).r;
    [...]
}
```

不过需要注意的一般来说反射率(albedo)纹理在美术人员创建的时候就已经在sRGB空间了，因此我们需要在光照计算之前先把他们转换到线性空间。一般来说，环境光遮蔽贴图(ambient occlusion maps)也需要我们转换到线性空间。不过金属性(Metallic)和粗糙度(Roughness)贴图大多数时间都会保证在线性空间中。

只是把之前的球体的材质性质换成纹理属性，就在视觉上有巨大的提升：



你可以在这里找到纹理贴图过的全部[代码](#)，以及我用的[纹理](#)(记得加上一张全白色的ao Map)。注意金属表面会在场景中看起来有点黑，因为他们没有漫反射。它们会在考虑环境镜面光照的时候看起来更加自然，不过这是我们下一个教程的事情了。

相比起在网上找到的其他PBR渲染结果来说，尽管在视觉上不算是非常震撼，因为我们还没考虑到[基于图片的关照, IBL](#)。我们现在也算是有了一个基于物理的渲染器了(虽然还没考虑IBL)! 你会发现你的光照看起来更加真实了。

译者注：

本章教程有几个小坑原作者没有说清楚，可能是希望读者自己思考，在这译者稍稍提醒一下：

- 首先是球体的生成，主流的球体顶点生成有两种方法，作者源码采用的是UVSphere方法， IcoSphere方法可以参考[这里](#)
- 对于贴图的PBR来说，我们需要TBN矩阵做坐标转换(切线空间-> 世界空间 或者 世界空间 -> 切线空间，参考 法线贴图 章节。)。这有两种方法，一种是在片段着色器中使用叉乘计算TBN矩阵(作者采用的方法)；另外一种是在根据顶点預计算TBN然后VAO中传入TBN矩阵，理论上来说后者会比较快(但是比较麻烦)，不过在译者的实际测试中两者速度差距不大。

2.7.5 IBL

漫反射辐照度

原文 [Diffuse irradiance](#)

作者 JoeyDeVries

翻译 [flyingSnow](#)

校对

基于图像的光照(Image based lighting, IBL)是一类光照技术的集合。其光源不是如[前一节教程](#)中描述的可分解的直接光源，而是将周围环境整体视为一个大光源。IBL 通常使用（取自现实世界或从3D场景生成的）环境立方体贴图(Cubemap)，我们可以将立方体贴图的每个像素视为光源，在渲染方程中直接使用它。这种方式可以有效地捕捉环境的全局光照和氛围，使物体更好地融入其环境。

由于基于图像的光照算法会捕捉部分甚至全部的环境光照，通常认为它是一种更精确的环境光照输入格式，甚至也可以说是一种全局光照的粗略近似。基于此特性，IBL 对 PBR 很有意义，因为当我们把环境光纳入计算之后，物体在物理方面看起来会更加准确。

要开始将 IBL 引入我们的 PBR 系统，让我们再次快速看一下反射方程：

如前所述，我们的主要目标是计算半球上所有入射光方向的积分。解决上一节教程中的积分非常简单，因为我们事先已经知道了对积分有贡献的、若干精确的光线方向。然而这次，来自周围环境的每个方向的入射光都可能具有一些辐射度，使得解决积分变得不那么简单。这为解决积分提出了两个要求：

- 给定任何方向向量，我们需要一些方法来获取这个方向上场景的辐射度。
- 解决积分需要快速且实时。

现在看，第一个要求相对容易些。我们已经有了一些思路：表示环境或场景辐照度的一种方式是（预处理过的）环境立方体贴图，给定这样的立方体贴图，我们可以将立方体贴图的每个纹素视为一个光源。使用一个方向向量对此立方体贴图进行采样，我们就可以获取该方向上的场景辐照度。

如此，给定方向向量，获取此方向上场景辐射度的方法就简化为：

```
vec3 radiance = texture(_cubemapEnvironment, w_i).rgb;
```

为了以更有效的方式解决积分，我们需要对其大部分结果进行预处理——或称**预算算**。为此，我们必须深入研究反射方程：

仔细研究反射方程，我们发现 BRDF 的漫反射 和 镜面 项是相互独立的，我们可以将积分分成两部分：

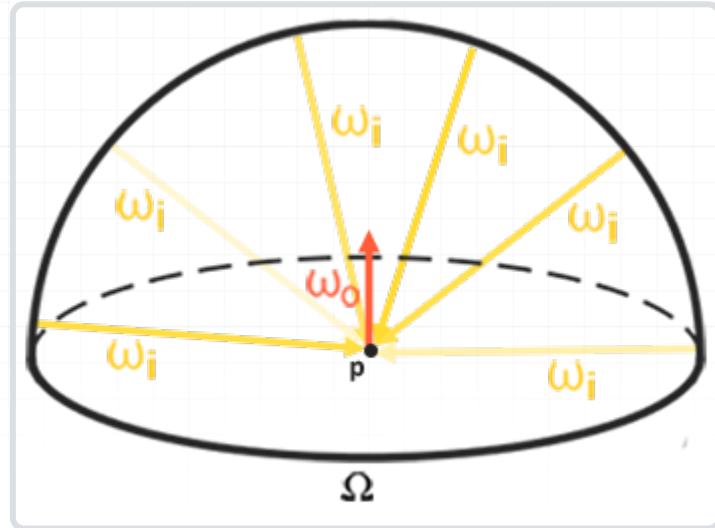
通过将积分分成两部分，我们可以分开研究漫反射和镜面反射部分，本教程的重点是漫反射积分部分。

仔细观察漫反射积分，我们发现漫反射兰伯特项是一个常数项（颜色、折射率 和 在整个积分是常数），不依赖于任何积分变量。基于此，我们可以将常数项移出漫反射积分：

这给了我们一个只依赖于 的积分（假设 位于环境贴图的中心）。有了这些知识，我们就可以计算或预算一个新的立方体贴图，它在每个采样方向——也就是纹素——中存储漫反射积分的结果，这些结果是通过**卷积**计算出来的。

卷积的特性是，对数据集中的一个条目做一些计算时，要考虑到数据集中的所有其他条目。这里的数据集就是场景的辐射度或环境贴图。因此，要对立方体贴图中的每个采样方向做计算，我们都会考虑半球上的所有其他采样方向。

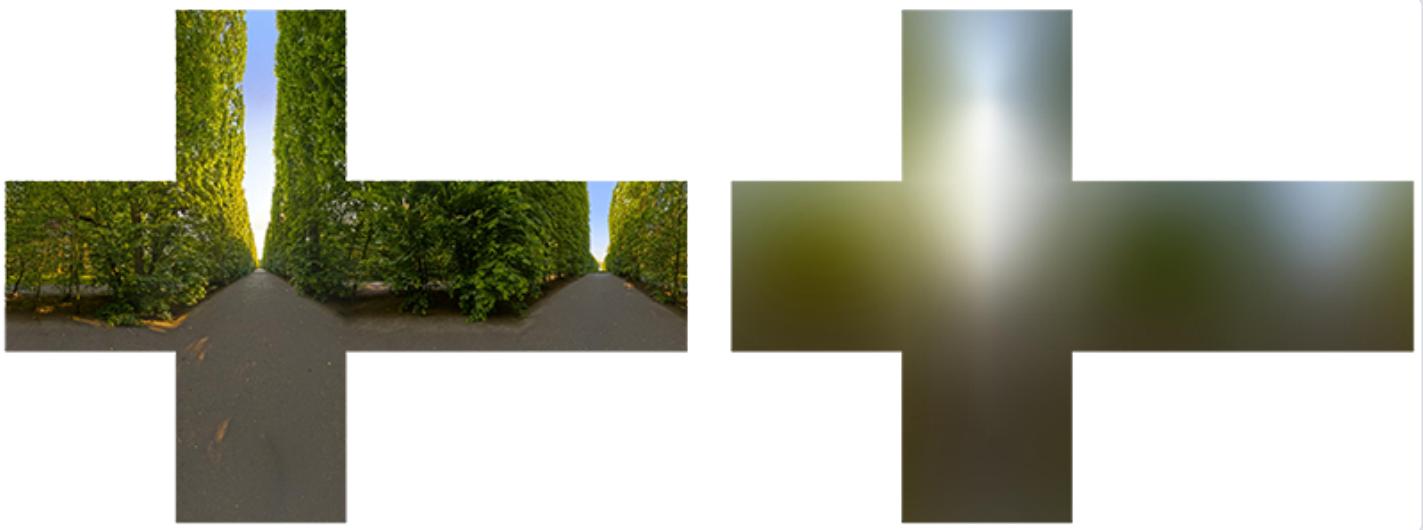
为了对环境贴图进行卷积，我们通过对半球上的大量方向进行离散采样并对其辐射度取平均值，来计算每个输出采样方向的积分。用来采样方向的半球，要面向卷积的输出采样方向。



这个预计算的立方体贴图，在每个采样方向上存储其积分结果，可以理解为场景中所有能够击中面向的表面的间接漫反射光的预计算总和。这样的立方体贴图被称为**辐照度图**，因为经过卷积计算的立方体贴图能让我们从任何方向有效地直接采样场景（预计算好的）辐照度。

辐射方程也依赖了位置，不过这里我们假设它位于辐照度图的中心。这就意味着所有漫反射间接光只能来自同一个环境贴图，这样可能会破坏现实感（特别是在室内）。渲染引擎通过在场景中放置多个反射探针来解决此问题，每个反射探针单独预算其周围环境的辐照度图。这样，位置处的辐照度（以及辐射度）是取离其最近的反射探针之间的辐照度（辐射度）内插值。目前，我们假设总是从中心采样环境贴图，把反射探针的讨论留给后面的教程。

下面是一个环境立方体贴图及其生成的辐照度图的示例（由[Wave 引擎](#)提供），每个方向的场景辐射度取平均值。



由于立方体贴图每个纹素中存储了（方向的）卷积结果，辐照度图看起来有点像环境的平均颜色或光照图。使用任何一个向量对立方体贴图进行采样，就可以获取该方向上的场景辐照度。

PBR 和 HDR

我们在[光照教程](#)中简单提到过：在 PBR 渲染管线中考虑高动态范围(High Dynamic Range, HDR)的场景光照非常重要。由于 PBR 的大部分输入基于实际物理属性和测量，因此为入射光值找到其[物理等效值](#)是很重要的。无论我们是对光线的辐射通量进行研究性猜测，还是使用它们的直接物理等效值，诸如一个简单灯泡和太阳之间的这种差异都是很重要的，如果不在[HDR](#) 渲染环境中工作，就无法正确指定每个光的相对强度。

因此，PBR 和 HDR 需要密切合作，但这些与基于图像的光照有什么关系？我们在之前的教程中已经看到，让 PBR 在 HDR 下工作还比较容易。然而，回想一下基于图像的光照，我们将环境的间接光强度建立在环境立方体贴图的颜色值上，我们需要某种方式将光照的高动态范围存储到环境贴图中。

我们一直使用的环境贴图是以立方体贴图形式储存——如同一个[天空盒](#)——属于低动态范围(Low Dynamic Range, LDR)。我们直接使用各个面的图像的颜色值，其范围介于 0.0 和 1.0 之间，计算过程也是照值处理。这样虽然可能适合视觉输出，但作为物理输入参数，没有什么用处。

辐射度的 HDR 文件格式

谈及辐射度的文件格式，辐射度文件的格式（扩展名为 .hdr）存储了一张完整的立方体贴图，所有六个面数据都是浮点数，允许指定 0.0 到 1.0 范围之外的颜色值，以使光线具有正确的颜色强度。这个文件格式使用了一个聪明的技巧来存储每个浮点值：它并非直接存储每个通道的 32 位数据，而是每个通道存储 8 位，再以 alpha 通道存放指数——虽然确实会导致精度损失，但是非常有效率，不过需要解析程序将每种颜色重新转换为它们的浮点数等效值。

[sIBL 档案](#) 中有很多可以免费获取的辐射度 HDR 环境贴图，下面是一个示例：



可能与您期望的完全不同，因为图像非常扭曲，并且没有我们之前看到的环境贴图的六个立方体贴图面。这张环境贴图是从球体投影到平面上，以使我们可以轻松地将环境信息存储到一张[等距柱状投影图](#)(Equirectangular Map) 中。有一点确实需要说明：水平视角附近分辨率较高，而底部和顶部方向分辨率较低，在大多数情况下，这是一个不错的折衷方案，因为对于几乎所有渲染器来说，大部分有意义的光照和环境信息都在水平视角附近方向。

HDR 和 stb_image.h

直接加载辐射度 HDR 图像需要一些[文件格式](#)的知识，虽然不是很困难，但仍然很麻烦。幸运的是，一个常用的头文件库 [stb_image.h](#) 支持将辐射度 HDR 图像直接加载为一个浮点数数组，完全符合我们的需要。将 stb_image 添加到项目中之后，加载HDR图像非常简单，如下：

```
#include "stb_image.h"
[...]

stbi_set_flip_vertically_on_load(true);
int width, height, nrComponents;
float *data = stbi_loadf("newport_loft.hdr", &width, &height, &nrComponents, 0);
unsigned int hdrTexture;
if (data)
{
    glGenTextures(1, &hdrTexture);
    glBindTexture(GL_TEXTURE_2D, hdrTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, width, height, 0, GL_RGB, GL_FLOAT, data);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    stbi_image_free(data);
}
else
{
    std::cout << "Failed to load HDR image." << std::endl;
}
```

stb_image.h 自动将 HDR 值映射到一个浮点数列表：默认情况下，每个通道32位，每个颜色 3 个通道。我们要将等距柱状投影 HDR 环境贴图转存到 2D 浮点纹理中，这就是所要做的全部工作。

从等距柱状投影到立方体贴图

当然也可以直接使用等距柱状投影图获取环境信息，但是这些操作还是显得相对昂贵，在这种情况下，直接采样立方体贴图的性能更高。因此，在本教程中，我们首先将等距柱状投影图转换为立方体贴图以备进一步处理。请注意，在此过程中，我们还将展示如何对等距柱状格式的投影图采样，如同采样 3D 环境贴图一样，您可以自由选择您喜欢的任何解决方案。

要将等距柱状投影图转换为立方体贴图，我们需要渲染一个（单位）立方体，并从内部将等距柱状图投影到立方体的每个面，并将立方体的六个面的图像构造成立方体贴图。此立方体的顶点着色器只是按原样渲染立方体，并将其局部坐标作为 3D 采样向量传递给片段着色器：

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 localPos;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    localPos = aPos;
    gl_Position = projection * view * vec4(localPos, 1.0);
}
```

而在片段着色器中，我们为立方体的每个部分着色，方法类似于将等距柱状投影图整齐地折叠到立方体的每个面一样。为了实现这一点，我们先获取片段的采样方向，这个方向是从立方体的局部坐标进行插值得到的，然后使用此方向向量和一些三角学魔法对等距柱状投影图进行采样，如同立方体图本身一样。我们直接将结果存储到立方体每个面的片段中，以下就是我们需要做的：

```
#version 330 core
out vec4 FragColor;
in vec3 localPos;

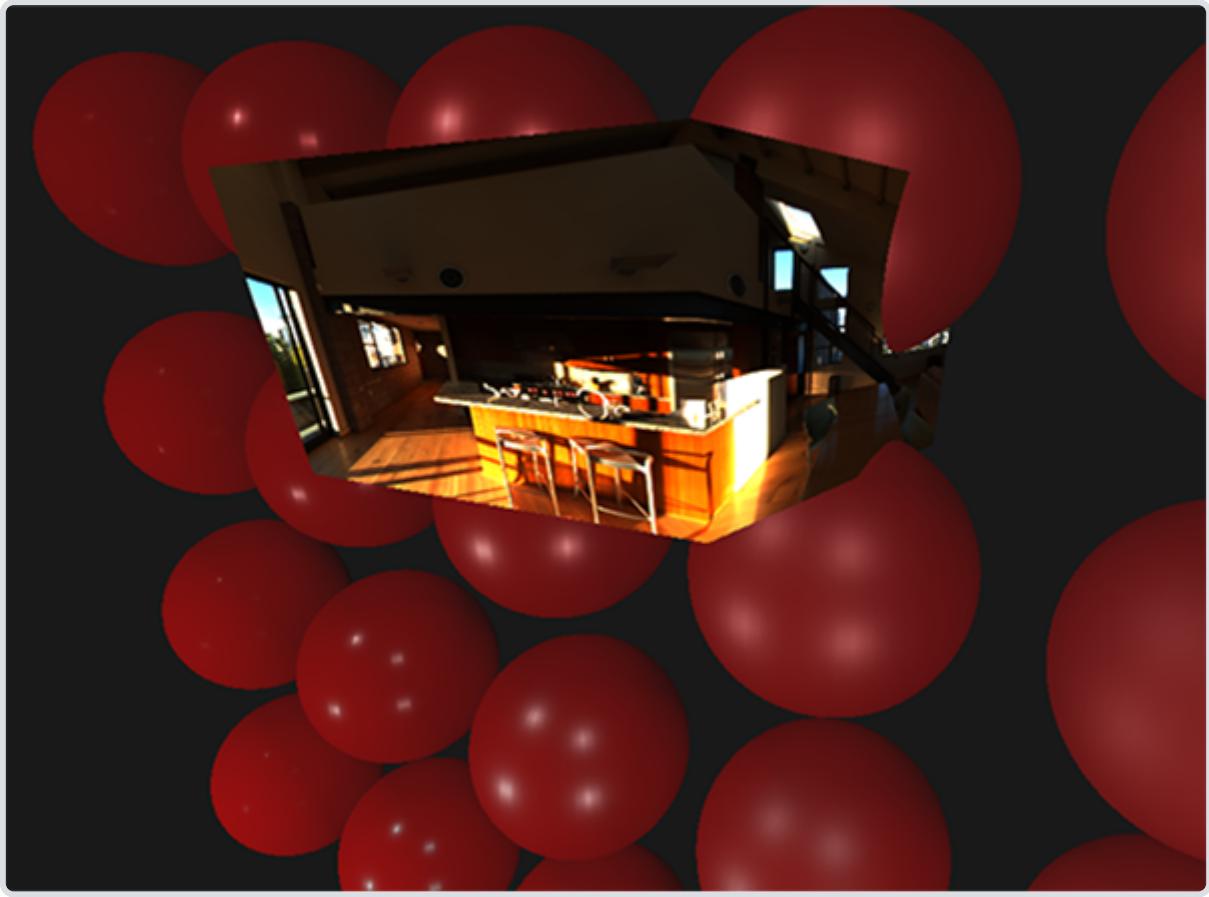
uniform sampler2D equirectangularMap;

const vec2 invAtan = vec2(0.1591, 0.3183);
vec2 SampleSphericalMap(vec3 v)
{
    vec2 uv = vec2(atan(v.z, v.x), asin(v.y));
    uv *= invAtan;
    uv += 0.5;
    return uv;
}

void main()
{
    vec2 uv = SampleSphericalMap(normalize(localPos)); // make sure to normalize localPos
    vec3 color = texture(equirectangularMap, uv).rgb;

    FragColor = vec4(color, 1.0);
}
```

如果给定HDR等距柱状投影图，在场景的中心渲染一个立方体，将得到如下所示的内容：



这表明我们有效地将等距柱状投影图映射到了立方体，但我们还需要将源HDR图像转换为立方体贴图纹理。为了实现这一点，我们必须对同一个立方体渲染六次，每次面对立方体的一个面，并用[帧缓冲](#)对象记录其结果：

```
unsigned int captureFB0, captureRB0;
 glGenFramebuffers(1, &captureFB0);
 glGenRenderbuffers(1, &captureRB0);

 glBindFramebuffer(GL_FRAMEBUFFER, captureFB0);
 glBindRenderbuffer(GL_RENDERBUFFER, captureRB0);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, 512, 512);
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, captureRB0);
```

当然，我们此时就可以生成相应的立方体贴图了，首先为其六个面预先分配内存：

```
unsigned int envCubemap;
 glGenTextures(1, &envCubemap);
 glBindTexture(GL_TEXTURE_CUBE_MAP, envCubemap);
 for (unsigned int i = 0; i < 6; ++i)
 {
    // note that we store each face with 16 bit floating point values
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB16F,
                 512, 512, 0, GL_RGB, GL_FLOAT, nullptr);
 }

 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

那剩下要做的就是将等距柱状 2D 纹理捕捉到立方体贴图的面上。

之前在[帧缓冲和点阴影](#)教程中讨论过的代码细节，我就不再次详细说明，实际过程可以概括为：面向立方体六个面设置六个不同的视图矩阵，给定投影矩阵的 fov 为 90 度以捕捉整个面，并渲染立方体六次，将结果存储在浮点帧缓冲中：

```
glm::mat4 captureProjection = glm::perspective(glm::radians(90.0f), 1.0f, 0.1f, 10.0f);
glm::mat4 captureViews[] =
{
    glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3( 1.0f, 0.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
    glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(-1.0f, 0.0f, 0.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
    glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3( 0.0f, 1.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f)),
    glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3( 0.0f, -1.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f)),
    glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3( 0.0f, 0.0f, 1.0f), glm::vec3(0.0f, -1.0f, 0.0f)),
    glm::lookAt(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3( 0.0f, 0.0f, -1.0f), glm::vec3(0.0f, 1.0f, 0.0f))
};

// convert HDR equirectangular environment map to cubemap equivalent
equirectangularToCubemapShader.use();
equirectangularToCubemapShader.setInt("equirectangularMap", 0);
equirectangularToCubemapShader.setMat4("projection", captureProjection);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, hdrTexture);

glViewport(0, 0, 512, 512); // don't forget to configure the viewport to the capture dimensions.
glBindFramebuffer(GL_FRAMEBUFFER, captureFB0);
for (unsigned int i = 0; i < 6; ++i)
{
    equirectangularToCubemapShader.setMat4("view", captureViews[i]);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                          GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, envCubemap, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    renderCube(); // renders a 1x1 cube
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

我们采用帧缓冲的颜色值并围绕立方体贴图的每个面切换纹理目标，直接将场景渲染到立方体贴图的一个面上。一旦这个流程完毕——我们只需做一次——立方体贴图 `envCubemap` 就应该是原 HDR 图的环境立方体贴图版。

让我们编写一个非常简单的天空盒着色器来测试立方体贴图，用来显示周围的立方体贴图：

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 projection;
uniform mat4 view;

out vec3 localPos;

void main()
{
    localPos = aPos;

    mat4 rotView = mat4(mat3(view)); // remove translation from the view matrix
    vec4 clipPos = projection * rotView * vec4(localPos, 1.0);

    gl_Position = clipPos.xyww;
}
```

注意这里的小技巧 `xyww` 可以确保渲染的立方体片段的深度值总是 1.0，即最大深度，如[立方体贴图](#)教程中所述。注意我们需要将深度比较函数更改为 `GL_EQUAL`：

```
glDepthFunc(GL_EQUAL);
```

这个片段着色器直接使用立方体的片段局部坐标，对环境立方体贴图采样：

```
#version 330 core
out vec4 FragColor;

in vec3 localPos;

uniform samplerCube environmentMap;

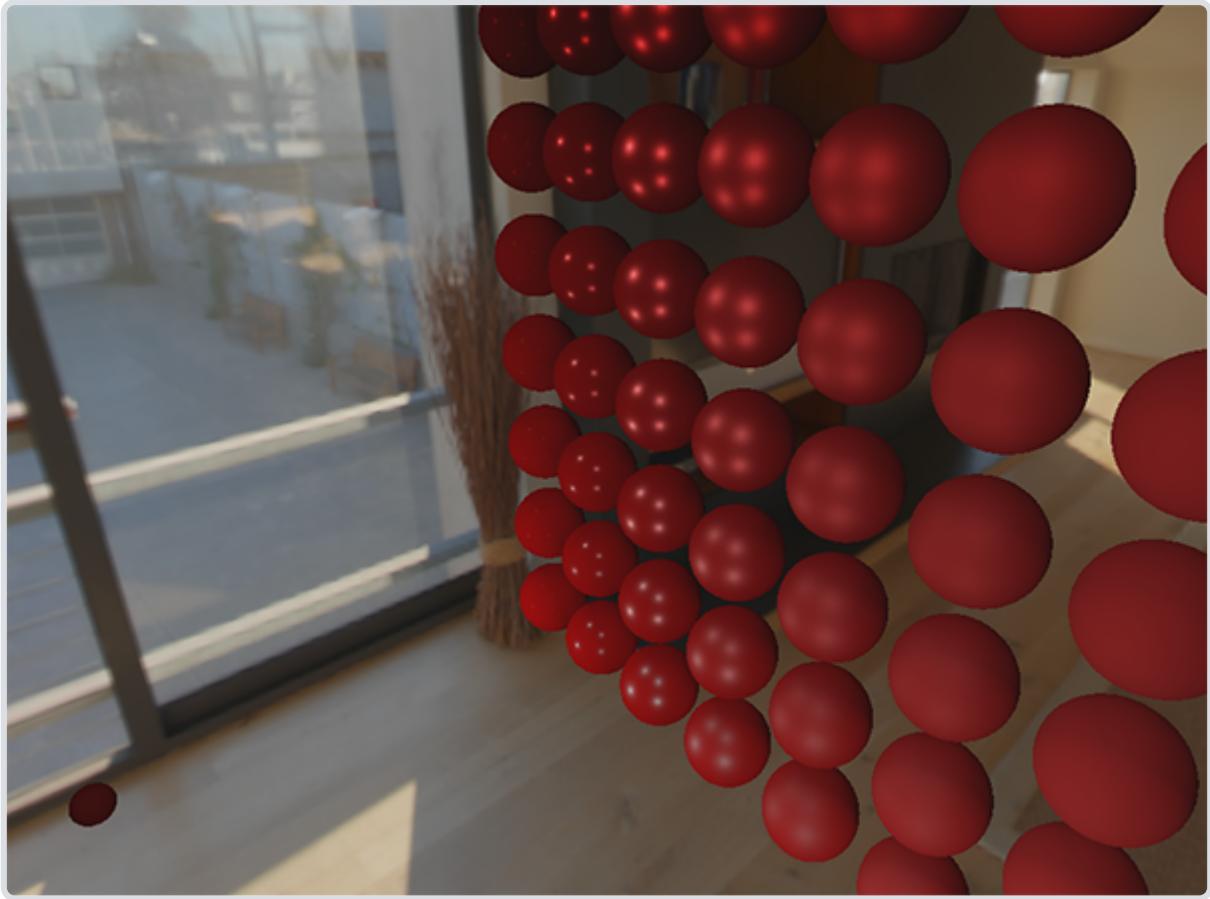
void main()
{
    vec3 envColor = texture(environmentMap, localPos).rgb;

    envColor = envColor / (envColor + vec3(1.0));
    envColor = pow(envColor, vec3(1.0/2.2));

    FragColor = vec4(envColor, 1.0);
}
```

我们使用插值的立方体顶点坐标对环境贴图进行采样，这些坐标直接对应于正确的采样方向向量。注意，相机的平移分量被忽略掉了，在立方体上渲染此着色器会得到非移动状态下的环境贴图。另外还请注意，当我们将环境贴图的 HDR 值直接输出到默认的 LDR 帧缓冲时，希望对颜色值进行正确的色调映射。此外，默认情况下，几乎所有 HDR 图都处于线性颜色空间中，因此我们需要在写入默认帧缓冲之前应用[伽马校正](#)。

现在，在之前渲染的球体上渲染环境贴图，效果应该如下图：



好的...我们用了相当多的设置终于来到了这里，我们设法成功地读取了 HDR 环境贴图，将它从等距柱状投影图转换为立方体贴图，并将 HDR 立方体贴图作为天空盒渲染到了场景中。此外，我们设置了一个小系统来渲染立方体贴图的所有六个面，我们在计算环境贴图卷积时还会需要它。您可以在[此处](#)找到整个转化过程的源代码。

立方体贴图的卷积

如本节教程开头所述，我们的主要目标是计算所有间接漫反射光的积分，其中光照的辐照度以环境立方体贴图的形式给出。我们已经知道，在方向上采样 HDR 环境贴图，可以获得场景在此方向上的辐射度。虽然如此，要解决积分，我们仍然不能仅从一个方向对环境贴图采样，而要从半球上所有可能的方向进行采样，这对于片段着色器而言还是过于昂贵。

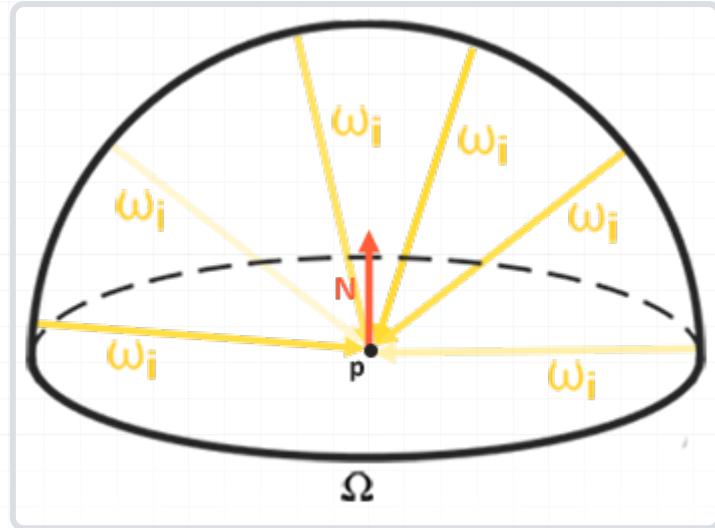
然而，计算上又不可能从 的每个可能的方向采样环境光照，理论上可能的方向数量是无限的。不过我们可以对有限数量的方向采样以近似求解，在半球内均匀间隔或随机取方向可以获得一个相当精确的辐照度近似值，从而离散地计算积分。

然而，对于每个片段实时执行此操作仍然太昂贵，因为仍然需要非常大的样本数量才能获得不错的结果，因此我们希望可以**预算**。既然半球的朝向决定了我们捕捉辐照度的位置，我们可以预先计算每个可能的半球朝向的辐照度，这些半球朝向涵盖了所有可能的出射方向：

给定任何方向向量，我们可以对预算的辐照度图采样以获取方向的总漫反射辐照度。为了确定片段上间接漫反射光的数量（辐照度），我们获取以表面法线为中心的半球的总辐照度。获取场景辐照度的方法就简化为：

```
vec3 irradiance = texture(irradianceMap, N);
```

现在，为了生成辐照度贴图，我们需要将环境光照求卷积，转换为立方体贴图。假设对于每个片段，表面的半球朝向法向量，对立方体贴图进行卷积等于计算朝向的半球中每个方向的总平均辐射率。



值得庆幸的是，本节教程中所有繁琐的设置并非毫无用处，因为我们现在可以直接获取转换后的立方体贴图，在片段着色器中对其进行卷积，渲染所有六个面，将其结果用帧缓冲捕捉到新的立方体贴图中。之前已经将等距柱状投影图转换为立方体贴图，这次我们可以采用完全相同的方法，但使用不同的片段着色器：

```
#version 330 core
out vec4 FragColor;
in vec3 localPos;

uniform samplerCube environmentMap;

const float PI = 3.14159265359;

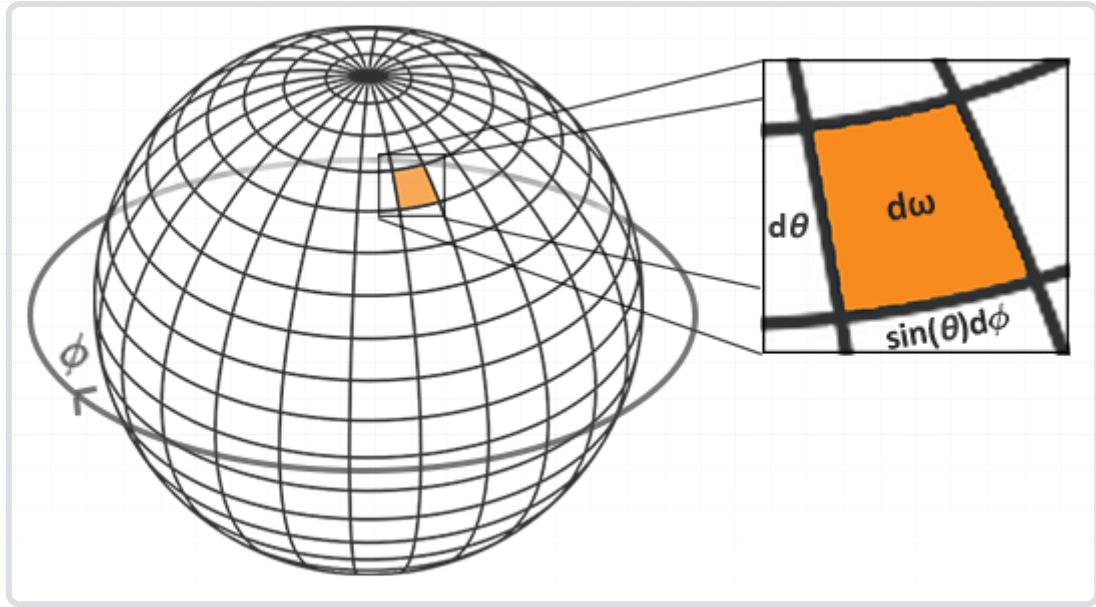
void main()
{
    // the sample direction equals the hemisphere's orientation
    vec3 normal = normalize(localPos);

    vec3 irradiance = vec3(0.0);

    [...] // convolution code

    FragColor = vec4(irradiance, 1.0);
}
```

`environmentMap` 是从等距柱状投影图转换而来的 HDR 立方体贴图。有很多方法可以对环境贴图进行卷积，但是对于本教程，我们的方法是：对于立方体贴图的每个纹素，在纹素所代表的方向的半球 内生成固定数量的采样向量，并对采样结果取平均值。数量固定的采样向量将均匀地分布在半球内部。注意，积分是连续函数，在采样向量数量固定的情况下离散地采样只是一种近似计算方法，我们采样的向量越多，就越接近正确的结果。反射方程的积分 是围绕立体角 旋转，而这个立体角相当难以处理。为了避免对难处理的立体角求积分，我们使用球坐标 和 来代替立体角。



对于围绕半球大圆的航向角，我们在到内采样，而从半球顶点出发的倾斜角，采样范围是到。于是我们更新一下反射积分方程：

求解积分需要我们在半球内采集固定数量的离散样本并对其结果求平均值。分别给每个球坐标轴指定离散样本数量 和以求其黎曼和，积分式会转换为以下离散版本：

当我们离散地对两个球坐标轴进行采样时，每个采样近似代表了半球上的一小块区域，如上图所示。注意，由于球的一般性质，当采样区域朝向中心顶部会聚时，天顶角变高，半球的离散采样区域变小。为了平衡较小的区域贡献度，我们使用来权衡区域贡献度，这就是多出来的的作用。

给定每个片段的积分球坐标，对半球进行离散采样，过程代码如下：

```

vec3 irradiance = vec3(0.0);

vec3 up      = vec3(0.0, 1.0, 0.0);
vec3 right   = cross(up, normal);
up        = cross(normal, right);

float sampleDelta = 0.025;
float nrSamples = 0.0;
for(float phi = 0.0; phi < 2.0 * PI; phi += sampleDelta)
{
    for(float theta = 0.0; theta < 0.5 * PI; theta += sampleDelta)
    {
        // spherical to cartesian (in tangent space)
        vec3 tangentSample = vec3(sin(theta) * cos(phi), sin(theta) * sin(phi), cos(theta));
        // tangent space to world
        vec3 sampleVec = tangentSample.x * right + tangentSample.y * up + tangentSample.z * N;

        irradiance += texture(environmentMap, sampleVec).rgb * cos(theta) * sin(theta);
        nrSamples++;
    }
}
irradiance = PI * irradiance * (1.0 / float(nrSamples));

```

我们以一个固定的 `sampleDelta` 增量值遍历半球，减小（或增加）这个增量将会增加（或减少）精确度。

在两层循环内，我们获取一个球面坐标并将它们转换为 3D 直角坐标向量，将向量从切线空间转换为世界空间，并使用此向量直接采样 HDR 环境贴图。我们将每个采样结果加到 `irradiance`，最后除以采样的总数，得到平均采样辐照度。请注意，我们将采样的颜色值乘以系数 $\cos(\theta)$ ，因为较大角度的光较弱，而系数 $\sin(\theta)$ 则用于权衡较高半球区域的较小采样区域的贡献度。

现在剩下要做的就是设置 OpenGL 渲染代码，以便我们可以对之前捕捉的 `envCubemap` 求卷积。首先我们创建一个辐照度立方体贴图（重复一遍，我们只需要在渲染循环之前执行一次）：

```
unsigned int irradianceMap;
 glGenTextures(1, &irradianceMap);
 glBindTexture(GL_TEXTURE_CUBE_MAP, irradianceMap);
 for (unsigned int i = 0; i < 6; ++i)
 {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB16F, 32, 32, 0,
                 GL_RGB, GL_FLOAT, nullptr);
 }
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

由于辐照度图对所有周围的辐射值取了平均值，因此它丢失了大部分高频细节，所以我们可以以较低的分辨率（32x32）存储，并让 OpenGL 的线性滤波完成大部分工作。接下来，我们将捕捉到的帧缓冲图像缩放到新的分辨率：

```
glBindFramebuffer(GL_FRAMEBUFFER, captureFB0);
 glBindRenderbuffer(GL_RENDERBUFFER, captureRB0);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, 32, 32);
```

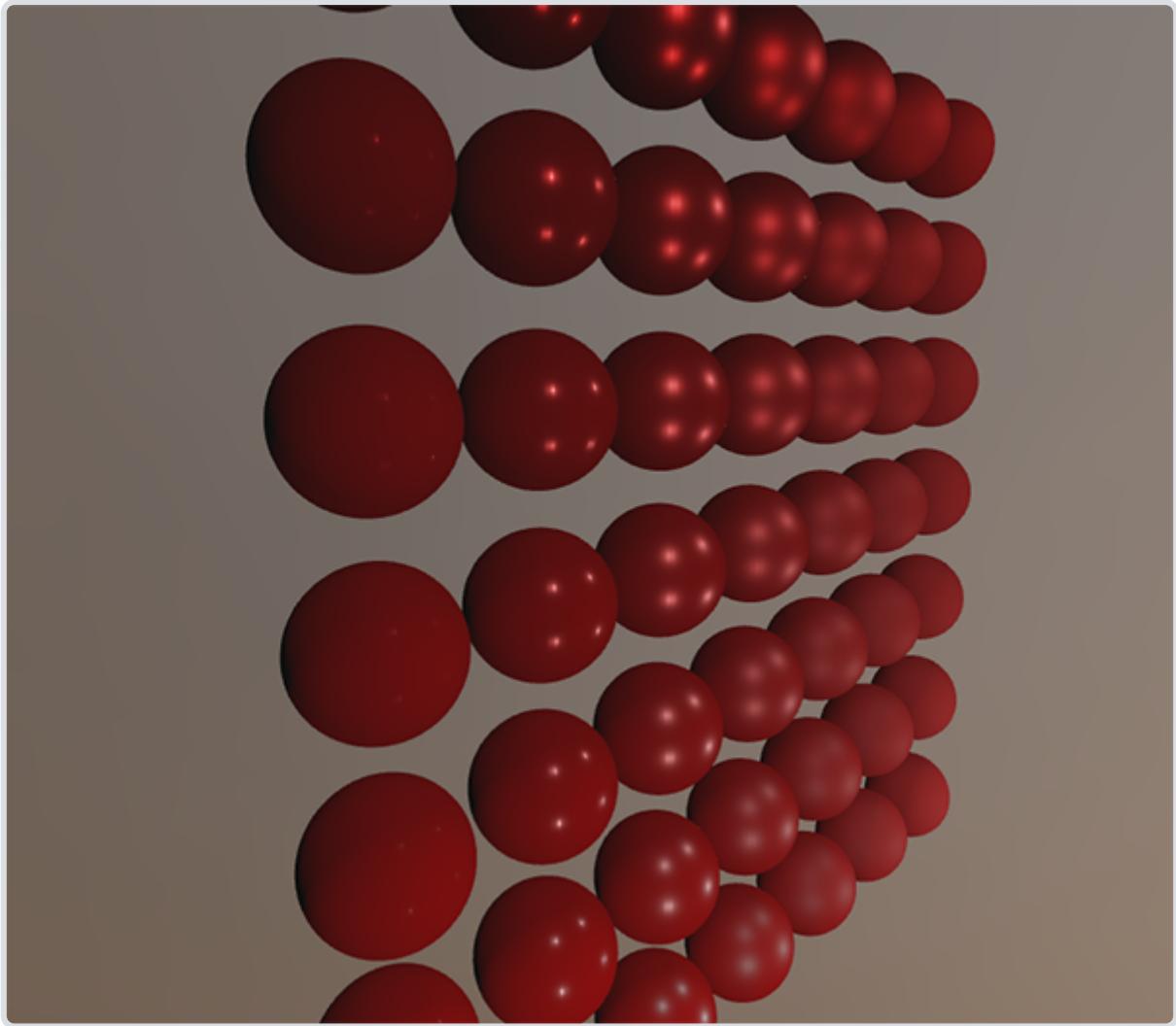
我们使用卷积着色器——和捕捉环境立方体贴图类似的方式——来对环境贴图求卷积：

```
irradianceShader.use();
irradianceShader.setInt("environmentMap", 0);
irradianceShader.setMat4("projection", captureProjection);
glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_CUBE_MAP, envCubemap);

glViewport(0, 0, 32, 32); // don't forget to configure the viewport to the capture dimensions.
glBindFramebuffer(GL_FRAMEBUFFER, captureFB0);
for (unsigned int i = 0; i < 6; ++i)
{
    irradianceShader.setMat4("view", captureViews[i]);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                          GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, irradianceMap, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    renderCube();
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

现在，完成这个流程之后，我们应该得到了一个预算算好的辐照度图，可以直接将其用于IBL 计算。为了查看我们是否成功地对环境贴图进行了卷积，让我们将天空盒的环境采样贴图替换为辐照度贴图：



如果它看起来像模糊的环境贴图，说明您已经成功地对环境贴图进行了卷积。

PBR 和间接辐照度光照

辐照度图表示所有周围的间接光累积的反射率的漫反射部分的积分。注意光不是来自任何直接光源，而是来自周围环境，我们将间接漫反射和间接镜面反射视为环境光，取代了我们之前设定的常数项。

首先，务必将预算算的辐照度图添加为一个立方体采样器：

```
uniform samplerCube irradianceMap;
```

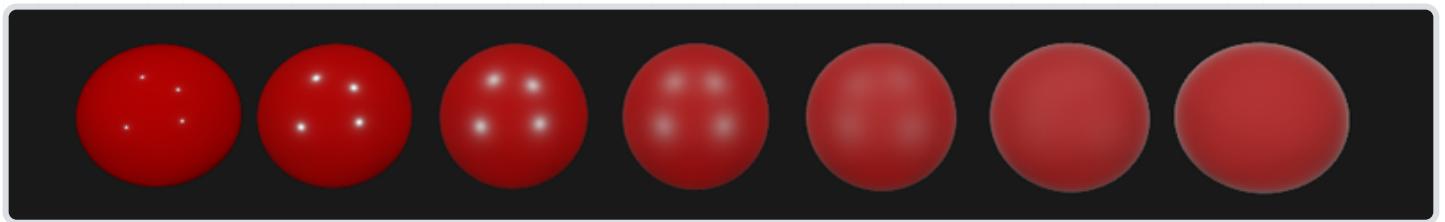
给定一张辐照度图，它存储了场景中的所有间接漫反射光，获取片段的辐照度就简化为给定法线的一次纹理采样：

```
// vec3 ambient = vec3(0.03);
vec3 ambient = texture(irradianceMap, N).rgb;
```

然而，由于间接光照包括漫反射和镜面反射两部分，正如我们从分割版的反射方程中看到的那样，我们需要对漫反射部分进行相应的加权。与我们在前一节教程中所做的类似，我们使用菲涅耳公式来计算表面的间接反射率，我们从中得出折射率或称漫反射率：

```
vec3 kS = fresnelSchlick(max(dot(N, V), 0.0), F0);
vec3 kD = 1.0 - kS;
vec3 irradiance = texture(irradianceMap, N).rgb;
vec3 diffuse    = irradiance * albedo;
vec3 ambient    = (kD * diffuse) * ao;
```

由于环境光来自半球内围绕法线 N 的所有方向，因此没有一个确定的半向量来计算菲涅耳效应。为了模拟菲涅耳效应，我们用法线和视线之间的夹角计算菲涅耳系数。然而，之前我们是以受粗糙度影响的微表面半向量作为菲涅耳公式的输入，但我们目前没有考虑任何粗糙度，表面的反射率总是会相对较高。间接光和直射光遵循相同的属性，因此我们期望较粗糙的表面在边缘反射较弱。由于我们没有考虑表面的粗糙度，间接菲涅耳反射在粗糙非金属表面上看起来有点过强（为了演示目的略微夸大）：



我们可以通过在 [Sébastien Lagarde](#) 提出的 Fresnel-Schlick 方程中加入粗糙度项来缓解这个问题：

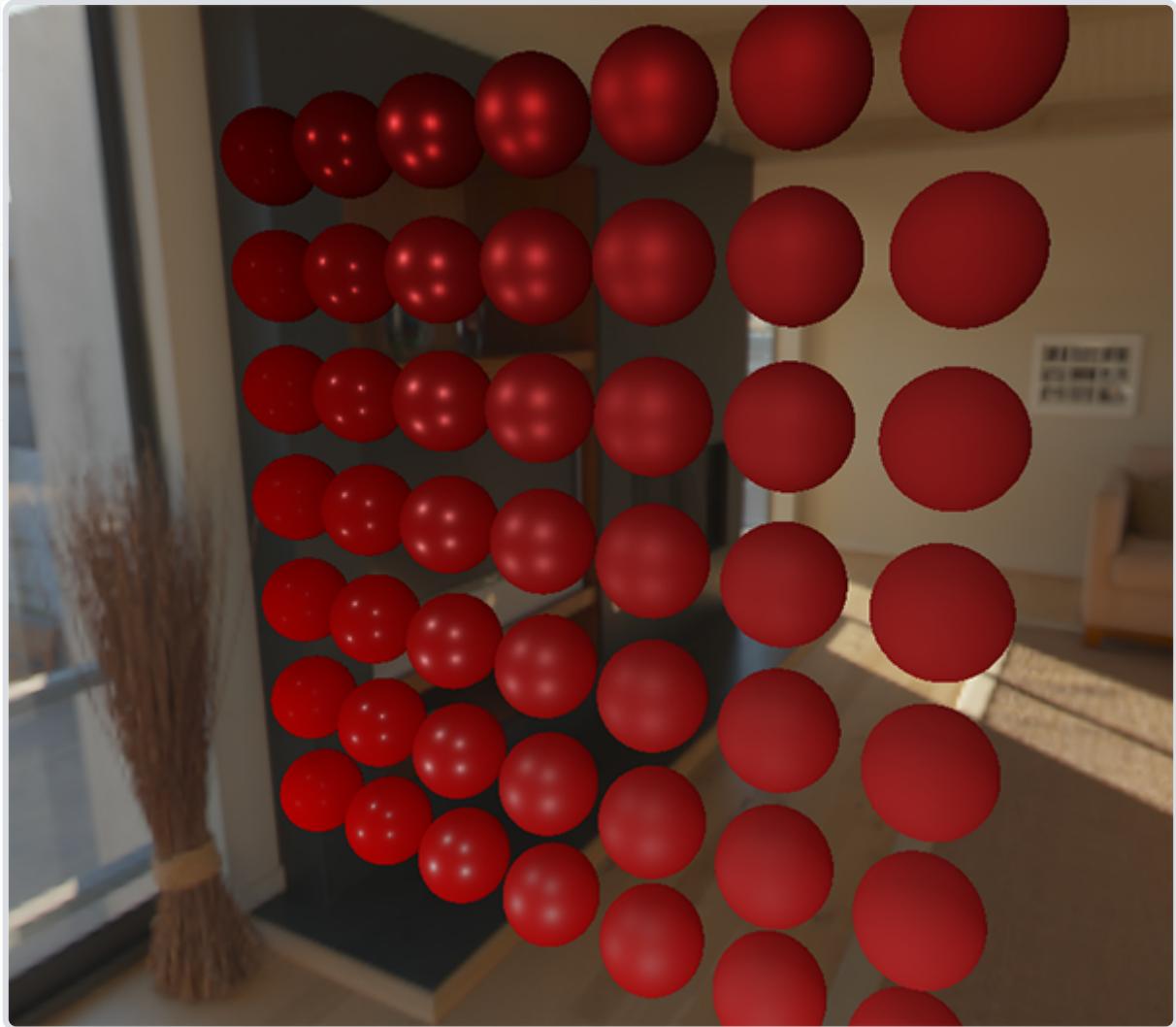
```
vec3 fresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness)
{
    return F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(1.0 - cosTheta, 5.0);
}
```

在计算菲涅耳效应时纳入表面粗糙度，环境光代码最终确定为：

```
vec3 kS = fresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);
vec3 kD = 1.0 - kS;
vec3 irradiance = texture(irradianceMap, N).rgb;
vec3 diffuse    = irradiance * albedo;
vec3 ambient    = (kD * diffuse) * ao;
```

如您所见，实践上基于图像的光照计算非常简单，只需要采样一次立方体贴图，大部分的工作量在于将环境贴图预算算或卷积成辐照度图。

回到我们在[光照教程](#)中建立的初始场景，场景中排列的球体金属度沿垂直方向递增，粗糙度沿水平方向递增。向场景中添加基于漫反射图像的光照之后，它看起来像这样：



现在看起来仍然有点奇怪，因为金属度较高的球体需要某种形式的反射以便看起来更像金属表面（因为金属表面没有漫反射），不过目前只有来自点光源的反射——而且可以说几乎没有。不过尽管如此，您也可以看出，球体在环境中的感觉更加和谐了（特别是在环境贴图之间切换的时候），因为表面会正确地响应环境光照。

您可以在[此处](#)找到以上讨论过的整套源代码。在[下一节教程](#)中，我们将添加反射积分的间接镜面反射部分，此时我们将真正看到 PBR 的力量。

进阶阅读

- [Coding Labs: Physically based rendering](#): 介绍 PBR，如何以及为何要生成辐照度图。
- [The Mathematics of Shading](#): 借助 ScratchAPixel，对本教程中涉及的一些数学知识的简要介绍，特别是关于极坐标和积分。

镜面反射 IBL

原文 [Specular IBL](#)

作者 JoeyDeVries

翻译 [flyingSnow](#)

校对

在[上一节教程](#)中，我们预算算了辐照度图作为光照的间接漫反射部分，以将 PBR 与基于图像的照明相结合。在本教程中，我们将重点关注反射方程的镜面部分：

你会注意到 Cook-Torrance 镜面部分（乘以）在整个积分上不是常数，不仅受入射光方向影响，还受视角影响。如果试图解算所有入射光方向加所有可能的视角方向的积分，二者组合数会极其庞大，实时计算太昂贵。Epic Games 提出了一个解决方案，他们预算算镜面部分的卷积，为实时计算作了一些妥协，这种方案被称为[分割求和近似法](#)（split sum approximation）。分割求和近似将方程的镜面部分分割成两个独立的部分，我们可以单独求卷积，然后在 PBR 着色器中求和，以用于间接镜面反射部分 IBL。分割求和近似法类似于我们之前求辐照图预卷积的方法，需要 HDR 环境贴图作为其卷积输入。为了理解，我们回顾一下反射方程，但这次只关注镜面反射部分（在[上一节教程](#)中已经剥离了漫反射部分）：

由于与辐照度卷积相同的（性能）原因，我们无法以合理的性能实时求解积分的镜面反射部分。因此，我们最好预算算这个积分，以得到像镜面 IBL 贴图这样的东西，用片段的法线对这张图采样并计算。但是，有一个地方有点棘手：我们能够预算算辐照度图，是因为其积分仅依赖于，并且可以将漫反射反射率常数项移出积分，但这一次，积分不仅仅取决于，从 BRDF 可以看出：

这次积分还依赖，我们无法用两个方向向量采样预算算的立方体图。如前一个教程中所述，位置与此处无关。在实时状态下，对每种可能的和的组合预算算该积分是不可行的。Epic Games 的分割求和近似法将预算算分成两个单独的部分求解，再将两部分组合起来得到后文给出的预算算结果。分割求和近似法将镜面反射积分拆成两个独立的积分：

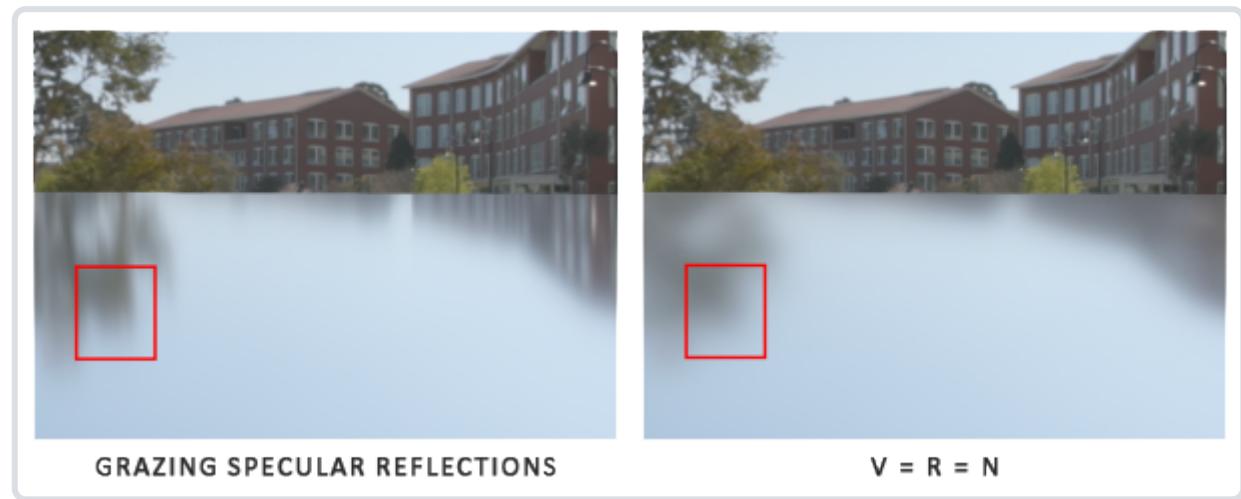
卷积的第一部分被称为[预滤波环境贴图](#)，它类似于辐照度图，是预先计算的环境卷积贴图，但这次考虑了粗糙度。因为随着粗糙度的增加，参与环境贴图卷积的采样向量会更分散，导致反射更模糊，所以对于卷积的每个粗糙度级别，我们将按顺序把模糊后的结果存储在预滤波贴图的 mipmap 中。例如，预过滤的环境贴图在其 5 个 mipmap 级别中存储 5 个不同粗糙度值的预卷积结果，如下图所示：



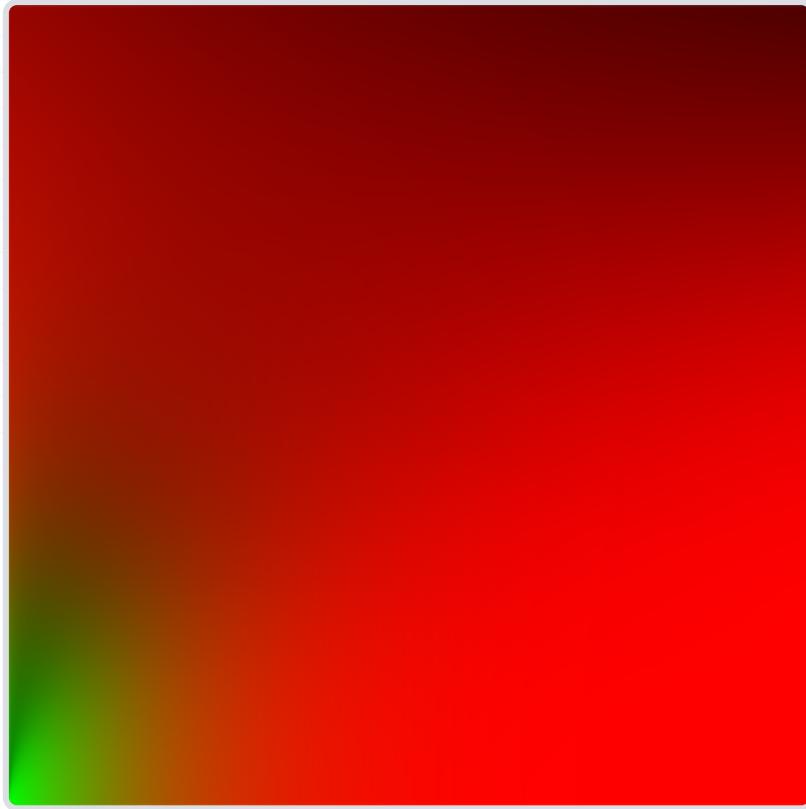
我们使用 Cook-Torrance BRDF 的法线分布函数(NDF)生成采样向量及其散射强度，该函数将法线和视角方向作为输入。由于我们在卷积环境贴图时事先不知道视角方向，因此 Epic Games 假设视角方向——也就是镜面反射方向——总是等于输出采样方向，以作进一步近似。翻译成代码如下：

```
vec3 N = normalize(w_o);
vec3 R = N;
vec3 V = R;
```

这样，预过滤的环境卷积就不需要关心视角方向了。这意味着当从如下图的角度观察表面的镜面反射时，得到的掠角镜面反射效果不是很好（图片来自文章《Moving Frostbite to PBR》）。然而，通常可以认为这是一个体面的妥协：



等式的第二部分等于镜面反射积分的 BRDF 部分。如果我们假设每个方向的入射辐射度都是白色的（因此），就可以在给定粗糙度、光线 法线 夹角 的情况下，预算算 BRDF 的响应结果。Epic Games 将预算算好的 BRDF 对每个粗糙度和入射角的组合的响应结果存储在一张 2D 查找纹理(LUT)上，称为**BRDF积分贴图**。2D 查找纹理存储是菲涅耳响应的系数 (R 通道) 和偏差值 (G 通道)，它为我们提供了分割版镜面反射积分的第二个部分：



生成查找纹理的时候，我们以 BRDF 的输入（范围在 0.0 和 1.0 之间）作为横坐标，以粗糙度作为纵坐标。有了此 BRDF 积分贴图和预过滤的环境贴图，我们就可以将两者结合起来，以获得镜面反射积分的结果：

```
float lod = getMipLevelFromRoughness(roughness);
vec3 prefilteredColor = textureCubeLod(PrefilteredEnvMap, refVec, lod);
vec2 envBRDF = texture2D(BRDFIntegrationMap, vec2(NdotV, roughness)).xy;
vec3 indirectSpecular = prefilteredColor * (F * envBRDF.x + envBRDF.y)
```

至此，你应该对 Epic Games 的分割求和近似法的原理，以及它如何近似求解反射方程的间接镜面反射部分有了一些基本印象。让我们现在尝试一下自己构建预卷积部分。

预滤波HDR环境贴图

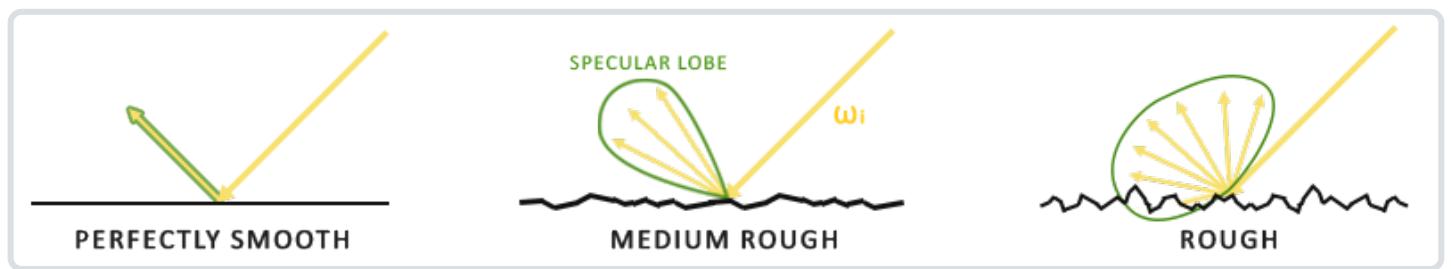
预滤波环境贴图的方法与我们对辐射度贴图求卷积的方法非常相似。对于卷积的每个粗糙度级别，我们将按顺序把模糊后的结果存储在预滤波贴图的 mipmap 中。首先，我们需要生成一个新的立方体贴图来保存预过滤的环境贴图数据。为了确保为其 mip 级别分配足够的内存，一个简单方法是调用 `glGenerateMipmap`。

```
unsigned int prefilterMap;
 glGenTextures(1, &prefilterMap);
 glBindTexture(GL_TEXTURE_CUBE_MAP, prefilterMap);
 for (unsigned int i = 0; i < 6; ++i)
 {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB16F, 128, 128, 0, GL_RGB, GL_FLOAT, nullptr);
 }
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

 glGenerateMipmap(GL_TEXTURE_CUBE_MAP);
```

注意，因为我们计划采样 `prefilterMap` 的 mipmap，所以需要确保将其缩小过滤器设置为 `GL_LINEAR_MIPMAP_LINEAR` 以启用三线性过滤。它存储的是预滤波的镜面反射，基础 mip 级别的分辨率是每面 128×128 ，对于大多数反射来说可能已经足够了，但如果场景里有大量光滑材料（想想汽车上的反射），可能需要提高分辨率。

在上一节教程中，我们使用球面坐标生成均匀分布在半球上的采样向量，以对环境贴图进行卷积。虽然这个方法非常适用于辐照度，但对于镜面反射效果较差。镜面反射依赖于表面的粗糙度，反射光线可能比较松散，也可能比较紧密，但是一定会围绕着反射向量，除非表面极度粗糙：



所有可能出射的反射光构成的形状称为**镜面波瓣**。随着粗糙度的增加，镜面波瓣的大小增加；随着入射光方向不同，形状会发生变化。因此，镜面波瓣的形状高度依赖于材质。在微表面模型里给定入射光方向，则镜面波瓣指向微平面的半向量的反射方向。考虑到大多数光线最终会反射到一个基于半向量的镜面波瓣内，采样时以类似的方式选取采样向量是有意义的，因为大部分其余的向量都被浪费掉了，这个过程称为**重要性采样**。

蒙特卡洛积分和重要性采样

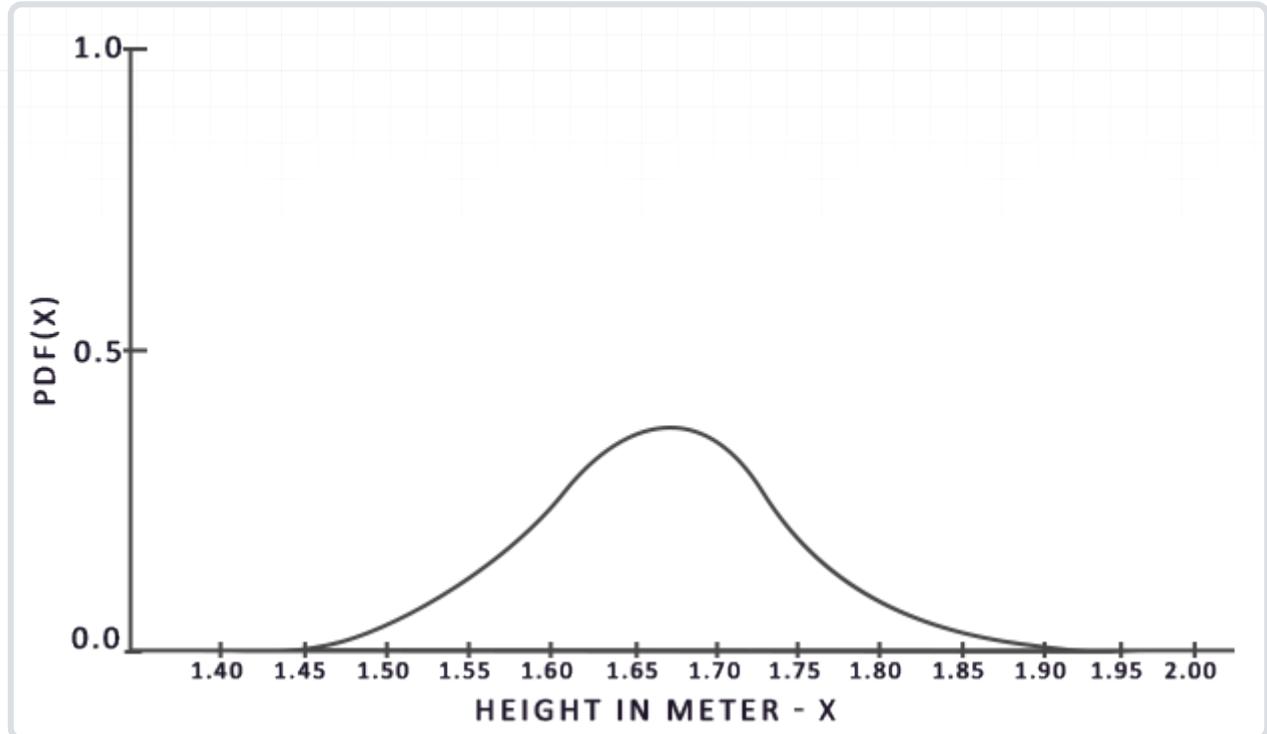
为了充分理解重要性采样，我们首先要了解一种数学结构，称为**蒙特卡洛积分**。蒙特卡洛积分主要是统计和概率理论的组合。蒙特卡洛可以帮助我们离散地解决人口统计问题，而不必考虑所有人。

例如，假设您想要计算一个国家所有公民的平均身高。为了得到结果，你可以测量每个公民并对他们的身高求平均，这样会得到你需要的确切答案。但是，由于大多数国家人海茫茫，这个方法不现实：需要花费太多精力和时间。

另一种方法是选择一个小得多的完全随机（无偏）的人口子集，测量他们的身高并对结果求平均。可能只测量 100 人，虽然答案并非绝对精确，但会得到一个相对接近真相的答案，这个理论被称作**大数定律**。我们的想法是，如果从总人口中测量一组较小的真正随机样本的，结果将相对接近真实答案，并随着样本数 的增加而愈加接近。

蒙特卡罗积分建立在大数定律的基础上，并采用相同的方法来求解积分。不为所有可能的（理论上是无限的）样本值 求解积分，而是简单地从总体中随机挑选样本 生成采样值并求平均。随着 的增加，我们的结果会越来越接近积分的精确结果：

为了求解这个积分，我们在 到 上采样 个随机样本，将它们加在一起并除以样本总数来取平均。代表**概率密度函数** (probability density function)，它的含义是特定样本在整个样本集上发生的概率。例如，人口身高的 pdf 看起来应该像这样：



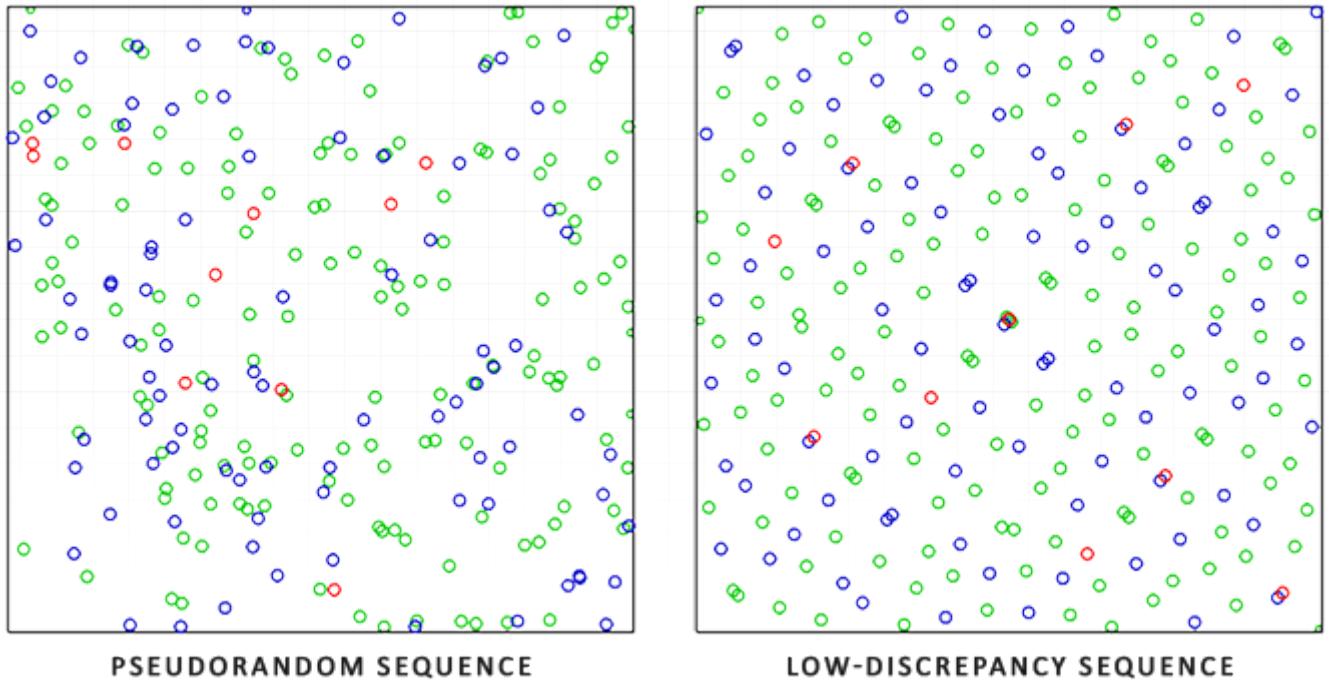
从该图中我们可以看出，如果我们对人口任意随机采样，那么挑选身高为 1.70 的人口样本的可能性更高，而样本身高为 1.50 的概率较低。

当涉及蒙特卡洛积分时，某些样本可能比其他样本具有更高的生成概率。这就是为什么对于任何一般的蒙特卡洛估计，我们都会根据 pdf 将采样值除以或乘以采样概率。到目前为止，我们每次需要估算积分的时候，生成的样本都是均匀分布的，概率完全相等。到目前为止，我们的估计是**无偏的**，这意味着随着样本数量的不断增加，我们最终将**收敛**到积分的精确解。

但是，某些蒙特卡洛估算具有**有偏的**，这意味着生成的样本并不是完全随机的，而是集中于特定的值或方向。这些有偏的蒙特卡洛估算具有**更快的收敛速度**，它们会以更快的速度收敛到精确解，但是由于其有偏性，可能永远不会收敛到精确解。通常来说，这是一个可以接受的折衷方案，尤其是在计算机图形学中。因为只要结果在视觉上可以接受，解决方案的精确性就不太重要。下文我们将会提到一种（有偏的）重要性采样，其生成的样本偏向特定的方向，在这种情况下，我们会将每个样本乘以或除以相应的 pdf 再求和。

蒙特卡洛积分在计算机图形学中非常普遍，因为它是一种以高效的离散方式对连续的积分求近似而且非常直观的方法：对任何面积/体积进行采样——例如半球 —— 在该面积/体积内生成数量 的随机采样，权衡每个样本对最终结果的贡献并求和。

蒙特卡洛积分是一个庞大的数学主题，在此不再赘述，但有一点需要提到：生成随机样本的方法也多种多样。默认情况下，每次采样都是我们熟悉的完全（伪）随机，不过利用半随机序列的某些属性，我们可以生成虽然是随机样本但具有一些有趣性质的样本向量。例如，我们可以对一种名为**低差异序列**的东西进行蒙特卡洛积分，该序列生成的仍然是随机样本，但样本分布更均匀：



当使用低差异序列生成蒙特卡洛样本向量时，该过程称为拟蒙特卡洛积分。拟蒙特卡洛方法具有更快的收敛速度，这使得它对于性能繁重的应用很有用。

鉴于我们新获得的有关蒙特卡洛（Monte Carlo）和拟蒙特卡洛（Quasi-Monte Carlo）积分的知识，我们可以使用一个有趣的属性来获得更快的收敛速度，这就是重要性采样。我们在前文已经提到过它，但是在镜面反射的情况下，反射的光向量被限制在镜面波瓣中，波瓣的大小取决于表面的粗糙度。既然镜面波瓣外的任何（拟）随机生成的样本与镜面面积分无关，因此将样本集中在镜面波瓣内生成是有意义的，但代价是蒙特卡洛估算会产生偏差。

本质上来说，这就是重要性采样的核心：只在某些区域生成采样向量，该区域围绕微表面半向量，受粗糙度限制。通过将拟蒙特卡洛采样与低差异序列相结合，并使用重要性采样偏置样本向量的方法，我们可以获得很高的收敛速度。因为我们求解的速度更快，所以要达到足够的近似度，我们所需要的样本更少。因此，这套组合方法甚至可以允许图形应用程序实时求解镜面面积分，虽然比预算结果还是要慢得多。

低差异序列

在本教程中，我们将使用重要性采样来预算间接反射方程的镜面反射部分，该采样基于拟蒙特卡洛方法给出了随机的低差异序列。我们将使用的序列被称为 Hammersley 序列，[Holger Dammtz](#) 曾仔细描述过它。Hammersley 序列是基于 Van Der Corput 序列，该序列是把十进制数字的二进制表示镜像翻转到小数点右边而得。（译注：原文为 Van Der Corpus 疑似笔误，下文各处同）

给出一些巧妙的技巧，我们可以在着色器程序中非常有效地生成 Van Der Corput 序列，我们将用它来获得 Hammersley 序列，设总样本数为 N ，样本索引为 i ：

```
float RadicalInverse_VdC(uint bits)
{
    bits = (bits << 16u) | (bits >> 16u);
    bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);
    bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);
    bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);
    bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);
    return float(bits) * 2.3283064365386963e-10; // / 0x100000000
}
// -----
vec2 Hammersley(uint i, uint N)
{
    return vec2(float(i)/float(N), RadicalInverse_VdC(i));
}
```

GLSL 的 `Hammersley` 函数可以获取大小为 N 的样本集中的低差异样本 i 。

无需位运算的 Hammersley 序列

并非所有 OpenGL 相关驱动程序都支持位运算符（例如 WebGL 和 OpenGL ES 2.0），在这种情况下，你可能需要不依赖位运算符的替代版本 Van Der Corput 序列：

```
float VanDerCorpus(uint n, uint base)
{
    float invBase = 1.0 / float(base);
    float denom   = 1.0;
    float result  = 0.0;

    for(uint i = 0u; i < 32u; ++i)
    {
        if(n > 0u)
        {
            denom   = mod(float(n), 2.0);
            result += denom * invBase;
            invBase = invBase / 2.0;
            n       = uint(float(n) / 2.0);
        }
    }

    return result;
}
// -----
vec2 HammersleyNoBitOps(uint i, uint N)
{
    return vec2(float(i)/float(N), VanDerCorpus(i, 2u));
}
```

请注意，由于旧硬件中的 GLSL 循环限制，该序列循环遍历了所有可能的 32 位，性能略差。但是如果你没有位运算符可用的话可以考虑它，它可以在所有硬件上运行。

GGX 重要性采样

有别于均匀或纯随机地（比如蒙特卡洛）在积分半球产生采样向量，我们的采样会根据粗糙度，偏向微表面的半向量的宏观反射方向。采样过程将与我们之前看到的过程相似：开始一个大循环，生成一个随机（低差异）序列值，用该序列值在切线空间中生成样本向量，将样本向量变换到世界空间并对场景的辐射度采样。不同之处在于，我们现在使用低差异序列值作为输入来生成采样向量：

```
const uint SAMPLE_COUNT = 4096u;
for(uint i = 0u; i < SAMPLE_COUNT; ++i)
{
    vec2 Xi = Hammersley(i, SAMPLE_COUNT);
}
```

此外，要构建采样向量，我们需要一些方法定向和偏移采样向量，以使其朝向特定粗糙度的镜面波瓣方向。我们可以如理论教程中所述使用 NDF，并将 GGX NDF 结合到 Epic Games 所述的球形采样向量的处理中：

```
vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness)
{
    float a = roughness*roughness;

    float phi = 2.0 * PI * Xi.x;
    float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a*a - 1.0) * Xi.y));
    float sinTheta = sqrt(1.0 - cosTheta*cosTheta);

    // from spherical coordinates to cartesian coordinates
    vec3 H;
    H.x = cos(phi) * sinTheta;
    H.y = sin(phi) * sinTheta;
    H.z = cosTheta;

    // from tangent-space vector to world-space sample vector
    vec3 up      = abs(N.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
    vec3 tangent = normalize(cross(up, N));
    vec3 bitangent = cross(N, tangent);

    vec3 sampleVec = tangent * H.x + bitangent * H.y + N * H.z;
    return normalize(sampleVec);
}
```

基于特定的粗糙度输入和低差异序列值 `xi`，我们获得了一个采样向量，该向量大体围绕着预估的微表面的半向量。注意，根据迪士尼对 PBR 的研究，Epic Games 使用了平方粗糙度以获得更好的视觉效果。

使用低差异 Hammersley 序列和上述定义的样本生成方法，我们可以最终完成预滤波器卷积着色器：

```
#version 330 core
out vec4 FragColor;
in vec3 localPos;

uniform samplerCube environmentMap;
uniform float roughness;

const float PI = 3.14159265359;

float RadicalInverse_VdC(uint bits);
vec2 Hammersley(uint i, uint N);
vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness);

void main()
{
    vec3 N = normalize(localPos);
    vec3 R = N;
    vec3 V = R;

    const uint SAMPLE_COUNT = 1024u;
    float totalWeight = 0.0;
    vec3 prefilteredColor = vec3(0.0);
    for(uint i = 0u; i < SAMPLE_COUNT; ++i)
    {
        vec2 Xi = Hammersley(i, SAMPLE_COUNT);
        vec3 H = ImportanceSampleGGX(Xi, N, roughness);
        vec3 L = normalize(2.0 * dot(V, H) * H - V);

        float NdotL = max(dot(N, L), 0.0);
        if(NdotL > 0.0)
        {
            prefilteredColor += texture(environmentMap, L).rgb * NdotL;
            totalWeight += NdotL;
        }
    }
    prefilteredColor = prefilteredColor / totalWeight;

    FragColor = vec4(prefilteredColor, 1.0);
}
```

输入的粗糙度随着预过滤的立方体贴图的 mipmap 级别变化（从0.0到1.0），我们根据粗糙度预过滤环境贴图，把结果存在 `prefilteredColor` 里。再用 `prefilteredColor` 除以采样权重总和，其中对最终结果影响较小 (`NdotL` 较小) 的采样最终权重也较小。

捕获预过滤 mipmap 级别

剩下要做的就是让 OpenGL 在多个 mipmap 级别上以不同的粗糙度值预过滤环境贴图。有了最开始的辐照度教程作为基础，实际上很简单：

```

prefilterShader.use();
prefilterShader.setInt("environmentMap", 0);
prefilterShader.setMat4("projection", captureProjection);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_CUBE_MAP, envCubemap);

glBindFramebuffer(GL_FRAMEBUFFER, captureFB0);
unsigned int maxMipLevels = 5;
for (unsigned int mip = 0; mip < maxMipLevels; ++mip)
{
    // resize framebuffer according to mip-level size.
    unsigned int mipWidth = 128 * std::pow(0.5, mip);
    unsigned int mipHeight = 128 * std::pow(0.5, mip);
    glBindRenderbuffer(GL_RENDERBUFFER, captureRBO);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, mipWidth, mipHeight);
    glViewport(0, 0, mipWidth, mipHeight);

    float roughness = (float)mip / (float)(maxMipLevels - 1);
    prefilterShader.setFloat("roughness", roughness);
    for (unsigned int i = 0; i < 6; ++i)
    {
        prefilterShader.setMat4("view", captureViews[i]);
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                               GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, prefilterMap, mip);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        renderCube();
    }
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);

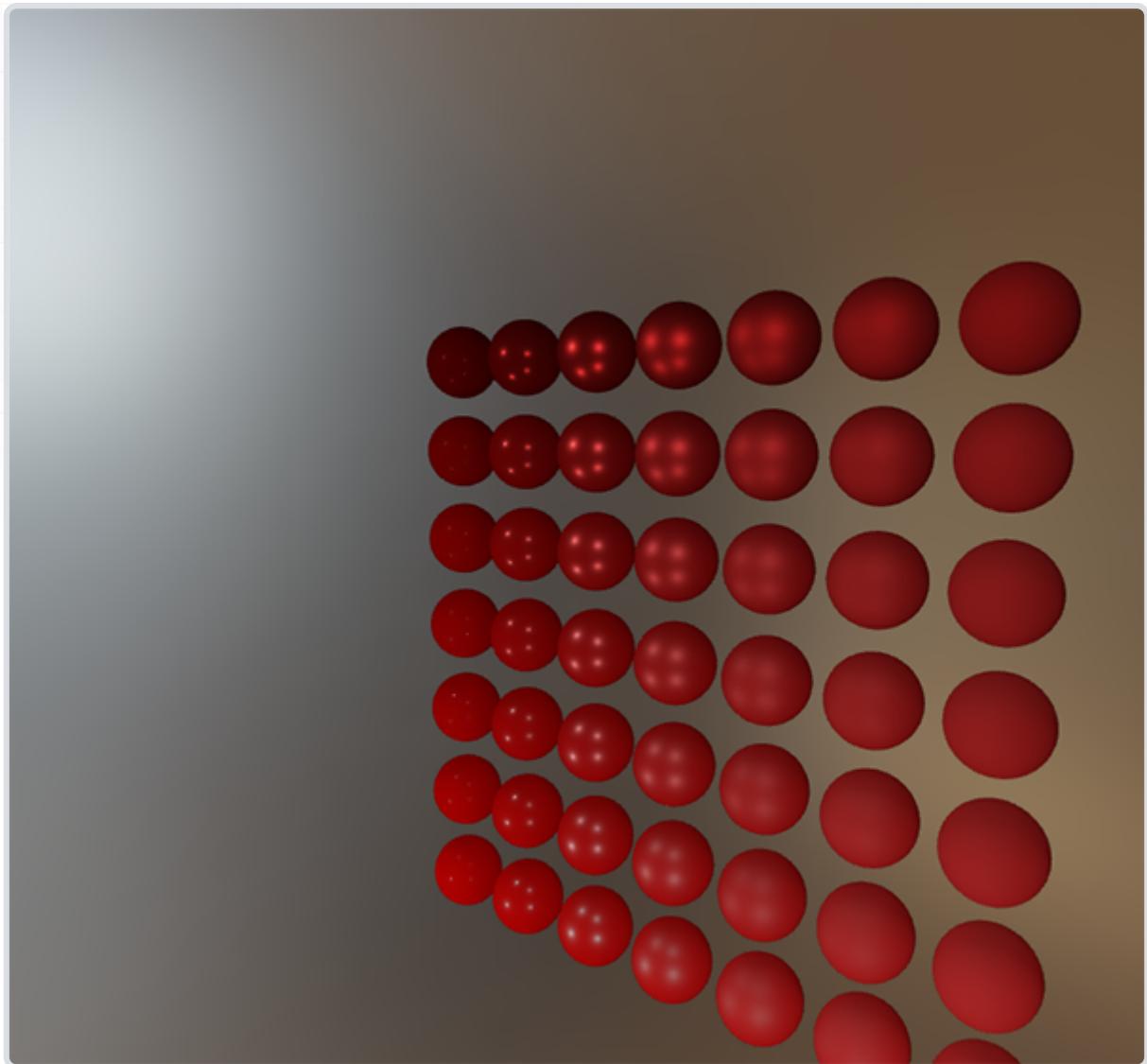
```

这个过程类似于辐照度贴图卷积，但是这次我们将帧缓冲区缩放到适当的 mipmap 尺寸，mip 级别每增加一级，尺寸缩小为一半。此外，我们在 `glFramebufferTexture2D` 的最后一个参数中指定要渲染的目标 mip 级别，然后将要预过滤的粗糙度传给预过滤着色器。

这样我们会得到一张经过适当预过滤的环境贴图，访问该贴图时指定的 mip 等级越高，获得的反射就越模糊。如果我们在天空盒着色器中显示这张预过滤的环境立方体贴图，并在其着色器中强制在其第一个 mip 级别以上采样，如下所示：

```
vec3 envColor = textureLod(environmentMap, WorldPos, 1.2).rgb;
```

我们得到的结果看起来确实像原始环境的模糊版本：



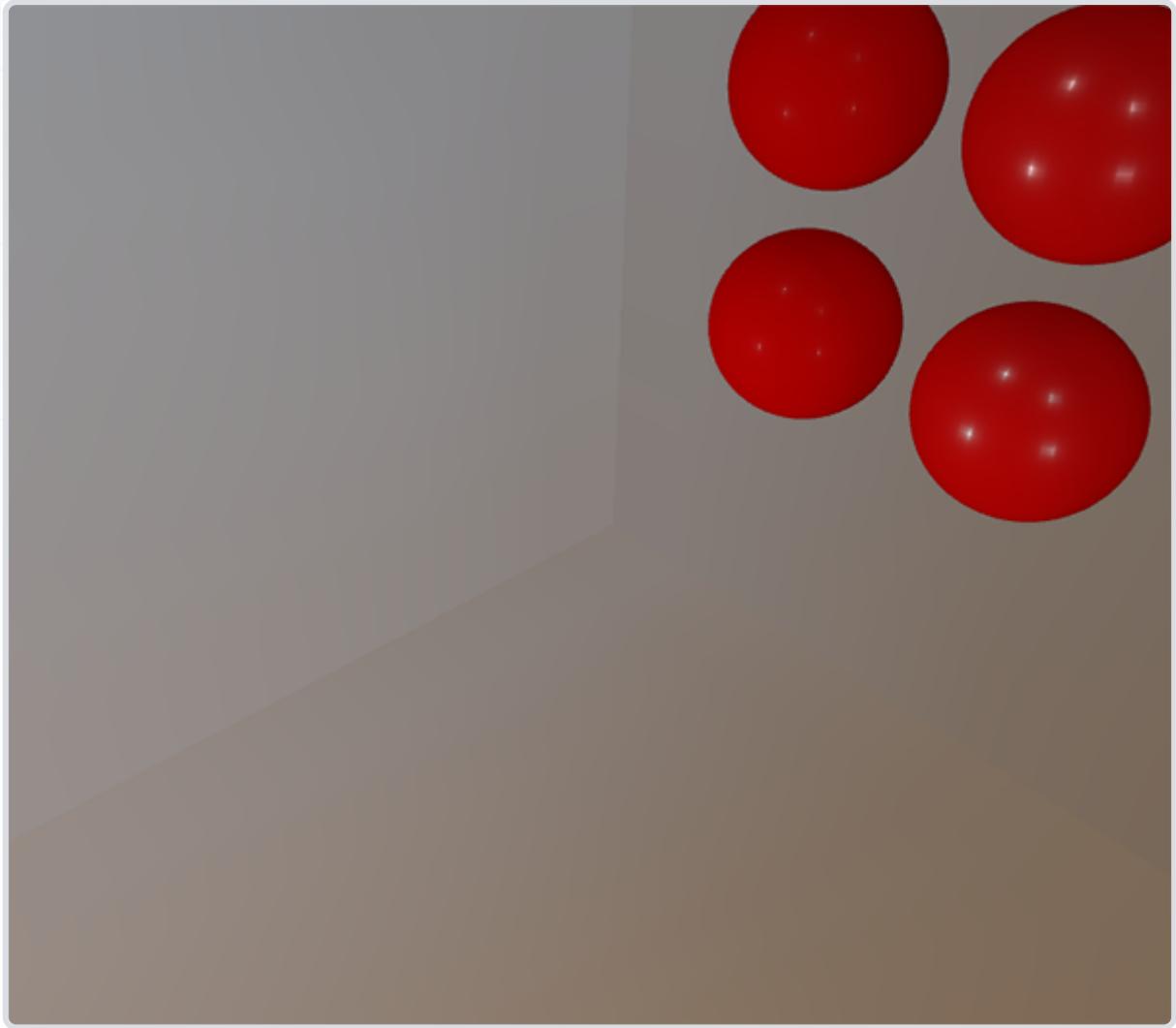
如果 HDR 环境贴图的预过滤看起来差不多没问题，尝试一下不同的 mipmap 级别，观察预过滤贴图随着 mip 级别增加，反射逐渐从锐利变模糊的过程。

预过滤卷积的伪像

当前的预过滤贴图可以在大多数情况下正常工作，不过你迟早会遇到几个与预过滤卷积直接相关的渲染问题。我将在这里列出最常见的一些问题，以及如何修复它们。

高粗糙度的立方体贴图接缝

在具有粗糙表面的表面上对预过滤贴图采样，也就等同于在较低的 mip 级别上对预过滤贴图采样。在对立方体贴图进行采样时，默认情况下，OpenGL 不会在立方体面之间进行线性插值。由于较低的 mip 级别具有更低的分辨率，并且预过滤贴图代表了与更大的采样波瓣卷积，因此缺乏立方体的面和面之间的滤波的问题就更明显：

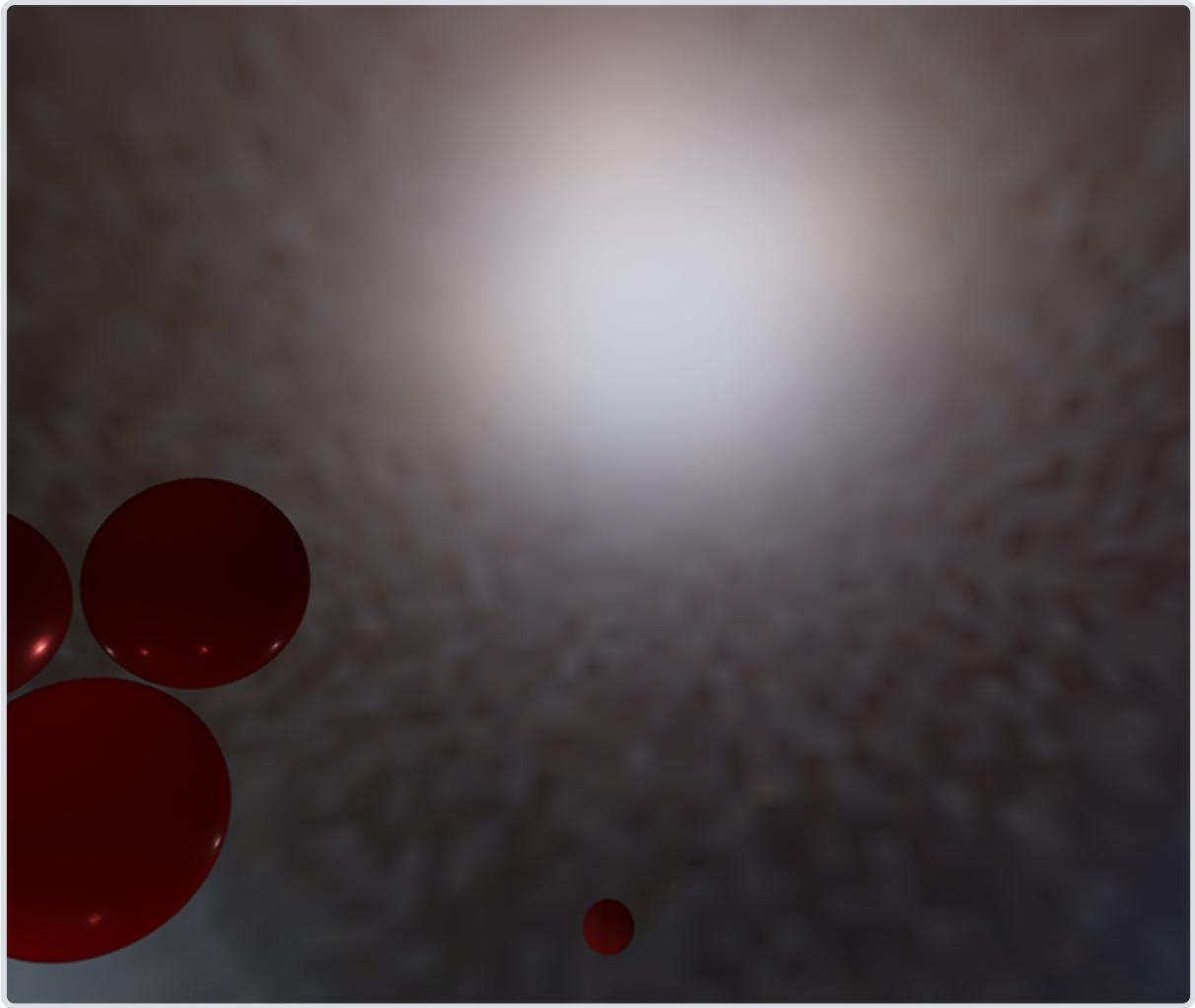


幸运的是，OpenGL 可以启用 `GL_TEXTURE_CUBE_MAP_SEAMLESS`，以为我们提供在立方体贴图的面之间进行正确过滤的选项：

```
glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
```

预过滤卷积的亮点

由于镜面反射中光强度的变化大，高频细节多，所以对镜面反射进行卷积需要大量采样，才能正确反映 HDR 环境反射的混乱变化。我们已经进行了大量的采样，但是在某些环境下，在某些较粗糙的 mip 级别上可能仍然不够，导致明亮区域周围出现点状图案：



一种解决方案是进一步增加样本数量，但在某些情况下还是不够。另一种方案如 [Chetan Jags](#) 所述，我们可以在预过滤卷积时，不直接采样环境贴图，而是基于积分的 PDF 和粗糙度采样环境贴图的 mipmap，以减少伪像：

```
float D = DistributionGGX(NdotH, roughness);
float pdf = (D * NdotH / (4.0 * HdotV)) + 0.0001;

float resolution = 512.0; // resolution of source cubemap (per face)
float saTexel = 4.0 * PI / (6.0 * resolution * resolution);
float saSample = 1.0 / (float(SAMPLE_COUNT) * pdf + 0.0001);

float mipLevel = roughness == 0.0 ? 0.0 : 0.5 * log2(saSample / saTexel);
```

既然要采样 mipmap，不要忘记在环境贴图上开启三线性过滤：

```
glBindTexture(GL_TEXTURE_CUBE_MAP, envCubemap);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

设置立方体贴图的基本纹理后，让 OpenGL 生成 mipmap：

```
// convert HDR equirectangular environment map to cubemap equivalent
[...]
// then generate mipmaps
glBindTexture(GL_TEXTURE_CUBE_MAP, envCubemap);
glGenerateMipmap(GL_TEXTURE_CUBE_MAP);
```

这个方法效果非常好，可以去除预过滤贴图中较粗糙表面上的大多数甚至全部亮点。

预算计算 BRDF

预过滤的环境贴图已经可以设置并运行，我们可以集中精力于求和近似的第二部分：BRDF。让我们再次简要回顾一下镜面部分的分割求和近似法：

我们已经在预过滤贴图的各个粗糙度级别上预算算了分割求和近似的左半部分。右半部分要求我们在、表面粗糙度、菲涅尔系数上计算 BRDF 方程的卷积。这等同于在纯白的环境光或者辐射度恒定为 =1.0 的设置下，对镜面 BRDF 求积分。对3个变量做卷积有点复杂，不过我们可以把 移出镜面 BRDF 方程：

为菲涅耳方程。将菲涅耳分母移到 BRDF 下面可以得到如下等式：

用 Fresnel-Schlick 近似公式替换右边的 可以得到：

让我们用 替换 以便更轻松地求解：

然后我们将菲涅耳函数 分拆到两个积分里：

这样，在整个积分上是恒定的，我们可以从积分中提取出。接下来，我们将替换回其原始形式，从而得到最终分割求和的 BRDF 方程：

公式中的两个积分分别表示 的比例和偏差。注意，由于 已经包含 项，它们被约分了，这里的 中不计算 项。

和之前卷积环境贴图类似，我们可以对 BRDF 方程求卷积，其输入是 和 的夹角，以及粗糙度，并将卷积的结果存储在纹理中。我们将卷积后的结果存储在 2D 查找纹理（Look Up Texture, LUT）中，这张纹理被称为 **BRDF 积分贴图**，稍后会将其用于 PBR 光照着色器中，以获得间接镜面反射的最终卷积结果。

BRDF 卷积着色器在 2D 平面上执行计算，直接使用其 2D 纹理坐标作为卷积输入（`NdotV` 和 `roughness`）。代码与预滤波器的卷积代码大体相似，不同之处在于，它现在根据 BRDF 的几何函数和 Fresnel-Schlick 近似来处理采样向量：

```

vec2 IntegrateBRDF(float NdotV, float roughness)
{
    vec3 V;
    V.x = sqrt(1.0 - NdotV*NdotV);
    V.y = 0.0;
    V.z = NdotV;

    float A = 0.0;
    float B = 0.0;

    vec3 N = vec3(0.0, 0.0, 1.0);

    const uint SAMPLE_COUNT = 1024u;
    for(uint i = 0u; i < SAMPLE_COUNT; ++i)
    {
        vec2 Xi = Hammersley(i, SAMPLE_COUNT);
        vec3 H = ImportanceSampleGGX(Xi, N, roughness);
        vec3 L = normalize(2.0 * dot(V, H) * H - V);

        float NdotL = max(L.z, 0.0);
        float NdotH = max(H.z, 0.0);
        float VdotH = max(dot(V, H), 0.0);

        if(NdotL > 0.0)
        {
            float G = GeometrySmith(N, V, L, roughness);
            float G_Vis = (G * VdotH) / (NdotH * NdotV);
            float Fc = pow(1.0 - VdotH, 5.0);

            A += (1.0 - Fc) * G_Vis;
            B += Fc * G_Vis;
        }
    }
    A /= float(SAMPLE_COUNT);
    B /= float(SAMPLE_COUNT);
    return vec2(A, B);
}

// -----
void main()
{
    vec2 integratedBRDF = IntegrateBRDF(TexCoords.x, TexCoords.y);
    FragColor = integratedBRDF;
}

```

如你所见，BRDF 卷积部分是从数学到代码的直接转换。我们将角度 和粗糙度作为输入，以重要性采样产生采样向量，在整个几何体上结合 BRDF 的菲涅耳项对向量进行处理，然后输出每个样本上的系数和偏差，最后取平均值。

你可能回想起[理论](#)教程中的一个细节：与 IBL 一起使用时，BRDF 的几何项略有不同，因为 变量的含义稍有不同：

由于 BRDF 卷积是镜面 IBL 积分的一部分，因此我们要在 Schlick-GGX 几何函数中使用：

```
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float a = roughness;
    float k = (a * a) / 2.0;

    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

// -----
float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}
```

请注意，虽然还是从 a 计算出来的，但这里的 a 不是 `roughness` 的平方——如同最初对 a 的其他解释那样——在这里我们假装平方过了。我不确定这样处理是否与 Epic Games 或迪士尼原始论文不一致，但是直接将 `roughness` 赋给 a 得到的 BRDF 积分贴图与 Epic Games 的版本完全一致。

最后，为了存储 BRDF 卷积结果，我们需要生成一张 512×512 分辨率的 2D 纹理。

```
unsigned int brdfLUTTexture;
 glGenTextures(1, &brdfLUTTexture);

// pre-allocate enough memory for the LUT texture.
 glBindTexture(GL_TEXTURE_2D, brdfLUTTexture);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RG16F, 512, 512, 0, GL_RG, GL_FLOAT, 0);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

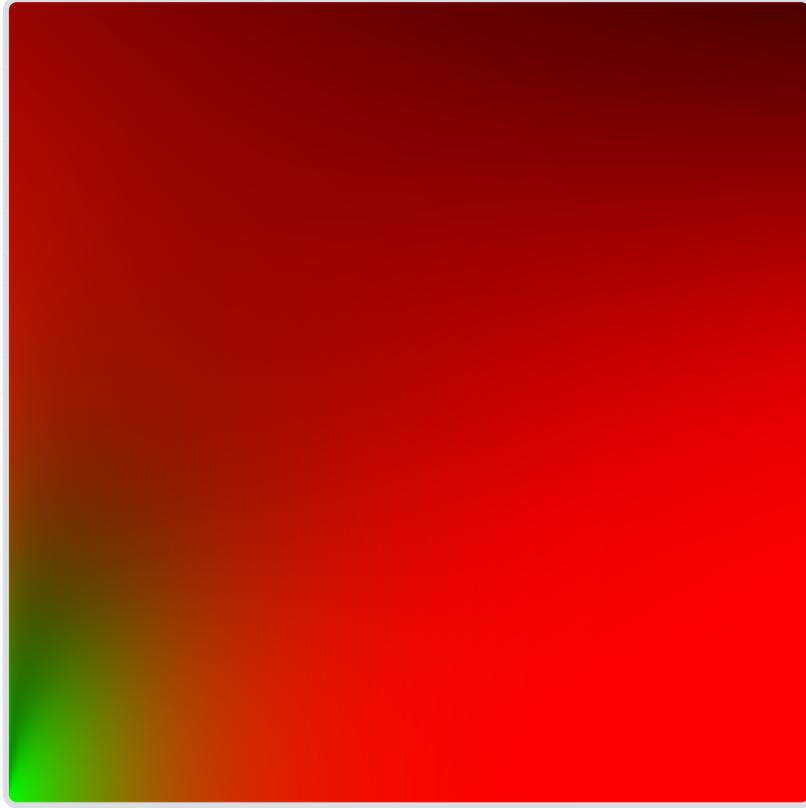
请注意，我们使用的是 Epic Games 推荐的16位精度浮点格式。将环绕模式设置为 `GL_CLAMP_TO_EDGE` 以防止边缘采样的伪像。然后，我们复用同一个帧缓冲区对象，并在 NDC (译注：Normalized Device Coordinates) 屏幕空间四边形上运行此着色器：

```
glBindFramebuffer(GL_FRAMEBUFFER, captureFB0);
glBindRenderbuffer(GL_RENDERBUFFER, captureRB0);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, 512, 512);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, brdfLUTTexture, 0);

glViewport(0, 0, 512, 512);
brdfShader.use();
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
RenderQuad();

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

分割积分和的 BRDF 卷积部分应该得到以下结果：



预过滤的环境贴图和 BRDF 的 2D LUT 都已经齐备，我们可以根据分割求和近似法重建间接镜面部分积分了。最后合并的结果将被用作间接镜面反射或环境镜面反射。

完成 IBL 反射

为了使反射方程的间接镜面反射部分正确运行，我们需要将分割求和近似法的两个部分缝合在一起。第一步是将预计算的光照数据声明到 PBR 着色器的最上面：

```
uniform samplerCube prefilterMap;
uniform sampler2D brdfLUT;
```

首先，使用反射向量采样预过滤的环境贴图，获取表面的间接镜面反射。请注意，我们会根据表面粗糙度在合适的 mip 级别采样，以使更粗糙的表面产生更模糊的镜面反射。

```
void main()
{
    [...]
    vec3 R = reflect(-V, N);

    const float MAX_REFLECTION_LOD = 4.0;
    vec3 prefilteredColor = textureLod(prefilterMap, R, roughness * MAX_REFLECTION_LOD).rgb;
    [...]
}
```

在预过滤步骤中，我们仅将环境贴图卷积最多 5 个 mip 级别（0到4），此处记为 `MAX_REFLECTION_LOD`，以确保不会对一个没有数据的 mip 级别采样。然后我们用已知的材质粗糙度和视线–法线夹角作为输入，采样 BRDF LUT。

```
vec3 F      = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);
vec2 envBRDF = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;
vec3 specular = prefilteredColor * (F * envBRDF.x + envBRDF.y);
```

这样我们就从 BRDF LUT 中获得了 的系数和偏移，这里我们就直接用间接光菲涅尔项 `F` 代替。把这个结果和 IBL 反射方程左边的预过滤部分结合起来，以重建整个近似积分，存入 `specular`。

于是我们得到了反射方程的间接镜面反射部分。现在，将其与 [上一节教程](#) 中的反射方程的漫反射部分结合起来，我们可以获得完整的 PBR IBL 结果：

```
vec3 F = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);

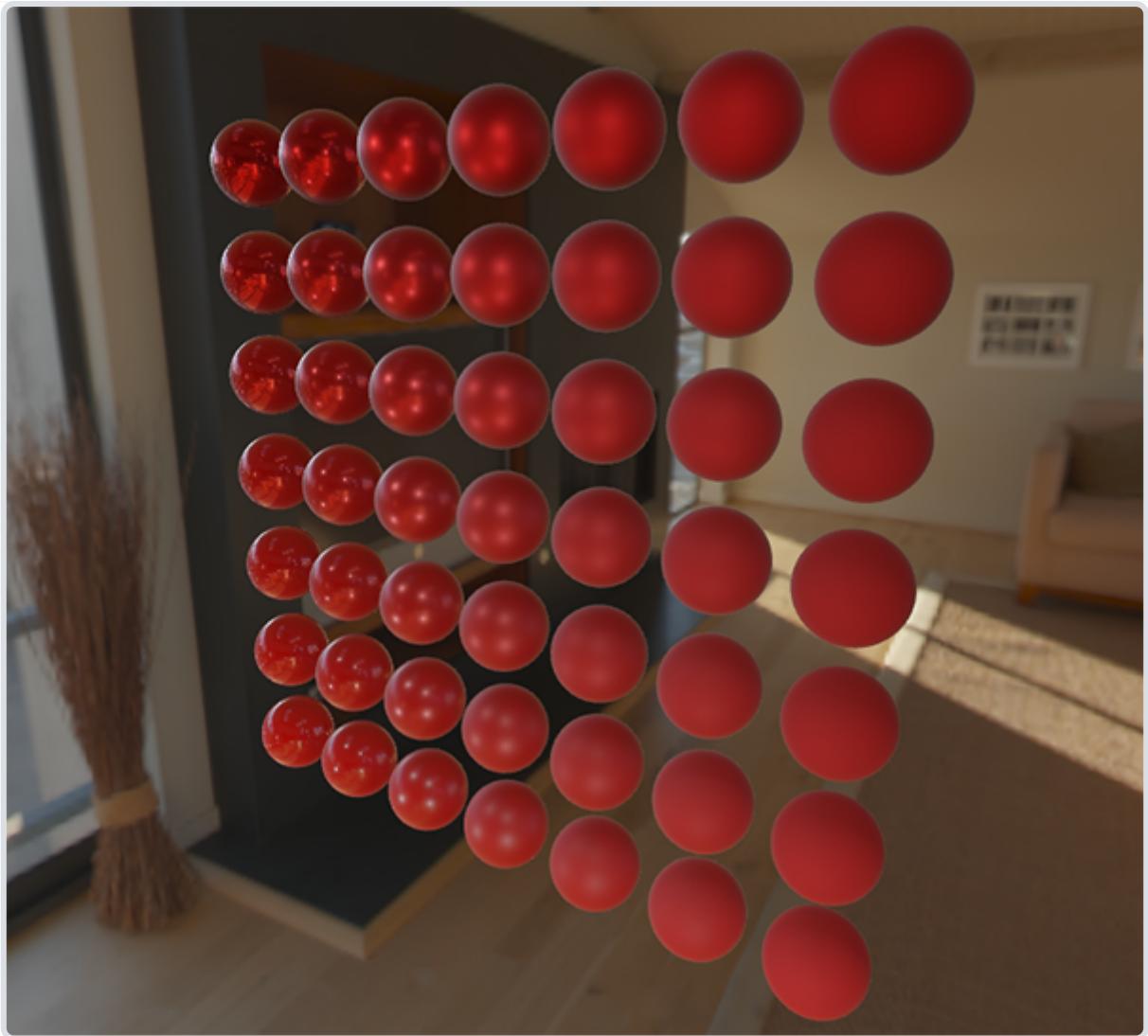
vec3 kS = F;
vec3 kD = 1.0 - kS;
kD *= 1.0 - metallic;

vec3 irradiance = texture(irradianceMap, N).rgb;
vec3 diffuse    = irradiance * albedo;

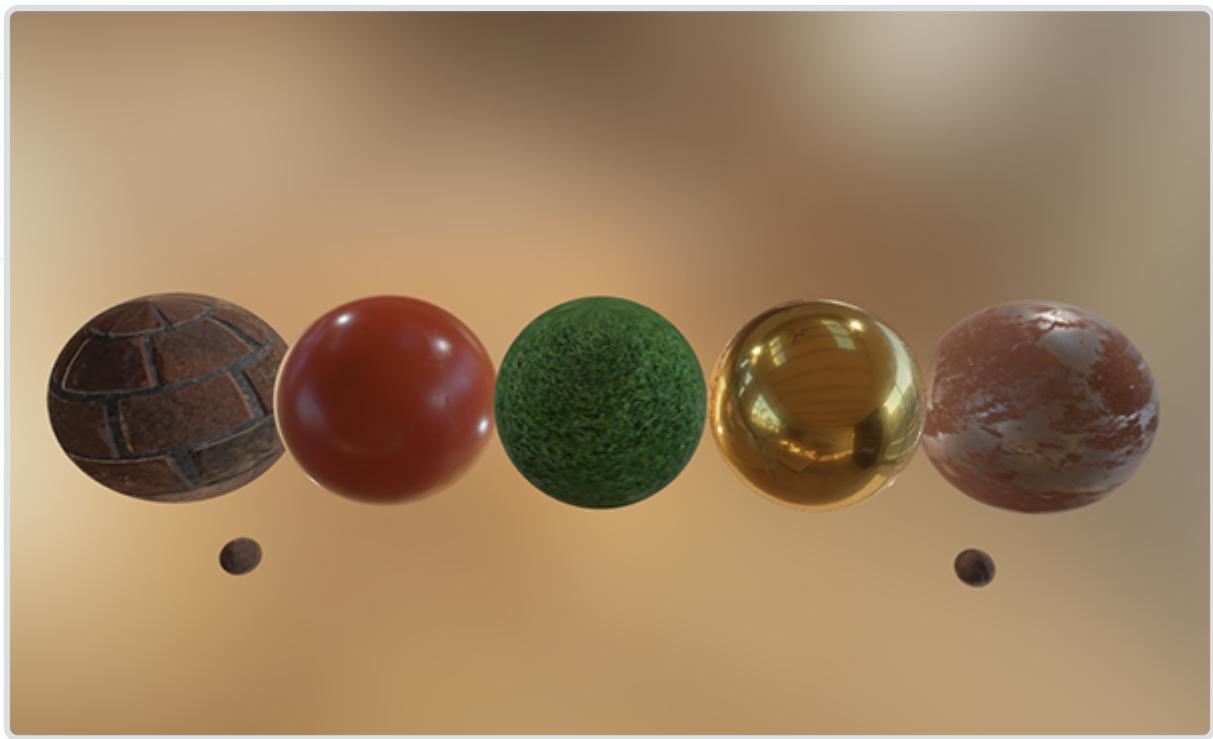
const float MAX_REFLECTION_LOD = 4.0;
vec3 prefilteredColor = textureLod(prefilterMap, R, roughness * MAX_REFLECTION_LOD).rgb;
vec2 envBRDF = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;
vec3 specular = prefilteredColor * (F * envBRDF.x + envBRDF.y);

vec3 ambient = (kD * diffuse + specular) * ao;
```

请注意，`specular` 没有乘以 `ks`，因为已经乘过了菲涅耳系数。现在，在一系列粗糙度和金属度各异的球上运行此代码，我们终于可以在最终的 PBR 渲染器中看到其真实颜色：



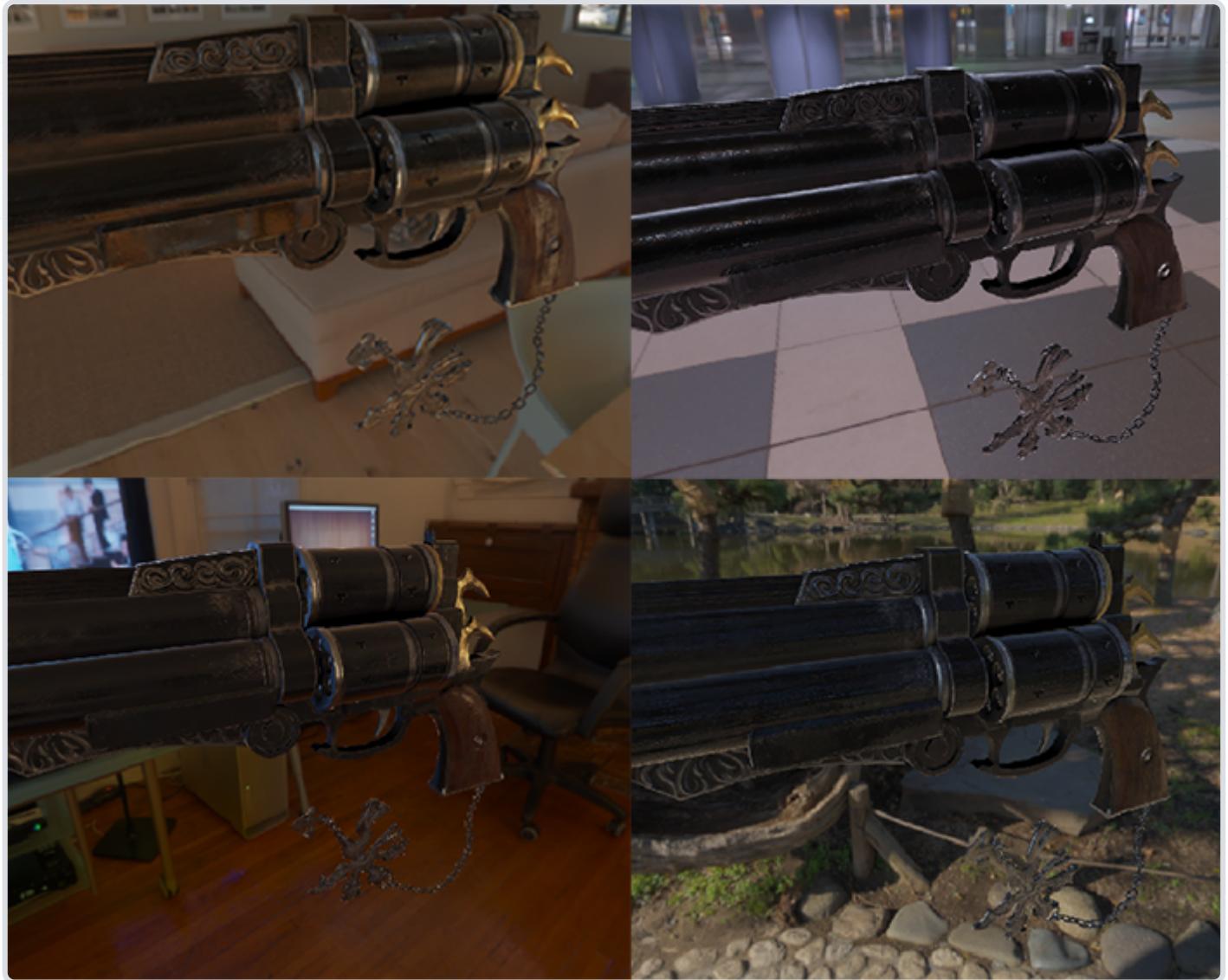
我们甚至可以再疯狂一点，使用一些带酷炫纹理的 [PBR 材质](#)：



或加载 Andrew Maximov 的[这款出色的免费 PBR 3D 模型](#):



我敢肯定我们都同意现在的光照看起来更具说服力。更妙的是，无论我们使用哪种环境贴图，我们的光照看起来都是物理正确的。下面，您将看到几张不同的预算算 HDR 贴图，它们完全改变了光照动态，但是不需要调整任何光照变量，在外观上依然正确！



好吧，这是一场长长的 PBR 冒险。有很多步骤可能会出错，所以如果遇到问题卡住，请仔细研究[球形场景或带纹理的场景代码示例](#)——也包括所有着色器，或者检查之后在评论中提问。

下一步是？

希望在本教程结束时，你会对 PBR 的相关内容有一个清晰的了解，甚至可以构造并运行一个实际的 PBR 渲染器。在这几节教程中，我们已经在应用程序开始阶段，渲染循环之前，预算算了所有 PBR 相关的基于图像的光照数据。出于教育目的，这很好，但对于任何 PBR 的实践应用来说，都不是很漂亮。首先，预算算实际上只需要执行一次，而不是每次启动时都要做。其次，当使用多个环境贴图时，你必须在每次程序启动时全部预算算一遍，这是个必须步骤。

因此，通常只需要一次将环境贴图预算算为辐照度贴图和预过滤贴图，然后将其存储在磁盘上（注意，BRDF 积分贴图不依赖于环境贴图，因此只需要计算或加载一次）。这意味着您需要提出一种自定义图像格式来存储 HDR 立方体贴图，包括其 mip 级别。或者将图像存储为某种可用格式——例如支持存储 mip 级别的 .dds——并按其格式加载。

此外，我们也在教程中描述了整个过程，包括生成预算算的 IBL 图像，以帮助我们进一步了解 PBR 管线。此外还可以通过 [cmftStudio](#) 或 [IBLBaker](#) 等一些出色的工具为您生成这些预算算贴图，也很好用。

有一点内容我们跳过了，即如何将预算算的立方体贴图作为反射探针：立方体贴图插值和视差校正。这是一个在场景中放置多个反射探针的过程，这些探针在特定位置拍摄场景的立方体贴图快照，然后我们可以将其卷积，作为相应部分场景的 IBL 数据。基于相机的位置对附近的探针插值，我们可以实现局部的细节丰富的 IBL，受到的唯一限制就是探针放置的数量。这样一来，例如从一个明亮的室外部分移动到较暗的室内部分时，IBL 就能正确更新。我将来会在某个地方编写有关反射探针的教程，但现在，我建议阅读下面 Chetan Jags 的文章来作为入门。

进阶阅读

- [Real Shading in Unreal Engine 4](#)：讲解了 Epic Games 的分割求和近似法。IBL PBR 部分的代码就脱胎于此文。
- [Physically Based Shading and Image Based Lighting](#)：Trent Reed 的精彩博客文章，介绍了如何将镜面反射 IBL 实时集成到 PBR 管道中。
- [Image Based Lighting](#)：Chetan Jags 对基于镜面反射的 IBL 及其一些注意事项（包括光照探针插值）进行了广泛的讲解。
- [Moving Frostbite to PBR](#)：Sébastien Lagarde 和 Charles de Rousiers 撰写的，对于如何将 PBR 集成到 AAA 游戏引擎进行了详尽而深入的概述。
- [Physically Based Rendering – Part Three](#)：JMonkeyEngine 团队对 IBL 和 PBR 进行了较高层次的概述。
- [Implementation Notes: Runtime Environment Map Filtering for Image Based Lighting](#)：Padraic Hennessy 撰写的大量有关预过滤 HDR 环境贴图并显著优化采样过程的文章。

2.8 实战

2.8.1 调试

原文 [Debugging](#)

作者 JoeydeVries

翻译 Krasjet

校对 暂无

图形编程可以带来很多的乐趣，然而如果什么东西渲染错误，或者甚至根本没有渲染，它同样可以给你带来大量的沮丧感！由于我们大部分时间都在与像素打交道，当出现错误的时候寻找错误的源头可能会非常困难。调试(Debug)这样的视觉错误与往常熟悉的CPU调试不同。我们没有一个可以用来输出文本的控制台，在GLSL代码中也不能设置断点，更没有办法检测GPU的运行状态。

在这篇教程中，我们将来见识几个调试OpenGL程序的技巧。OpenGL中的调试并不是很难，掌握住这些技巧对之后的学习会有非常大的帮助。

glGetError()

当你不正确使用OpenGL的时候（比如说在绑定之前配置一个缓冲），它会检测到，并在幕后生成一个或多个用户错误标记。我们可以使用一个叫做`glGetError`的函数查询这些错误标记，他会检测错误标记集，并且在OpenGL确实出错的时候返回一个错误值。

```
GLenum glGetError();
```

当`glGetError`被调用时，它要么会返回错误标记之一，要么返回无错误。`glGetError`会返回的错误值如下：

标记	代码 描述
GL_NO_ERROR	0 自上次调用 <code>glGetError</code> 以来没有错误
GL_INVALID_ENUM	1280 枚举参数不合法
GL_INVALID_VALUE	1281 值参数不合法
GL_INVALID_OPERATION	1282 一个指令的状态对指令的参数不合法
GL_STACK_OVERFLOW	1283 压栈操作造成栈上溢(Overflow)
GL_STACK_UNDERFLOW	1284 弹栈操作时栈在最低点（译注：即栈下溢(Underflow)）
GL_OUT_OF_MEMORY	1285 内存调用操作无法调用（足够的）内存
GL_INVALID_FRAMEBUFFER_OPERATION	1286 读取或写入一个不完整的帧缓冲

在OpenGL的函数文档中你可以找到函数在错误时生成的错误代码。比如说，如果你看一下[glBindTexture](#)函数的文档，在 *Errors* 部分中你可以看到它可能生成的所有用户错误代码。

当一个错误标记被返回的时候，将不会报告其它的错误标记。换句话说，当`glGetError`被调用的时候，它会清除所有的错误标记（在分布式系统上只会清除一个，见下面的注释）。这也就是说如果你在每一帧之后调用`glGetError`一次，它返回一个错误，但你不能确定这就是唯一的错误，并且错误的来源可能在这一帧的任意地方。

注意当OpenGL是分布式(Distributely)运行的时候，如在X11系统上，其它的用户错误代码仍然可以被生成，只要它们有着不同的错误代码。调用`glGetError`只会重置一个错误代码标记，而不是所有。由于这一点，我们通常会建议在一个循环中调用`glGetError`。

```
glBindTexture(GL_TEXTURE_2D, tex);
std::cout << glGetError() << std::endl; // 返回 0 (无错误)

glTexImage2D(GL_TEXTURE_3D, 0, GL_RGB, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
std::cout << glGetError() << std::endl; // 返回 1280 (非法枚举)

 glGenTextures(-5, textures);
std::cout << glGetError() << std::endl; // 返回 1281 (非法值)

std::cout << glGetError() << std::endl; // 返回 0 (无错误)
```

`glGetError`非常棒的一点就是它能够非常简单地定位错误可能的来源，并且验证OpenGL使用的正确性。比如说你获得了一个黑屏的结果但是不知道什么造成了它：是不是帧缓冲设置错误？是不是我忘记绑定纹理了？通过在代码中各处调用`glGetError`，你可以非常快速地查明OpenGL错误开始出现的位置，这也意味着这次调用之前的代码中哪里出错了。

默认情况下`glGetError`只会打印错误数字，如果你不去记忆的话会非常难以理解。通常我们会写一个助手函数来简便地打印出错误字符串以及错误检测函数调用的位置。

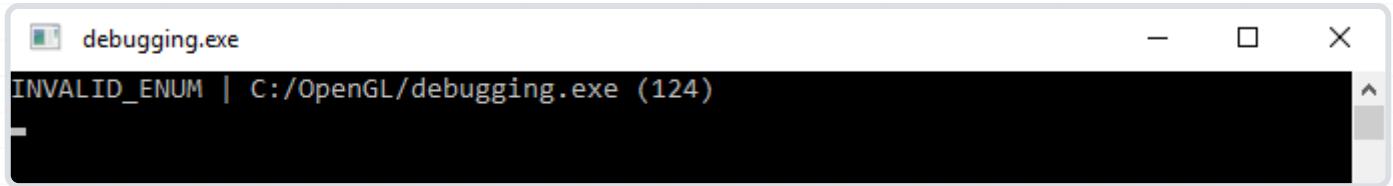
```
Glenum glCheckError_(const char *file, int line)
{
    Glenum errorCode;
    while ((errorCode = glGetError()) != GL_NO_ERROR)
    {
        std::string error;
        switch (errorCode)
        {
            case GL_INVALID_ENUM:           error = "INVALID_ENUM"; break;
            case GL_INVALID_VALUE:          error = "INVALID_VALUE"; break;
            case GL_INVALID_OPERATION:      error = "INVALID_OPERATION"; break;
            case GL_STACK_OVERFLOW:         error = "STACK_OVERFLOW"; break;
            case GL_STACK_UNDERFLOW:        error = "STACK_UNDERFLOW"; break;
            case GL_OUT_OF_MEMORY:          error = "OUT_OF_MEMORY"; break;
            case GL_INVALID_FRAMEBUFFER_OPERATION: error = "INVALID_FRAMEBUFFER_OPERATION"; break;
        }
        std::cout << error << " | " << file << "(" << line << ")" << std::endl;
    }
    return errorCode;
}

#define glCheckError() glCheckError_(__FILE__, __LINE__)
```

防止你不知道`__FILE__`和`__LINE__`这两个预处理指令(Preprocessor Directive)是什么，它们会在编译的时候被替换成编译时对应的文件与行号。如果我们坚持在代码中使用大量`glGetError`的调用，这就会让我们更加准确地知道哪个`glGetError`调用返回了错误（译注：记得`glGetError`显示的错误会发生在该次调用与上次调用之间，如果间隔太大的话需要检查的地方就太多了）。

```
glBindBuffer(GL_VERTEX_ARRAY, vbo);
glCheckError();
```

这会给我们以下的输出：



还有一个重要的事情需要知道，GLEW有一个历史悠久的bug，调用`glewInit()`会设置一个`GL_INVALID_ENUM`的错误标记，所以第一次调用的`glGetError`永远会猝不及防地给你返回一个错误代码。如果要修复这个bug，我们建议您在调用`glewInit`之后立即调用`glGetError`消除这个标记：

```
glewInit();
glGetError();
```

`glGetError`并不能帮助你很多，因为它返回的信息非常简单，但不可否认它经常能帮助你检查笔误或者快速定位错误来源。总而言之，是一个非常简单但有效的工具。

调试输出

虽然没有`glGetError`普遍，但一个叫做**调试输出**(Debug Output)的OpenGL拓展是一个非常有用的工具，它在4.3版本之后变为了核心OpenGL的一部分。通过使用调试输出拓展，OpenGL自身会直接发送一个比起`glGetError`更为完善的错误或警告信息给用户。它不仅提供了更多的信息，也能够帮助你使用调试器(Debugger)捕捉错误源头。

调试输出自4.3版本变为核心OpenGL的一部分，这也就是说你可以在任意运行OpenGL 4.3及以上版本的机器中找到这一功能。如果OpenGL低于这一版本，你可以查询 `ARB_debug_output` 或者 `AMD_debug_output` 拓展来获取它的功能。注意OS X好像不支持调试输出功能（从网上看到的，我暂时还没有测试。如果我错了请告诉我一下）

要想开始使用调试输出，我们先要在初始化进程中从OpenGL请求一个调试输出上下文。这个进程根据你的窗口系统会有所不同，这里我们只会讨论在GLFW中配置，但你可以在教程最后的附加资源处找到其它系统的相关资料。

GLFW中的调试输出

在GLFW中请求一个调试输出非常简单，我们只需要传递一个提醒到GLFW中，告诉它我们需要一个调试输出上下文即可。我们需要在调用`glfwCreateWindow`之前完成这一请求。

```
glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);
```

一旦GLFW初始化完成，如果我们使用的OpenGL 版本为4.3或以上的话我们就会有一个调试上下文了，如果不是的话则祈祷系统仍然能够请求一个调试上下文吧。如果还是不行的话我们必须使用它的OpenGL拓展来请求调试输出。

在调试上下文中使用OpenGL会明显更缓慢一点，所以当你在优化或者发布程序之前请将这一GLFW调试请求给注释掉。

要检查我们是否成功地初始化了调试上下文，我们可以对OpenGL进行查询：

```
GLint flags; glGetIntegerv(GL_CONTEXT_FLAGS, &flags);
if (flags & GL_CONTEXT_FLAG_DEBUG_BIT)
{
    // 初始化调试输出
}
```

调试输出工作的方式是这样的，我们首先将一个错误记录函数的回调（类似于GLFW输入的回调）传递给OpenGL，在这个回调函数中我们可以自由地处理OpenGL错误数据，在这里我们将输出一些有用的数据到控制台中。下面是这个就是OpenGL对调试输出所期待的回调函数的原型：

```
void APIENTRY glDebugOutput(GLenum source, GLenum type, GLuint id, GLenum severity,
                           GLsizei length, const GLchar *message, void *userParam);
```

注意在OpenGL的某些实现中最后一个参数为 `const void*` 而不是 `void*`。

有了这一大堆的数据，我们可以创建一个非常有用错误打印工具：

```

void APIENTRY glDebugOutput(GLenum source,
                            GLenum type,
                            GLuint id,
                            GLenum severity,
                            GLsizei length,
                            const GLchar *message,
                            void *userParam)
{
    // 忽略一些不重要的错误/警告代码
    if(id == 131169 || id == 131185 || id == 131218 || id == 131204) return;

    std::cout << "-----" << std::endl;
    std::cout << "Debug message (" << id << "): " << message << std::endl;

    switch (source)
    {
        case GL_DEBUG_SOURCE_API:           std::cout << "Source: API"; break;
        case GL_DEBUG_SOURCE_WINDOW_SYSTEM: std::cout << "Source: Window System"; break;
        case GL_DEBUG_SOURCE_SHADER_COMPILER: std::cout << "Source: Shader Compiler"; break;
        case GL_DEBUG_SOURCE_THIRD_PARTY:    std::cout << "Source: Third Party"; break;
        case GL_DEBUG_SOURCE_APPLICATION:   std::cout << "Source: Application"; break;
        case GL_DEBUG_SOURCE_OTHER:         std::cout << "Source: Other"; break;
    } std::cout << std::endl;

    switch (type)
    {
        case GL_DEBUG_TYPE_ERROR:          std::cout << "Type: Error"; break;
        case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR: std::cout << "Type: Deprecated Behaviour"; break;
        case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR: std::cout << "Type: Undefined Behaviour"; break;
        case GL_DEBUG_TYPE_PORTABILITY:     std::cout << "Type: Portability"; break;
        case GL_DEBUG_TYPE_PERFORMANCE:    std::cout << "Type: Performance"; break;
        case GL_DEBUG_TYPE_MARKER:         std::cout << "Type: Marker"; break;
        case GL_DEBUG_TYPE_PUSH_GROUP:     std::cout << "Type: Push Group"; break;
        case GL_DEBUG_TYPE_POP_GROUP:      std::cout << "Type: Pop Group"; break;
        case GL_DEBUG_TYPE_OTHER:          std::cout << "Type: Other"; break;
    } std::cout << std::endl;

    switch (severity)
    {
        case GL_DEBUG_SEVERITY_HIGH:       std::cout << "Severity: high"; break;
        case GL_DEBUG_SEVERITY_MEDIUM:     std::cout << "Severity: medium"; break;
        case GL_DEBUG_SEVERITY_LOW:        std::cout << "Severity: low"; break;
        case GL_DEBUG_SEVERITY_NOTIFICATION: std::cout << "Severity: notification"; break;
    } std::cout << std::endl;
    std::cout << std::endl;
}

```

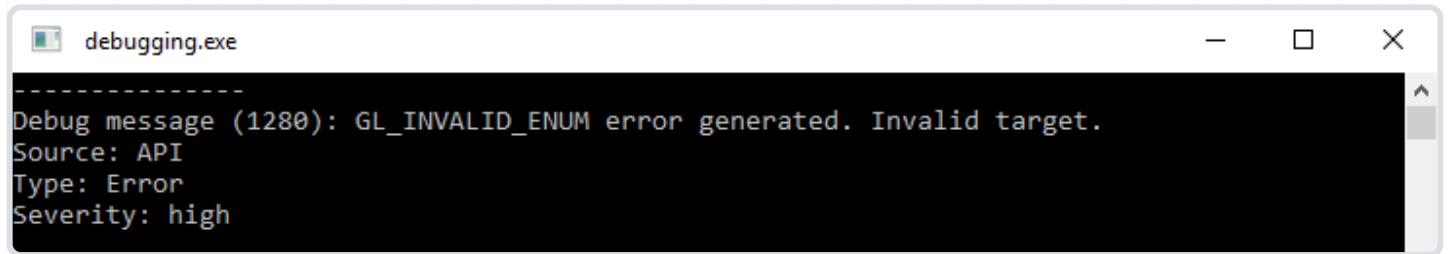
当调试输出检测到了一个OpenGL错误，它会调用这个回调函数，我们将可以打印出非常多的OpenGL错误信息。注意我们忽略掉了一些错误代码，这些错误代码一般不能给我们任何有用的信息（比如Nvidia驱动中的131185仅告诉我们缓冲成功创建了）。

过滤调试输出

有了`glDebugMessageControl`, 你可以潜在地过滤出你需要的错误类型。在这里我们不打算过滤任何来源, 类型或者严重等级。如果我们仅希望显示OpenGL API的高严重等级错误消息, 你可以设置为以下这样:

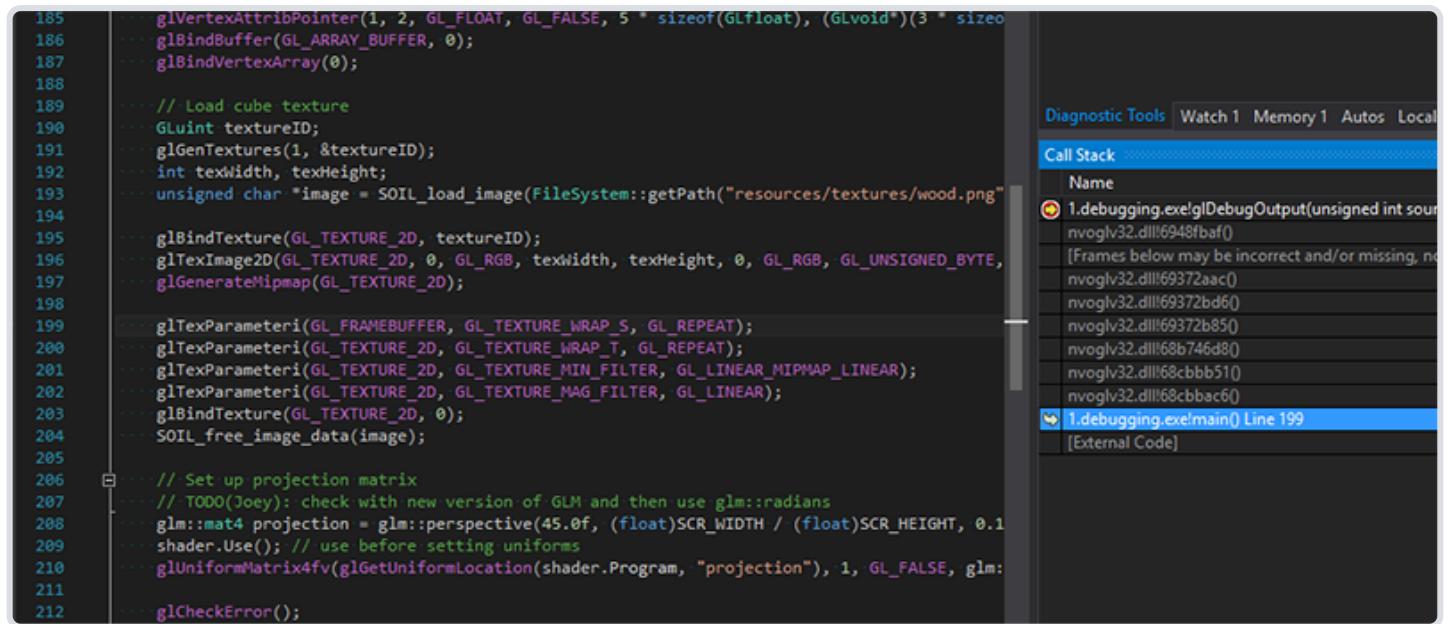
```
glDebugMessageControl(GL_DEBUG_SOURCE_API,
                      GL_DEBUG_TYPE_ERROR,
                      GL_DEBUG_SEVERITY_HIGH,
                      0, nullptr, GL_TRUE);
```

有了我们的配置, 如果你的上下文支持调试输出的话, 每个不正确的OpenGL指令都会打印出一大堆的有用数据。



回溯调试错误源

使用调试输出另一个很棒的技巧就是你可以很容易找出错误发生的准确行号或者调用。通过在`DebugOutput`中特定的错误类型上 (或者在函数的顶部, 如果你不关心类型的话) 设置一个断点, 调试器将会捕捉到抛出的错误, 你可以往上查找调用栈直到找到消息发出的源头。



这需要一些手动操作, 但如果你大致知道你在寻找什么, 这会非常有用, 能够帮助你快速定位错误。

自定义错误输出

除了仅仅是阅读信息，我们也可以使用`glDebugMessageInsert`将信息推送到调试输出系统：

```
glDebugMessageInsert(GL_DEBUG_SOURCE_APPLICATION, GL_DEBUG_TYPE_ERROR, 0,
                     GL_DEBUG_SEVERITY_MEDIUM, -1, "error message here");
```

如果你正在利用其它使用调试输出上下文的程序或OpenGL代码进行开发，这会非常有用。其它的开发者能快速了解你自定义OpenGL代码中任何报告出来的Bug。

总而言之，调试输出（如果你能使用它）对与快速捕捉错误是非常有用的，完全值得你花一点时间来配置，它能够省下你非常多的开发时间。你可以[在这里](#)找到源码，里面`glGetError`和调试输出上下文都有配置；看看你是否能够修复所有的错误。

调试着色器输出

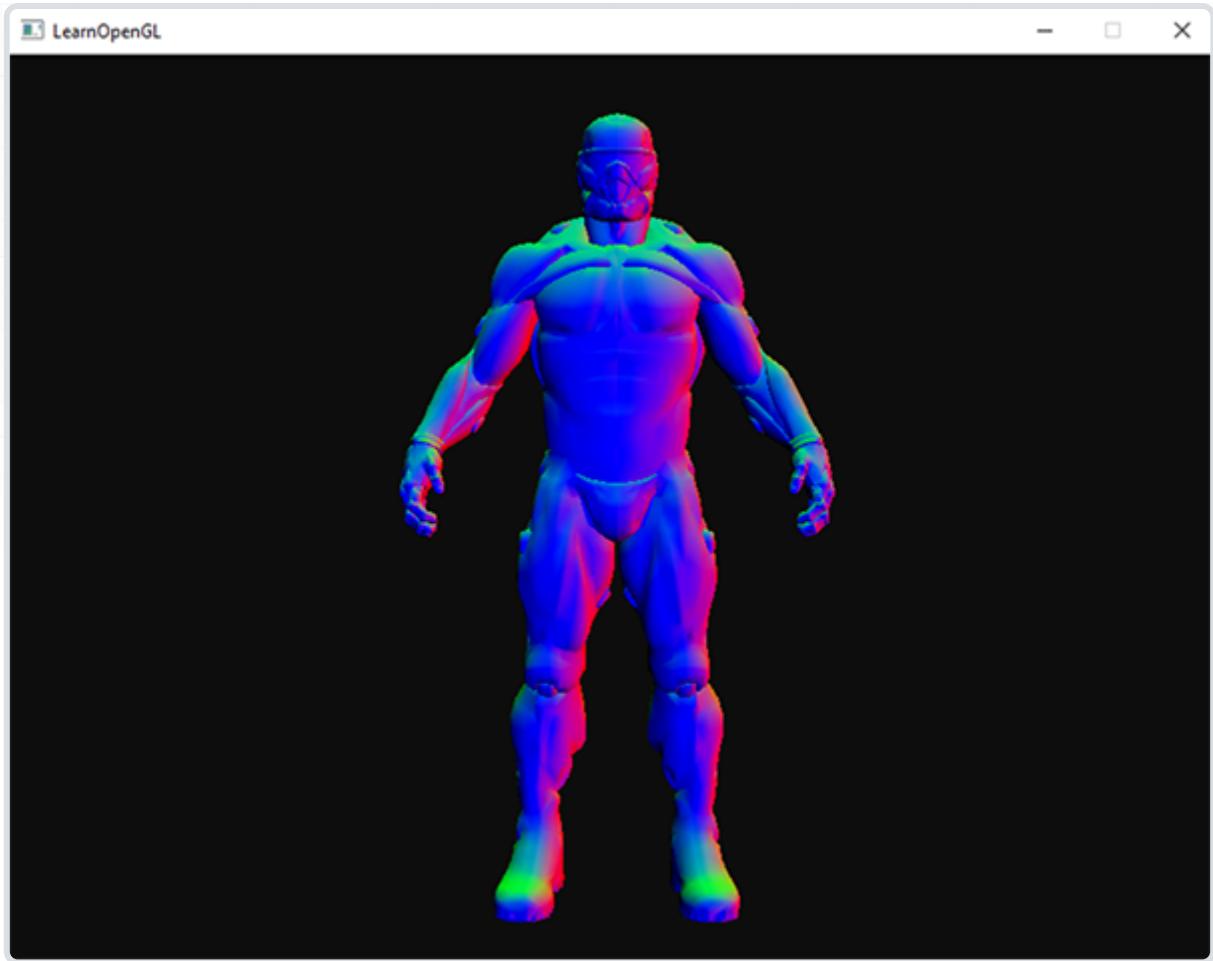
对于GLSL来说，我们不能访问像是`glGetError`这样的函数，也不能通过步进的方式运行着色器代码。如果你得到一个黑屏或者完全错误的视觉效果，通常想要知道着色器代码是否有误会非常困难。是的，我们是有编译错误报告来报告语法错误，但是捕捉语义错误又是一大难题。

一个经常使用的技巧就是将着色器程序中所有相关的变量直接发送到片段着色器的输出通道，以评估它们。通过直接输出着色器变量到输出颜色通道，我们通常可以通过观察视觉结果来获取有用的信息。比如说，如果我们想要检查一个模型的法向量是否正确，我们可以把它们（可以是变换过的也可以是没有变换过的）从顶点着色器传递到片段着色器中，在片段着色器中我们会用以下这种方式输出法向量：

```
#version 330 core
out vec4 FragColor;
in vec3 Normal;
[...]

void main()
{
    [...]
    FragColor.rgb = Normal;
    FragColor.a = 1.0f;
}
```

通过输出一个（非颜色）变量到这样子的输出颜色通道中，我们可以快速审查变量是否显示着正确的值。举例来说，如果最后的视觉效果完全是黑色的，则很清楚表明法向量没有正确地传递至着色器中。当它们都显示出来的时候，检查它们（大概）正确与否就会变得非常简单。



从视觉效果来看，我们可以看见法向量应该是正确的，因为纳米装的右侧大部分都是红色的（这表明法线大概（正确地）指向正x轴），并且类似的纳米装的前方大部分都为蓝色，即正z轴方向。

这一方法可以很容易拓展到你想要测试的任何变量。一旦你卡住了或者怀疑你的着色器有问题，尝试显示多个变量和/或中间结果，看看哪部分算法什么的没加上或者有错误。

OpenGL GLSL参考编译器

每一个驱动都有它自己的怪癖。比如说NVIDIA驱动会更宽容一点，通常会忽略一些限制或者规范，而ATI/AMD驱动则通常会严格执行OpenGL规范（在我看来会更好一点）。问题是在一台机器上的着色器到另一台机器上可能就由于驱动差异不能正常工作了。

通过多年的经验你会最终能够知道不同GPU供应商之间的细微差别，但如果你想要保证你的着色器代码在所有的机器上都能运行，你可以直接对着官方的标准使用OpenGL的GLSL参考编译器 (Reference Compiler) 来检查。你可以从[这里](#)下载所谓的GLSL语言校验器 (GLSL Lang Validator)的可执行版本，或者从[这里](#)找到完整的源码。

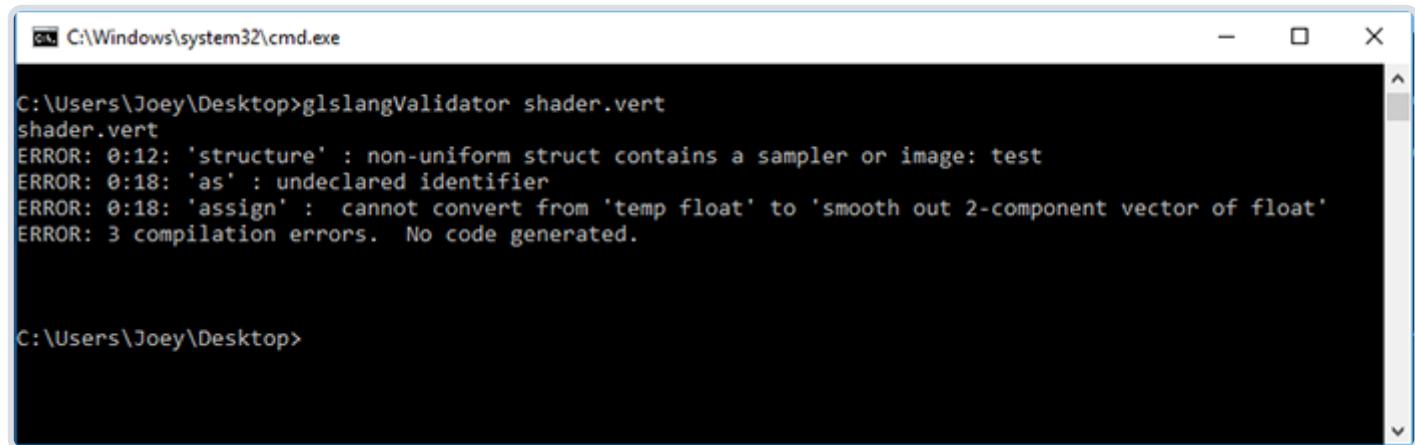
有了这个GLSL语言校验器，你可以很方便的检查你的着色器代码，只需要把着色器文件作为程序的第一个参数即可。注意GLSL语言校验器是通过下列固定的后缀名来决定着色器的类型的：

- **.vert**: 顶点着色器(Vertex Shader)
- **.frag**: 片段着色器(Fragment Shader)
- **.geom**: 几何着色器(Geometry Shader)
- **.tesc**: 细分控制着色器(Tessellation Control Shader)
- **.tese**: 细分计算着色器(Tessellation Evaluation Shader)
- **.comp**: 计算着色器(Compute Shader)

运行GLSL参考编译器非常简单：

```
glslangvalidator shaderFile.vert
```

注意如果没有检测到错误的话则没有输出。对一个不正确的顶点着色器使用GLSL参考编译器进行测试会输出以下结果：



```
C:\Windows\system32\cmd.exe
C:\Users\Joey\Desktop>glslangValidator shader.vert
shader.vert
ERROR: 0:12: 'structure' : non-uniform struct contains a sampler or image: test
ERROR: 0:18: 'as' : undeclared identifier
ERROR: 0:18: 'assign' : cannot convert from 'temp float' to 'smooth out 2-component vector of float'
ERROR: 3 compilation errors. No code generated.

C:\Users\Joey\Desktop>
```

它不会显示AMD，NVidia，以及Intel的GLSL编译器之间的细微差别，也不能保证你的着色器完全没有Bug，但它至少能够帮你对着直接的GLSL规范进行检查。

帧缓冲输出

你的调试工具箱中另外一个技巧就是在OpenGL程序中一块特定区域显示帧缓冲的内容。你可能会比较频繁地使用[帧缓冲](#)，但由于帧缓冲的魔法通常在幕后进行，有时候想要知道出什么问题会非常困难。在你的程序中显示帧缓冲的内容是一个很有用的技巧，帮助你快速检查错误。

注意，这里讨论的帧缓冲显示内容（附件）仅能在纹理附件上使用，而不能应用于渲染缓冲对象。

通过使用一个非常简单，只显示纹理的着色器，我们可以写一个助手函数快速在屏幕右上角显示任何纹理。

```
// 顶点着色器
#version 330 core
layout (location = 0) in vec2 position;
layout (location = 1) in vec2 texCoords;

out vec2 TexCoords;

void main()
{
    gl_Position = vec4(position, 0.0f, 1.0f);
    TexCoords = texCoords;
}

// 片段着色器
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D fboAttachment;

void main()
{
    FragColor = texture(fboAttachment, TexCoords);
}
```

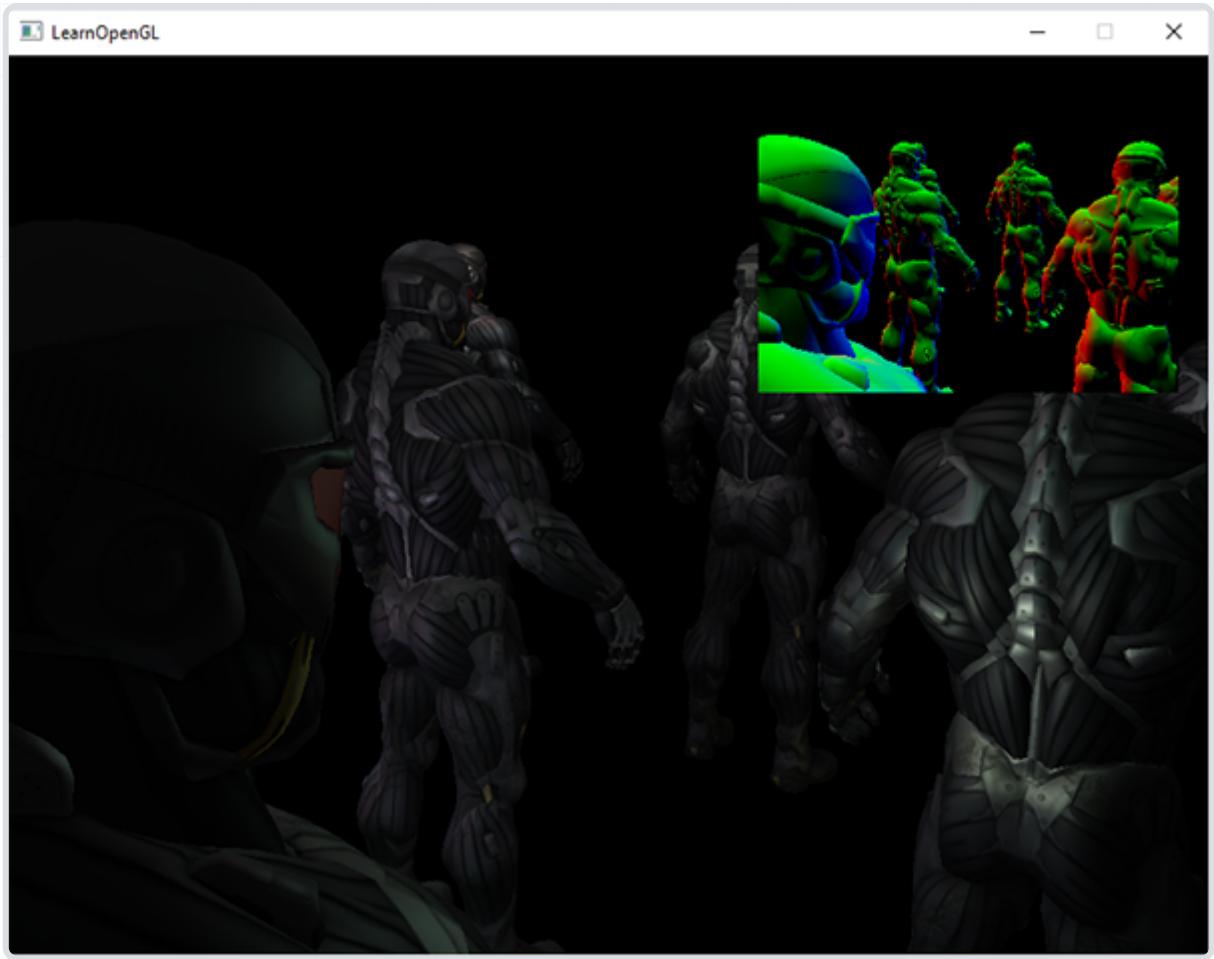
```
void DisplayFramebufferTexture(GLuint textureID)
{
    if(!notInitialized)
    {
        // 在屏幕右上角，使用NDC顶点坐标初始化着色器和VAO
        [...]
    }

    glBindTexture(GL_TEXTURE_2D, textureID);
    glBindVertexArray(vaoDebugTexturedRect);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);
    glUseProgram(0);
}

int main()
{
    [...]
    while (!glfwWindowShouldClose(window))
    {
        [...]
        DisplayFramebufferTexture(fboAttachment0);

        glfwSwapBuffers(window);
    }
}
```

这将在屏幕右上角给你一个小窗口，用来调试帧缓冲的输出。比如你想要检查延迟渲染器的几何渲染阶段中的法向量是否正确，使用这个会非常方便：



你当然可以拓展这个函数以支持渲染更多的纹理。这个方法能够非常快速地让你对帧缓冲内容有着持续的反馈。

外部调试软件

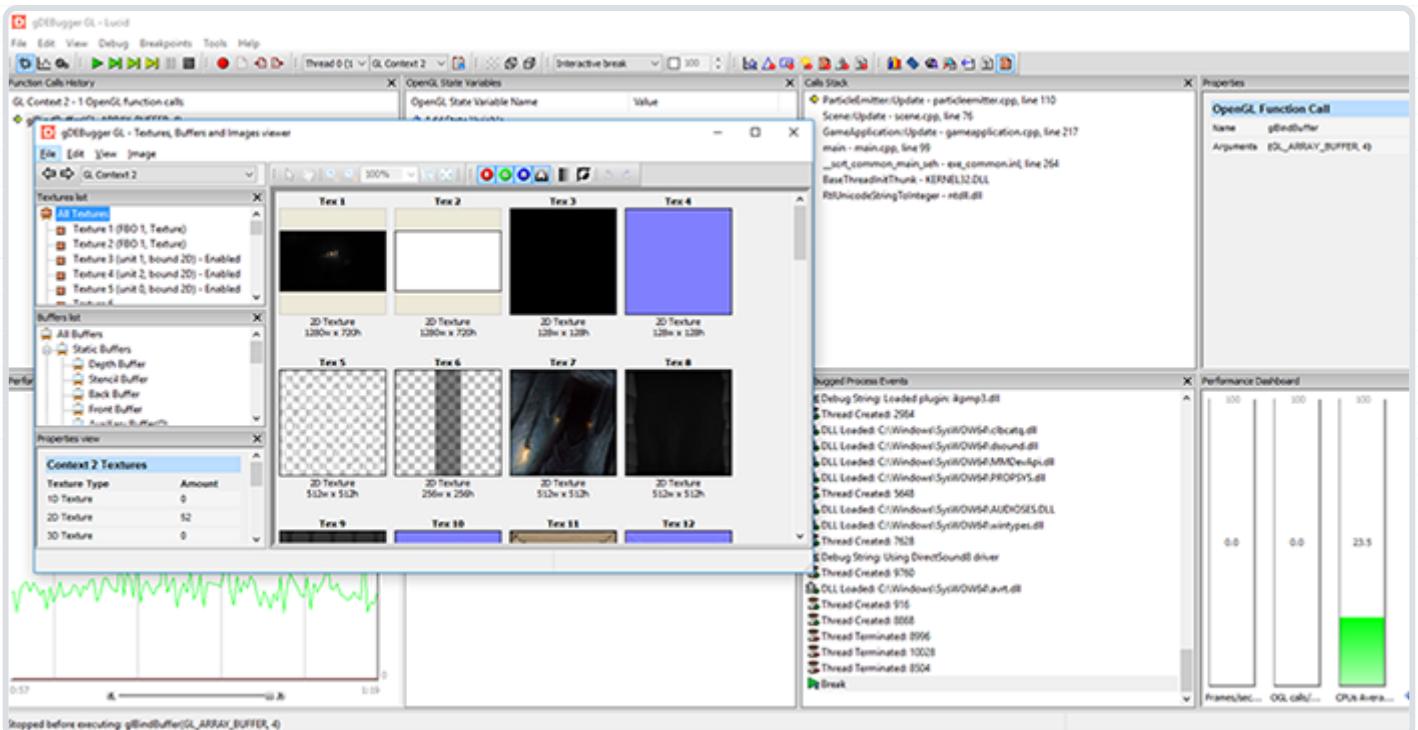
当上面所有介绍到的技巧都不能使用的时候，我们仍可以使用一个第三方的工具来帮助我们调试。第三方应用通常将它们自己注入到OpenGL驱动中，并且能够拦截各种OpenGL调用，给你大量有用的数据。这些工具可以在很多方面帮助到你：对OpenGL函数使用进行性能测试，寻找瓶颈，检查缓冲内存，显示纹理和帧缓冲附件。如果你正在写（大规模）产品代码，这类的工具在开发过程中是非常有用的。

我在下面列出了一些流行的调试工具，选几个尝试一下，看看哪个最适合你。

gDebugger

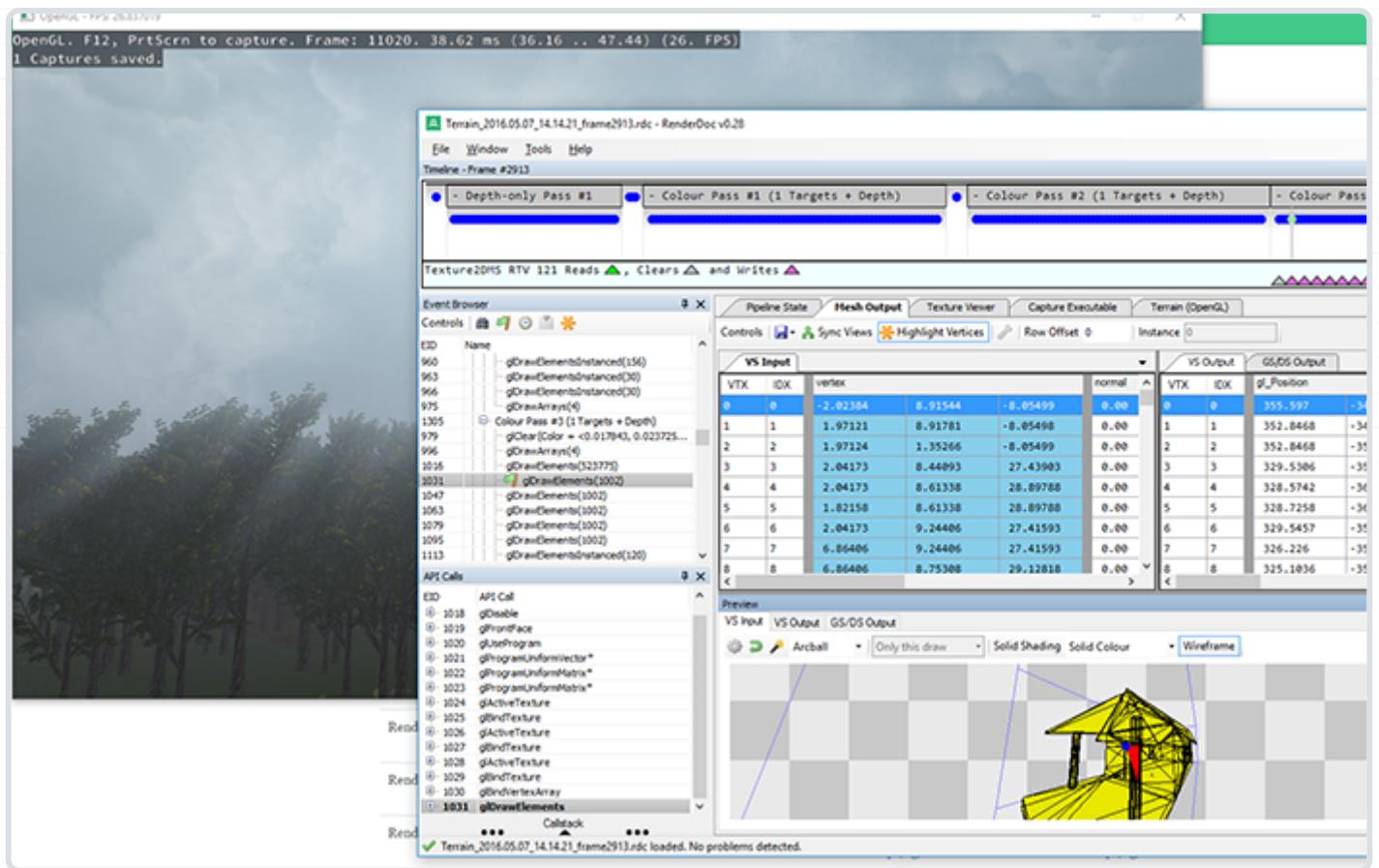
gDebugger是一个非常易用的跨平台OpenGL程序调试工具。gDebugger会在你运行的OpenGL程序边上，提供OpenGL状态的详细概况。你可以随时暂停程序来检查当前状态，纹理内容以及缓冲使用。你可以在[这里](#)下载gDebugger。

运行gDebugger只需要打开程序，创建一个工程，给它你OpenGL程序的位置与工作目录即可。



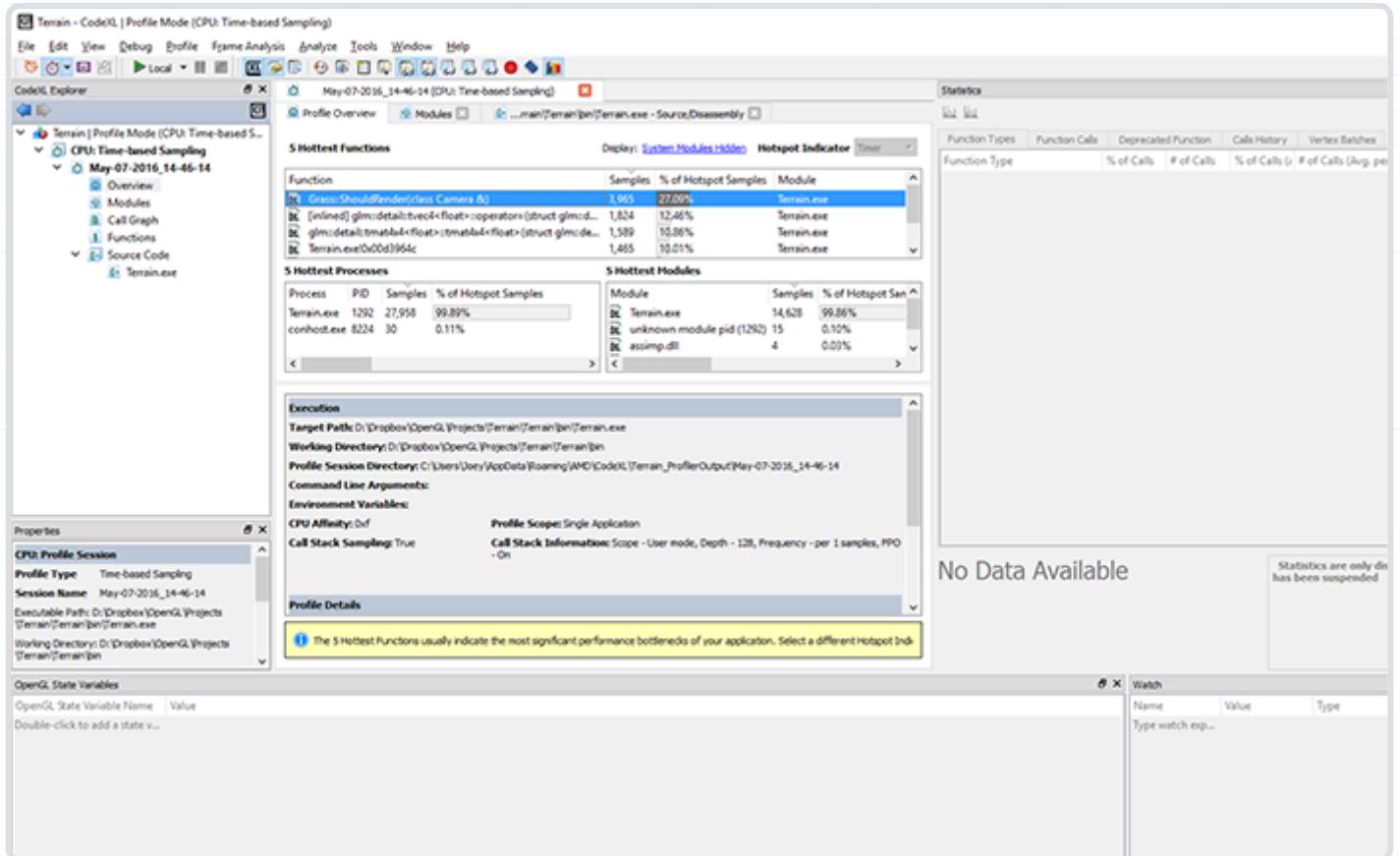
RenderDoc

RenderDoc是另外一个很棒的（完全开源的）独立调试工具。和gDebugger类似，你只需要设置捕捉的程序以及工作目录就行了。你的程序会正常运行，当你想要检查一个特定的帧的时候，你只需要让RenderDoc在程序当前状态下捕捉一个或多个帧即可。在捕捉的帧当中，你可以观察管线状态，所有OpenGL指令，缓冲储存，以及使用的纹理。



CodeXL

[CodeXL](#)是由AMD开发的一款GPU调试工具，它有独立版本也有Visual Studio插件版本。CodeXL可以给你非常多的信息，对于图形程序的性能测试也非常有用。CodeXL在Nvidia与Intel的显卡上也能运行，不过会不支持OpenCL调试。

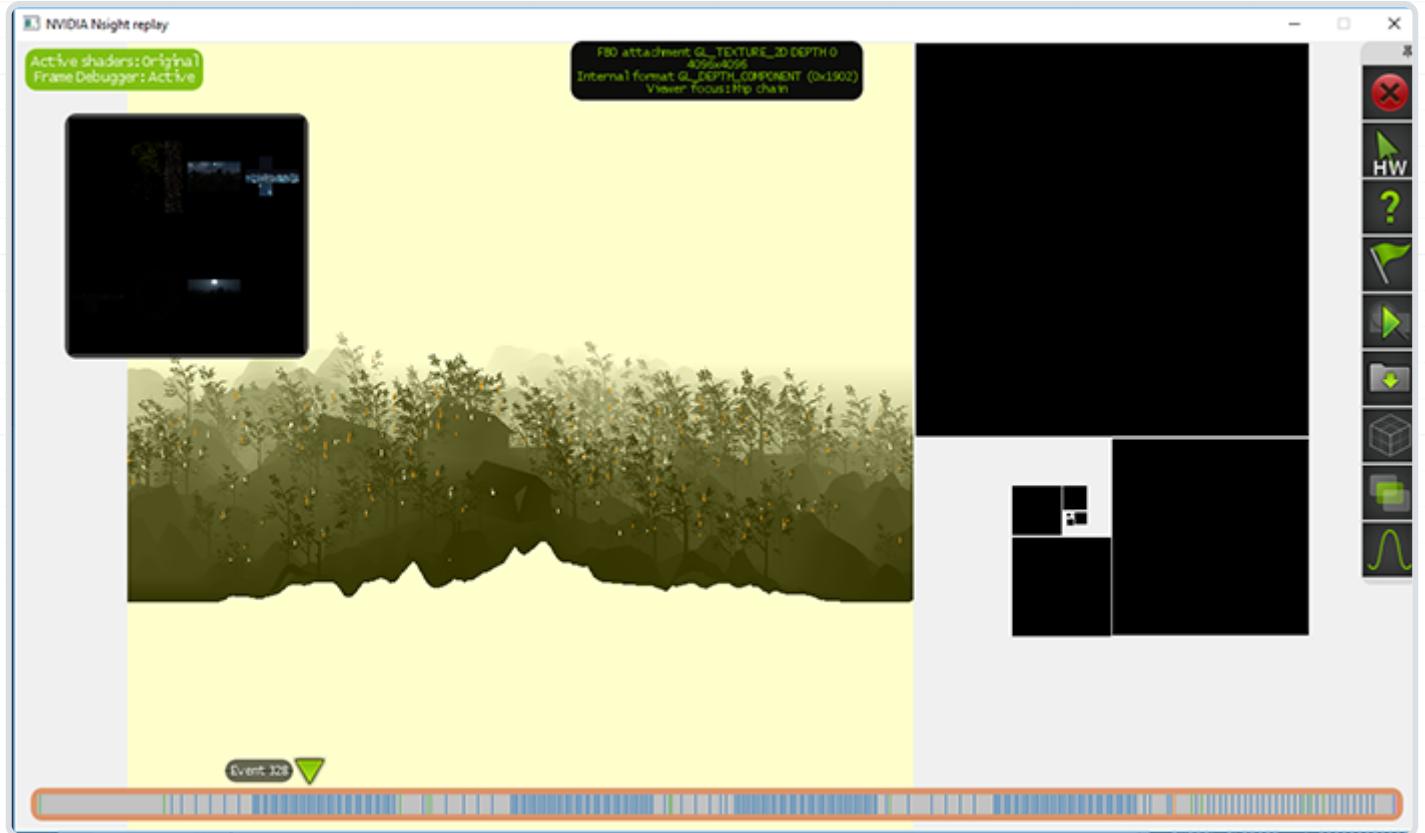


我没有太多的CodeXL使用经验，我个人觉得gDebugger和RenderDoc会更容易使用一点，但我仍把它列在这里，因为它仍是一个非常可靠的工具，并且主要是由最大的GPU制造商之一AMD开发的。

NVIDIA Nsight

NVIDIA流行的Nsight GPU调试工具并不是一个独立程序，而是一个Visual Studio IDE或者Eclipse IDE的插件。Nsight插件对图形开发者来说非常容易使用，因为它给出了GPU用量，逐帧GPU状态大量运行时的统计数据。

当你在Visual Studio（或Eclipse）使用Nsight的调试或者性能测试指令启动程序的时候，Nsight将会在程序自身中运行。Nsight非常棒的一点就是它在你的程序中渲染了一套GUI系统，你可以使用它获取你程序各种各样的有用的信息，可以是运行时也可以是逐帧分析。



Nsight是一款非常有用的工具，在我看来比上述其它工具都有更好的表现，但它仍有一个非常重要的缺点，它只能在NVIDIA的显卡上工作。如果你正在使用一款NVIDIA的显卡（并且使用Visual Studio），Nsight是非常值得一试的。

我知道我可能遗漏了一些其它的调试工具（比如我还能想到有Valve的[VOGL](#)和[APItrace](#)），但我觉得这个列表已经给你足够多的工具来实验了。我并不是之前提到的任何一个工具的专家，所以如果我在哪里讲错了请在评论区留言，我会很乐意改正。

附加资源

- [为什么你的代码会产生一个黑色窗口](#)：Reto Koradi列举了你的屏幕没有产生任何输出的可能原因。
- [调试输出](#)：Vallentin Source写的一份非常详细的调试输出教程，里面有在多个窗口系统中配置调试上下文的详细信息。

2.8.2 文本渲染

原文 [Text Rendering](#)

作者 JoeyDeVries

翻译 [Geequlim](#)

校对 ggy_1992, [BLumia](#)

当你在图形计算领域冒险到了一定阶段以后你可能会想使用OpenGL来绘制文本。然而，可能与你想象的并不一样，使用像OpenGL这样的底层库来把文本渲染到屏幕上并不是一件简单的事情。如果你只需要绘制128种不同的字符(Character)，那么事情可能会简单一些。但是如果你要绘制的字符有着不同的宽、高和边距，事情马上就复杂了。根据你使用语言的不同，你可能会需要多于128个字符。再者，如果你要绘制音乐符、数学符号这些特殊的符号；或者渲染竖排文本呢？一旦你把文本这些复杂的情况考虑进来，你就不会奇怪为什么OpenGL这样的底层API没有包含文本处理了。

由于OpenGL本身并没有包含任何的文本处理能力，我们必须自己定义一套全新的系统让OpenGL绘制文本到屏幕上。由于文本字符没有图元，我们必须有点创造力才行。需要使用的一些技术可以是：通过[GL_LINES](#)来绘制字形，创建文本的3D网格(Mesh)，或在3D环境中将字符纹理渲染到2D四边形上。

开发者最常用的一种方式是将字符纹理绘制到四边形上。绘制这些纹理四边形本身其实并不是很复杂，然而检索要绘制文本的纹理却变成了一项有挑战性的工作。本教程将探索多种文本渲染的实现方法，并且使用FreeType库实现一个更加高级但更灵活的渲染文本技术。

经典文本渲染：位图字体

早期的时候，渲染文本是通过选择一个需要的字体(Font)（或者自己创建一个），并提取这个字体中所有相关的字符，将它们放到一个单独的大纹理中来实现的。这样一张纹理叫做位图字体(Bitmap Font)，它在纹理的预定义区域中包含了我们想要使用的所有字符。字体的这些字符被称为字形(Glyph)。每个字形都关联着一个特定的纹理坐标区域。当你想要渲染一个字符的时候，你只需要通过渲染这一块特定的位图字体区域到2D四边形上即可。



你可以看到，我们取一张位图字体，（通过仔细选择纹理坐标）从纹理中采样对应的字形，并渲染它们到多个2D四边形上，最终渲染出“OpenGL”文本。通过启用[混合](#)，让背景保持透明，最终就能渲染一个字符串到屏幕上。这个位图字体是通过Codehead的位图[字体生成器](#)生成的。

使用这种方式绘制文本有许多优势也有很多缺点。首先，它相对来说很容易实现，并且因为位图字体已经预光栅化了，它的效率也很高。然而，这种方式不够灵活。当你想要使用不同的字体时，你需要重新编译一套全新的位图字体，而且你的程序会被限制在一个固定的分辨率。如果你对这些文本进行缩放的话你会看到文本的像素边缘。此外，这种方式通常会局限于非常小的字符集，如果你想让它来支持Extended或者Unicode字符的话就很不现实了。

这种绘制文本的方式曾经得益于它的高速和可移植性而非常流行，然而现在已经出现更加灵活的方式。其中一个是我们即将讨论的使用FreeType库来加载TrueType字体的方式。

现代文本渲染：FreeType

FreeType是一个能够用于加载字体并将他们渲染到位图以及提供多种字体相关的操作的软件开发库。它是一个非常受欢迎的跨平台字体库，它被用于Mac OS X、Java、PlayStation主机、Linux、Android等平台。FreeType的真正吸引力在于它能够加载TrueType字体。

TrueType字体不是用像素或其他不可缩放的方式来定义的，它是通过数学公式（曲线的组合）来定义的。类似于矢量图像，这些光栅化后的字体图像可以根据需要的字体高度来生成。通过使用TrueType字体，你可以轻易渲染不同大小的字形而不造成任何质量损失。

FreeType可以在他们的[官方网站](#)中下载到。你可以选择自己用源码编译这个库，如果支持你的平台的话，你也可以使用他们预编译好的库。请确认你将freetype.lib添加到你项目的链接库中，并且确认编译器知道头文件的位置。

然后请确认包含合适的头文件：

```
#include <ft2build.h>
#include FT_FREETYPE_H
```

由于FreeType的开发方式（至少在我写这篇文章的时候），你不能将它们的头文件放到一个新的目录下。它们应该保存在你include目录的根目录下。通过使用像 `#include <FreeType/ft2build.h>` 这样的方式导入FreeType可能会出现一些头文件冲突的问题。

FreeType所做的事就是加载TrueType字体并为每一个字形生成位图以及计算几个度量值(Metric)。我们可以提取出它生成的位图作为字形的纹理，并使用这些度量值定位字符的字形。

要加载一个字体，我们只需要初始化FreeType库，并且将这个字体加载为一个FreeType称之为面(Face)的东西。这里为我们加载一个从Windows/Fonts目录中拷贝来的TrueType字体文件arial.ttf。

```
FT_Library ft;
if (FT_Init_FreeType(&ft))
    std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;

FT_Face face;
if (FT_New_Face(ft, "fonts/arial.ttf", 0, &face))
    std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;
```

这些FreeType函数在出现错误时将返回一个非零的整数值。

当面加载完成之后，我们需要定义字体大小，这表示着我们要从字体面中生成多大的字形：

```
FT_Set_Pixel_Sizes(face, 0, 48);
```

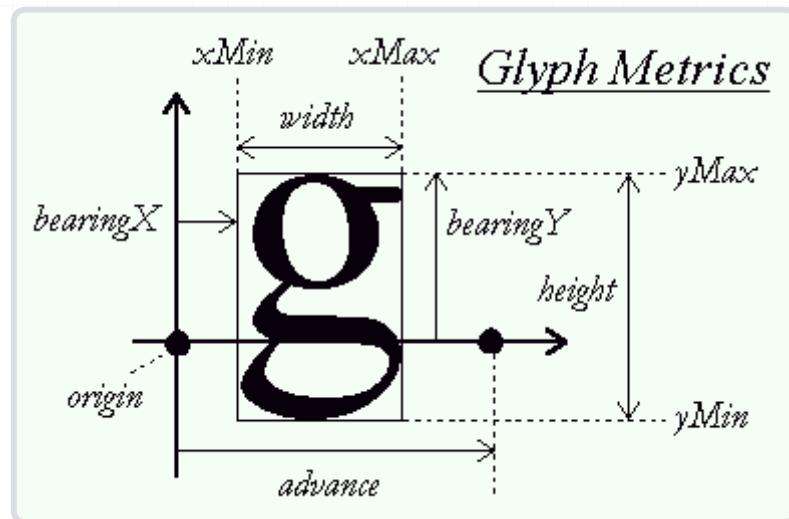
此函数设置了字体面的宽度和高度，将宽度值设为0表示我们要从字体面通过给定的高度中动态计算出字形的宽度。

一个FreeType面中包含了一个字形的集合。我们可以调用`FT_Load_Char`函数来将其中一个字形设置为激活字形。这里我们选择加载字符字形'X'：

```
if (FT_Load_Char(face, 'X', FT_LOAD_RENDER))
    std::cout << "ERROR::FREETYPE: Failed to load Glyph" << std::endl;
```

通过将`FT_LOAD_RENDER`设为加载标记之一，我们告诉FreeType去创建一个8位的灰度位图，我们可以通过`face->glyph->bitmap`来访问这个位图。

使用FreeType加载的每个字形没有相同的大小（不像位图字体那样）。使用FreeType生成的位图的大小恰好能包含这个字符可见区域。例如生成用于表示'.'的位图的大小要比表示'X'的小得多。因此，FreeType同样也加载了一些度量值来指定每个字符的大小和位置。下面这张图展示了FreeType对每一个字符字形计算的所有度量值。



每一个字形都放在一个水平的**基准线**(Baseline)上（即上图中水平箭头指示的那条线）。一些字形恰好位于基准线上（如'X'），而另一些则会稍微越过基准线以下（如'g'或'p'）（译注：即这些带有下伸部的字母，可以见[这里](#)）。这些度量值精确定义了摆放字形所需的每个字形距离基准线的偏移量，每个字形的大小，以及需要预留多少空间来渲染下一个字形。下面这个表列出了我们需要的所有属性。

属性	获取方式	生成位图描述
<code>width</code>	<code>face->glyph->bitmap.width</code>	位图宽度（像素）
<code>height</code>	<code>face->glyph->bitmap.rows</code>	位图高度（像素）
<code>bearingX</code>	<code>face->glyph->bitmap_left</code>	水平距离，即位图相对于原点的水平位置（像素）
<code>bearingY</code>	<code>face->glyph->bitmap_top</code>	垂直距离，即位图相对于基准线的垂直位置（像素）
<code>advance</code>	<code>face->glyph->advance.x</code>	水平预留值，即原点到下一个字形原点的水平距离（单位：1/64像素）

在需要渲染字符时，我们可以加载一个字符字形，获取它的度量值，并生成一个纹理，但每一帧都这样做会非常没有效率。我们应将这些生成的数据储存在程序的某一个地方，在需要渲染字符的时候再去调用。我们会定义一个非常方便的结构体，并将这些结构体存储在一个map中。

```
struct Character {
    GLuint TextureID; // 字形纹理的ID
    glm::ivec2 Size; // 字形大小
    glm::ivec2 Bearing; // 从基准线到字形左部/顶部的偏移值
    GLuint Advance; // 原点距下一个字形原点的距离
};

std::map<GLchar, Character> Characters;
```

对于这个教程来说，本着让一切简单的目的，我们只生成ASCII字符集的前128个字符。对每一个字符，我们生成一个纹理并保存相关数据至Character结构体中，之后再添加至characters这个映射表中。这样子，渲染一个字符所需的所有数据就都被储存下来备用。了。

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); //禁用字节对齐限制
for (GLubyte c = 0; c < 128; c++)
{
    // 加载字符的字形
    if (FT_Load_Char(face, c, FT_LOAD_RENDER))
    {
        std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;
        continue;
    }
    // 生成纹理
    GLuint texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(
        GL_TEXTURE_2D,
        0,
        GL_RED,
        face->glyph->bitmap.width,
        face->glyph->bitmap.rows,
        0,
        GL_RED,
        GL_UNSIGNED_BYTE,
        face->glyph->bitmap.buffer
    );
    // 设置纹理选项
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // 储存字符供之后使用
    Character character = {
        texture,
        glm::ivec2(face->glyph->bitmap.width, face->glyph->bitmap.rows),
        glm::ivec2(face->glyph->bitmap_left, face->glyph->bitmap_top),
        face->glyph->advance.x
    };
    Characters.insert(std::pair<GLchar, Character>(c, character));
}
```

在这个for循环中我们遍历了ASCII集中的全部128个字符，并获取它们对应的字符字形。对每一个字符，我们生成了一个纹理，设置了它的选项，并储存了它的度量值。有趣的是我们这里将纹理的internalFormat和format设置为GL_RED。通过字形生成的位图是一个8位灰度图，它的每一个颜色都由一个字节来表示。因此我们需要将位图缓冲的每一字节都作为纹理的颜色值。这是通过创建一个特殊的纹理实现的，这个纹理的每一字节都对应着纹理颜色的红色分量（颜色向量的第一个字节）。如果我们使用一个字节来表示纹理的颜色，我们需要注意OpenGL的一个限制：

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

OpenGL要求所有的纹理都是4字节对齐的，即纹理的大小永远是4字节的倍数。通常这并不会出现什么问题，因为大部分纹理的宽度都为4的倍数并/或每像素使用4个字节，但是现在我们每个像素只用了一个字节，它可以是任意的宽度。通过将纹理解压对齐参数设为1，这样才能确保不会有对齐问题（它可能会造成段错误）。

当你处理完字形后不要忘记清理FreeType的资源。

```
FT_Done_Face(face);
FT_Done_FreeType(ft);
```

着色器

我们将使用下面的顶点着色器来渲染字形：

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 pos, vec2 tex>
out vec2 TexCoords;

uniform mat4 projection;

void main()
{
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    TexCoords = vertex.zw;
}
```

我们将位置和纹理纹理坐标的数据合起来存在一个`vec4`中。这个顶点着色器将位置坐标与一个投影矩阵相乘，并将纹理坐标传递给片段着色器：

```
#version 330 core
in vec2 TexCoords;
out vec4 color;

uniform sampler2D text;
uniform vec3 textColor;

void main()
{
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);
    color = vec4(textColor, 1.0) * sampled;
}
```

片段着色器有两个uniform变量：一个是单颜色通道的字形位图纹理，另一个是颜色uniform，它可以用来调整文本的最终颜色。我们首先从位图纹理中采样颜色值，由于纹理数据中仅存储着红色分量，我们就采样纹理的r分量来作为取样的alpha值。通过变换颜色的alpha值，最终的颜色在字形背景颜色上会是透明的，而在真正的字符像素上是不透明的。我们也将RGB颜色与`textColor`这个uniform相乘，来变换文本颜色。

当然我们需要启用混合才能让这一切行之有效：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

对于投影矩阵，我们将使用一个正射投影矩阵(Orthographic Projection Matrix)。对于文本渲染我们（通常）都不需要透视，使用正射投影同样允许我们在屏幕坐标系中设定所有的顶点坐标，如果我们使用如下方式配置：

```
glm::mat4 projection = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f);
```

我们设置投影矩阵的底部参数为`0.0f`，并将顶部参数设置为窗口的高度。这样做的结果是我们指定了y坐标的范围为屏幕底部(0.0f)至屏幕顶部(600.0f)。这意味着现在点(0.0, 0.0)对应左下角（译注：而不再是窗口正中间）。

最后要做的事是创建一个VBO和VAO用来渲染四边形。现在我们在初始化VBO时分配足够的内存，这样我们可以在渲染字符的时候再来更新VBO的内存。

```
GLuint VAO, VBO;
 glGenVertexArrays(1, &VA0);
 glGenBuffers(1, &VBO);
 glBindVertexArray(VAO);
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 6 * 4, NULL, GL_DYNAMIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), 0);
 glBindBuffer(GL_ARRAY_BUFFER, 0);
 glBindVertexArray(0);
```

每个2D四边形需要6个顶点，每个顶点又是由一个4float向量（译注：一个纹理坐标和一个顶点坐标）组成，因此我们将VBO的内存分配为 $6 * 4$ 个float的大小。由于我们会在绘制字符时经常更新VBO的内存，所以我们将内存类型设置为[GL_DYNAMIC_DRAW](#)。

渲染一行文本

要渲染一个字符，我们从之前创建的[Characters](#)映射表中取出对应的[Character](#)结构体，并根据字符的度量值来计算四边形的维度。根据四边形的维度我们就能动态计算出6个描述四边形的顶点，并使用[glBufferSubData](#)函数更新VBO所管理内存的内容。

我们创建一个叫做`RenderText`的函数渲染一个字符串：

```
void RenderText(Shader &s, std::string text, GLfloat x, GLfloat y, GLfloat scale, glm::vec3 color)
{
    // 激活对应的渲染状态
    s.Use();
    glUniform3f(glGetUniformLocation(s.Program, "textColor"), color.x, color.y, color.z);
    glActiveTexture(GL_TEXTURE0);
    glBindVertexArray(VAO);

    // 遍历文本中所有的字符
    std::string::const_iterator c;
    for (c = text.begin(); c != text.end(); c++)
    {
        Character ch = Characters[*c];

        GLfloat xpos = x + ch.Bearing.x * scale;
        GLfloat ypos = y - (ch.Size.y - ch.Bearing.y) * scale;

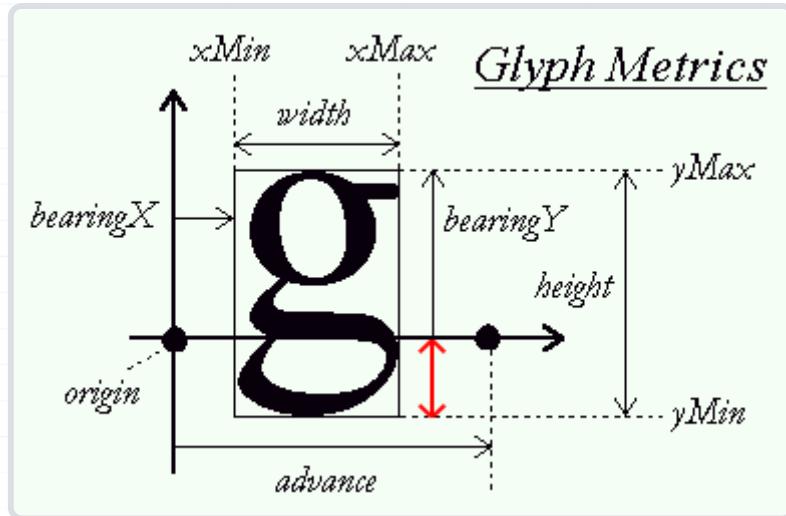
        GLfloat w = ch.Size.x * scale;
        GLfloat h = ch.Size.y * scale;
        // 对每个字符更新VBO
        GLfloat vertices[6][4] = {
            {xpos,      ypos + h,      0.0, 0.0 },
            {xpos,      ypos,          0.0, 1.0 },
            {xpos + w,  ypos,          1.0, 1.0 },
            {xpos,      ypos + h,      0.0, 0.0 },
            {xpos + w,  ypos,          1.0, 1.0 },
            {xpos + w,  ypos + h,      1.0, 0.0 }
        };
        // 在四边形上绘制字形纹理
        glBindTexture(GL_TEXTURE_2D, ch.textureID);
        // 更新VBO内存的内容
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        // 绘制四边形
        glDrawArrays(GL_TRIANGLES, 0, 6);
        // 更新位置到下一个字形的原点，注意单位是1/64像素
        x += (ch.Advance >> 6) * scale; // 位偏移6个单位来获取单位为像素的值 (2^6 = 64)
    }
    glBindVertexArray(0);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

这个函数的内容应该非常明显了：我们首先计算出四边形的原点坐标（为`xpos`和`ypos`）和它的大小（为`w`和`h`），并生成6个顶点形成这个2D四边形；注意我们将每个度量值都使用`scale`进行缩放。接下来我们更新了VBO的内容、并渲染了这个四边形。

其中这行代码需要加倍留意：

```
GLfloat ypos = y - (ch.Size.y - ch.Bearing.y);
```

一些字符（如'p'或'q'）需要被渲染到基准线以下，因此字形四边形也应该被摆放在`RenderText`的`y`值以下。`ypos`的偏移量可以从字形的度量值中得出：



要计算这段距离，即偏移量，我们需要找出字形在基准线之下延展出去的距离。在上图中这段距离用红色箭头标出。从度量值中可以看到，我们可以通过用字形的高度减去 `bearingY` 来计算这段向量的长度。对于那些正好位于基准线上的字符（如'X'），这个值正好是 0.0。而对于那些超出基准线的字符（如'g'或'j'），这个值则是正的。

如果你每件事都做对了，那么你现在已经可以使用下面的语句成功渲染字符串了：

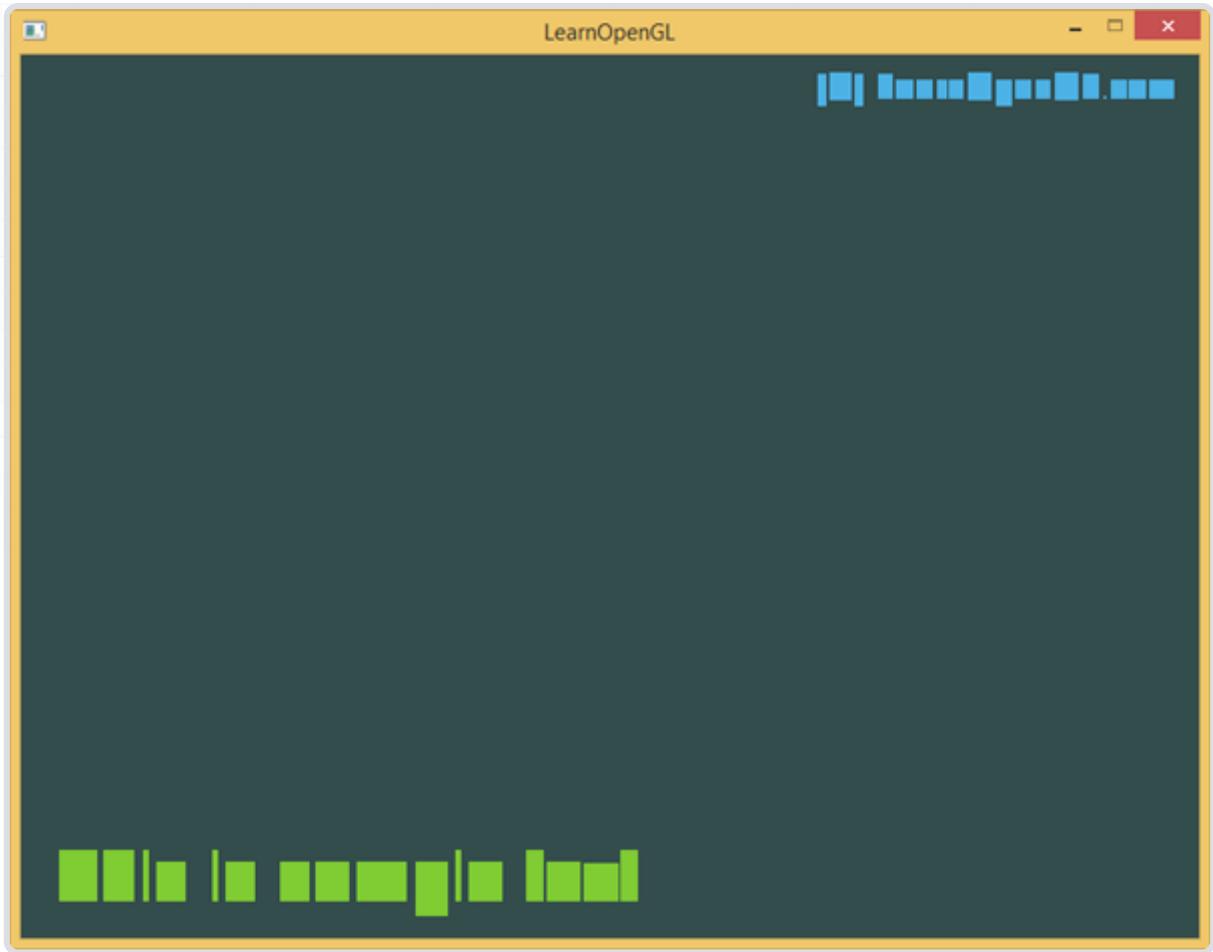
```
RenderText(shader, "This is sample text", 25.0f, 25.0f, 1.0f, glm::vec3(0.5, 0.8f, 0.2f));
RenderText(shader, "(C) LearnOpenGL.com", 540.0f, 570.0f, 0.5f, glm::vec3(0.3, 0.7f, 0.9f));
```

渲染效果看上去像这样：



你可以从[这里](#)获取这个例子的源代码。

为了让你更好理解我们是怎么计算四边形顶点的，我们可以关闭混合来看看真正渲染出来的四边形是什么样子的：



可以看到，大部分的四边形都位于一条（想象中的）基准线上，而对应'p'或'('字形的四边形则稍微向下偏移了一些。

更进一步

本教程演示了如何使用FreeType库绘制TrueType文本。这种方式灵活、可缩放并支持多种字符编码。然而，由于我们对每一个字形都生成并渲染了纹理，你的应用程序可能并不需要这么强大的功能。性能更好的位图字体也许是更可取的，因为对所有的字形我们只需要一个纹理。当然，最好的方式是结合这两种方式，动态生成包含所有字符字形的位图字体纹理，并用FreeType加载。这为渲染器节省了大量的纹理切换的开销，并且根据字形的排列紧密程度也可以节省很多的性能开销。

另一个使用FreeType字体的问题是字形纹理是储存为一个固定的字体大小的，因此直接对其放大就会出现锯齿边缘。此外，对字形进行旋转还会使它们看上去变得模糊。这个问题可以通过储存每个像素距最近的字形轮廓的距离，而不是光栅化的像素颜色，来缓解。这项技术被称为**有向距离场**(Signed Distance Fields)，Valve在几年前发表过一篇了[论文](#)，讨论了他们通过这项技术来获得非常棒的3D渲染效果。

2.8.3 2D游戏

Breakout

原文 [Breakout](#)

作者 JoeyDeVries

翻译 HHZ(qq:1158332489)

校对 Krasjet

看完前面的教程之后我们已经了解了非常多的OpenGL内部工作原理，并且我们已经能够用这些知识绘制一些复杂的图形。然而，除了之前的几个技术演示之外，我们还没有真正利用OpenGL开发一个实际应用。这篇教程为OpenGL 2D游戏制作系列教程的入门篇。这个系列教程将展示我们该如何将OpenGL应用到更大，更复杂的环境中。注意这个系列教程不一定会引入新的OpenGL概念，但会或多或少地向我们展示如何将所学的概念应用到更大的程序中去。

由于我们希望事情能够简单一点，我们会以别人开发过的一个2D街机游戏为基础，开发我们自己的2D游戏。这就是 [Breakout](#)，一个于1976年发布的运行在Atari 2600主机上的经典2D游戏。游戏要求玩家通过操控一个挡板反弹小球，以破坏所有的砖块，而不能让小球到达屏幕底端。当玩家破坏了所有砖块的时候即为完成了游戏。

下面我们可以看到Breakout原本在Atari 2600上是什么样子的：



游戏有以下几个机制：

- 玩家能够控制一个小挡板，它只能在屏幕范围内左右移动。
- 球在屏幕上运动，每一次碰撞都会使球根据碰撞位置改变运动方向。对屏幕边框，砖块，挡板都需要有碰撞判定。
- 当球运动到屏幕底部边界的时候，游戏结束或者玩家丧失一点生命值。
- 球碰到砖块的时候，砖块会被破坏。
- 当所有砖块都被破坏，玩家获胜。
- 球的方向可以通过球碰撞在挡板上的位置（即到挡板中心的距离）来操控。

由于小球会不时通过缝隙到达砖墙的上方，它会在顶部边缘与砖块层的上边缘之间不断弹跳。小球会一直持续这种状态直到最终再次进入缝隙掉下来。这也是这个游戏名字的来源，小球必须要从缝隙中逃脱(Break out)出来。

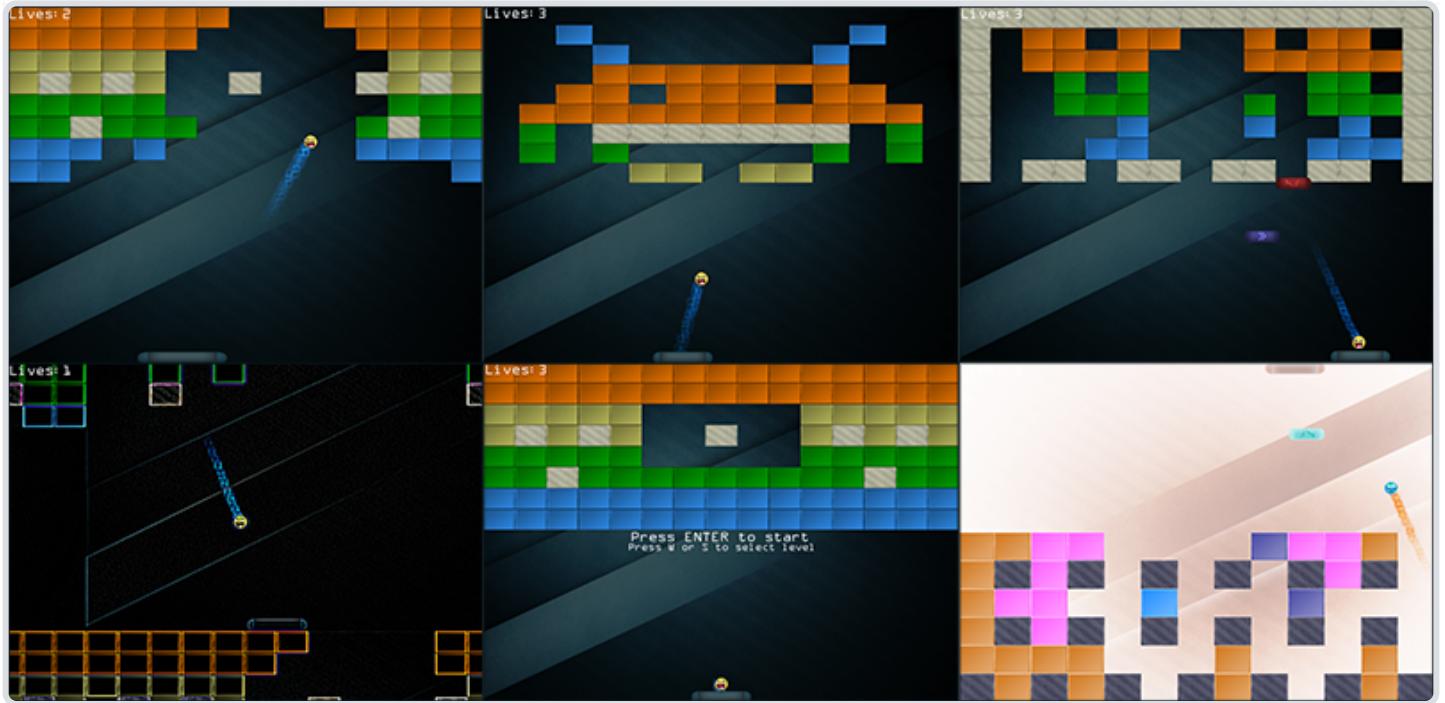
OpenGL Breakout

我们将把这个经典的街机游戏作为我们之后要完全使用OpenGL实现这个游戏的基础。这个版本的Breakout将会运行在显卡上，所以我们能够改进经典的Breakout游戏，给它加一些很棒的特性。

除了以上这些经典的机制，我们的Breakout还将拥有：

- 你见过最棒的画质
- 粒子系统
- 高级文字渲染
- 道具
- 后期处理特效
- 多个（可自定义的）关卡

为了引起大家的兴趣，你可以看看游戏完成之后的样子：



这些教程将会结合之前教程中的大量概念，并且会展示如何让它们以一个整体工作。所以，在你开始这个系列教程的学习之前，请至少完成[「入门」](#)章节。

而且，有些教程会需求其它一些教程的概念（比如说在[「高级OpenGL」](#)章节中的帧缓冲），所以当必要的时候我会列出需求的教程。

如果你已准备好开发这个游戏，可以开始[下一节](#)的学习了。

准备工作

原文 [Setting up](#)

作者 JoeyDeVries

翻译 [ZMANT](#)

校对 Krasjet

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

在开始真正写游戏机制之前，我们首先需要配置一个简单的框架，用来存放这个游戏，这个游戏将会用到几个第三方库，它们的大多数都已经在前面的教程中介绍过了。在需要用到新的库的时候，我会作出适当的介绍。

首先，我们定义一个所谓的**超级**(Uber)游戏类，它会包含所有相关的渲染和游戏代码。这个游戏类的主要作用是（简单）管理你的游戏代码，并与此同时将所有的窗口代码从游戏中解耦。这样子的话，你就可以把相同的类迁移到完全不同的窗口库（比如SDL或SFML）而不需要做太多的工作。

抽象并归纳游戏或图形代码至类与对象中有成千上万种方式。在这个系列教程中你所看到的仅是其中的一种。如果你觉得能有更好的方式进行实现，你可以尝试改进我的这个实现。

这个游戏类封装了一个初始化函数、一个更新函数、一个处理输入函数以及一个渲染函数：

```
class Game
{
public:
    // 游戏状态
    GameState State;
    GLboolean Keys[1024];
    GLuint Width, Height;
    // 构造函数/析构函数
    Game(GLuint width, GLuint height);
    ~Game();
    // 初始化游戏状态 (加载所有的着色器/纹理/关卡)
    void Init();
    // 游戏循环
    void ProcessInput(GLfloat dt);
    void Update(GLfloat dt);
    void Render();
};
```

这个类应该包含了所有在一个游戏类中会出现的东西。我们通过给定一个宽度和高度（对应于你玩游戏时的分辨率）来初始化这个游戏，并且使用`Init`函数来加载着色器、纹理并且初始化所有的游戏状态。我们可以通过调用`ProcessInput`函数，并使用存储在`Keys`数组里的数据来处理输入。并且在`Update`函数里面我们可以更新游戏设置状态（比如玩家/球的移动）。最后，我们还可以调用`Render`函数来对游戏进行渲染。注意，我们将运动逻辑与渲染逻辑分开了。

这个Game类同样封装了一个叫做state的变量，它的类型是GameState，定义如下：

```
// 代表了游戏的当前状态
enum GameState {
    GAME_ACTIVE,
    GAME_MENU,
    GAME_WIN
};
```

这个类可以帮助我们跟踪游戏的当前状态。这样的话我们就可以根据当前游戏的状态来决定渲染和/或者处理不同的元素(Item)了（比如当我们在游戏菜单界面的时候就可能需要渲染和处理不同的元素了）。

目前为止，这个游戏类的函数还完全是空的，因为我们还没有写游戏的实际代码，但这里是Game类的[头文件](#)和[代码文件](#)。

工具类

因为我们正在开发一个大型应用，所以我们将不得不频繁地重用一些OpenGL的概念，比如纹理和着色器等。因此，为这两个元素创建一个更加易用的接口也是情理之中的事了，就像在我们前面教程中创建的那个着色器类一样。

着色器类会接受两个或三个（如果有几何着色器）字符串，并生成一个编译好的着色器（如果失败的话则生成错误信息）。这个着色器类也包含很多工具(Utility)函数来帮助快速设置uniform值。纹理类会接受一个字节(Byte)数组以及宽度和高度，并（根据设定的属性）生成一个2D纹理图像。同样，这个纹理类也会封装一些工具函数。

我们并不会深入讨论这些类的实现细节，因为学到这里你应该可以很容易地理解它们是如何工作的了。出于这个原因，你可以在下面找到它们的头文件和代码文件，都有详细的注释：

- 着色器：[头文件](#), [代码](#)
- 纹理：[头文件](#), [代码](#)

注意当前的纹理类仅是为2D纹理设计的，但你很容易就可以将其扩张至更多的纹理类型。

资源管理

尽管着色器与纹理类的函数本身就很棒了，它们仍需要有一个字节数组或一些字符串来调用它们。我们可以很容易将文件加载代码嵌入到它们自己的类中，但这稍微有点违反了[单一功能原则](#)(Single Responsibility Principle)，即这两个类应当分别仅仅关注纹理或者着色器本身，而不是它们的文件加载机制。

出于这个原因，我们通常会用一个更加有组织的方法（译注：来实现文件的加载），就是创建一个所谓[资源管理器](#)的实体，专门加载游戏相关的资源。创建一个资源管理器有多种方法。在这个教程中我们选择使用一个单一实例(Singleton)的静态资源管理器，（由于它静态的本质）它在整个工程中都可以使用，它会封装所有的已加载资源以及一些相关的加载功能。

使用一个具有静态属性的单一实例类有很多优点也有很多缺点。它主要的缺点就是这样会损失OOP属性，并且丧失构造与析构的控制。不过，对于我们这种小项目来说是这些问题也很容易处理的。

和其它类的文件一样，这个资源管理器的代码如下：

- 资源管理器：[头文件](#), [代码](#)

通过使用资源管理器，我们可以很容易地把着色器加载到程序里面：

```
Shader shader = ResourceManager::LoadShader("vertex.vs", "fragment.vs", nullptr, "test");
// 接下来使用它
shader.Use();
// 或者
ResourceManager::GetShader("test").Use();
```

`Game`类、资源管理器类，以及很容易管理的`Shader`和`Texture2D`类一起组成了之后教程的基础，我们之后会广泛使用这些类来实现我们的Breakout游戏。

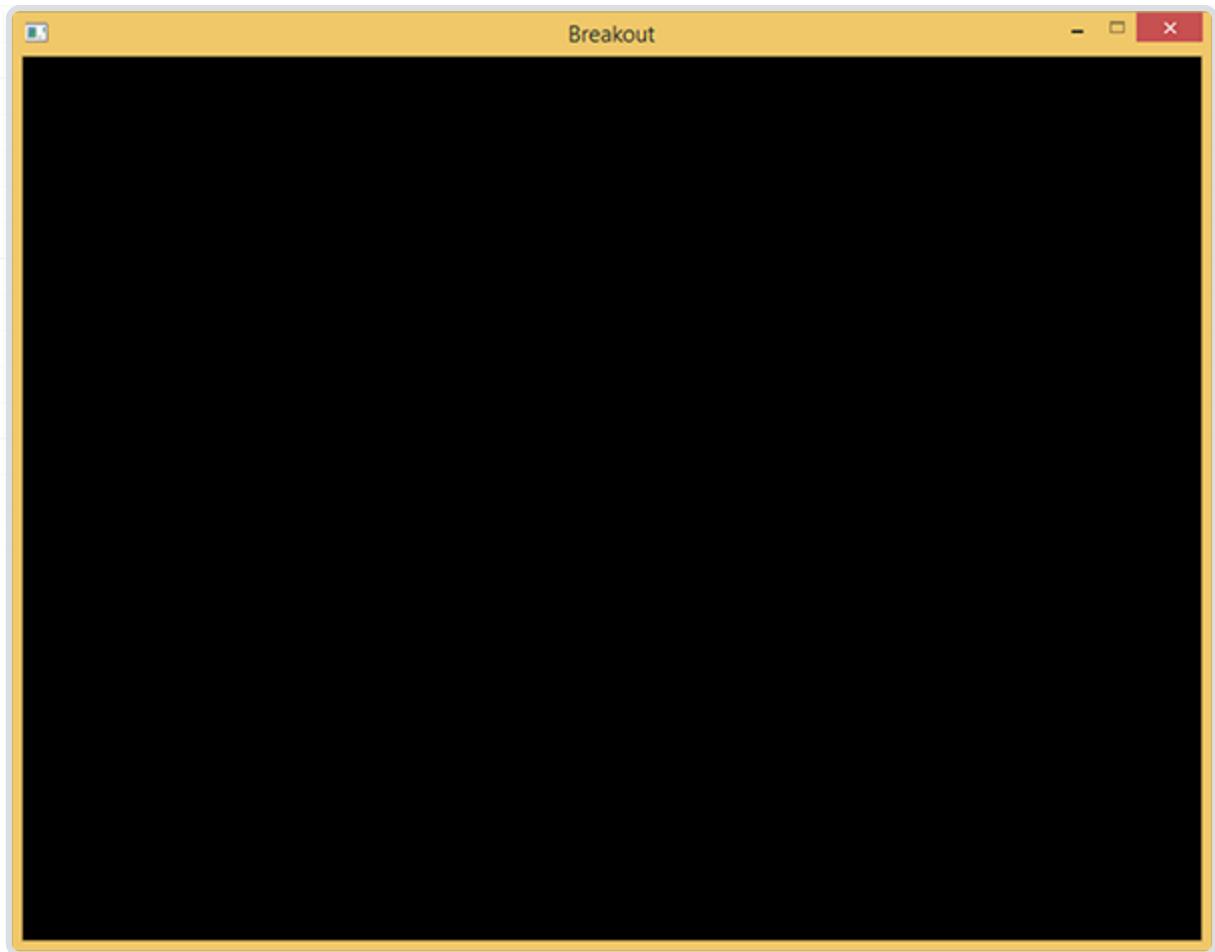
程序

我们仍然需要为这个游戏创建一个窗口并且设置一些OpenGL的初始状态。我们确保使用OpenGL的面剔除功能和混合功能。我们不需要使用深度测试，因为这个游戏完全是2D的，所有顶点都有相同的z值，所以开启深度测试并没有什么用，反而可能造成深度冲突(Z-fighting)。

这个Breakout游戏的起始代码非常简单：我们用GLFW创建一个窗口，注册一些回调函数，创建一个Game对象，并将所有相关的信息都传到游戏类中。代码如下：

- 程序：[代码](#)

运行这个代码，你应该能得到下面的输出：



现在我们已经为之后的教程构建了一个坚实的框架，我们将不断地拓展这个游戏类，封装新的功能。如果你准备好了，就可以开始[下一节](#)的学习了。

渲染精灵

原文 [Rendering Sprites](#)

作者 JoeyDeVries

翻译 [ZMANT](#)

校对 Krasjet

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

为了给我们当前这个黑漆漆的游戏世界带来一点生机，我们将会渲染一些精灵(Sprite)来填补这些空虚。**精灵**有很多种定义，但这里主要是指一个2D图片，它通常是和一些摆放相关的属性数据一起使用，比如位置、旋转角度以及二维的大小。简单来说，精灵就是那些可以在2D游戏中渲染的图像/纹理对象。

我们可以像前面大多数教程里做的那样，用顶点数据创建2D形状，将所有数据传进GPU并手动变换图形。然而，在我们这样的大型应用中，我们最好是对2D形状渲染做一些抽象化。如果我们要对每一个对象手动定义形状和变换的话，很快就会变得非常凌乱了。

在这个教程中，我们将会定义一个渲染类，让我们用最少的代码渲染大量的精灵。这样，我们就可以从散沙一样的OpenGL渲染代码中抽象出游戏代码，这也是在一个大型工程中常用的做法。虽然我们首先还要去配置一个合适的投影矩阵。

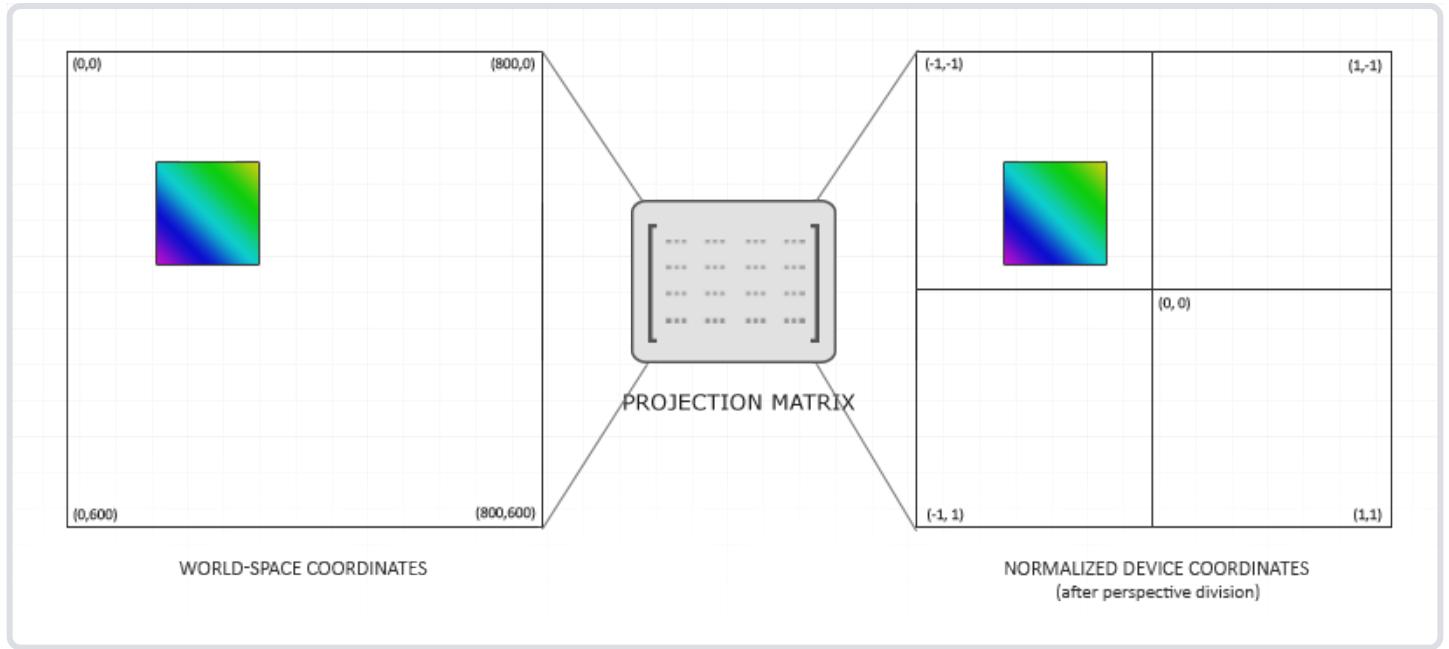
2D投影矩阵

从这个[坐标系统](#)教程中，我们明白了投影矩阵的作用是把观察空间坐标转化为标准化设备坐标。通过生成合适的投影矩阵，我们就可以在不同的坐标系下计算，这可能比把所有的坐标都指定为标准化设备坐标（再计算）要更容易处理。

我们不需要对坐标系应用透视，因为这个游戏完全是2D的，所以一个正射投影矩阵(Orthographic Projection Matrix)就可以了。由于正射投影矩阵几乎直接变换所有的坐标至裁剪空间，我们可以定义如下的投影矩阵指定世界坐标为屏幕坐标：

```
glm::mat4 projection = glm::ortho(0.0f, 800.0f, 600.0f, 0.0f, -1.0f, 1.0f);
```

前面的四个参数依次指定了投影平截头体的左、右、下、上边界。这个投影矩阵把所有在0到800之间的x坐标变换到-1到1之间，并把所有在0到600之间的y坐标变换到-1到1之间。这里我们指定了平截头体顶部的y坐标值为0，底部的y坐标值为600。所以，这个场景的左上角坐标为(0,0)，右下角坐标为(800,600)，就像屏幕坐标那样。观察空间坐标直接对应最终像素的坐标。



这样我们就可以指定所有的顶点坐标为屏幕上的像素坐标了，这对2D游戏来说相当直观。

渲染精灵

渲染一个实际的精灵应该不会太复杂。我们创建一个有纹理的四边形，它在之后可以使用一个模型矩阵来变换，然后我们会用之前定义的正射投影矩阵来投影它。

由于Breakout是一个静态的游戏，这里不需要观察/摄像机矩阵，我们可以直接使用投影矩阵把世界空间坐标变换到裁剪空间坐标。

为了变换精灵我们会使用下面这个顶点着色器：

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 position, vec2 texCoords>
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 projection;

void main()
{
    TexCoords = vertex.zw;
    gl_Position = projection * model * vec4(vertex.xy, 0.0, 1.0);
}
```

注意，我们仅用了一个`vec4`变量来存储位置和纹理坐标数据。因为位置和纹理坐标数据都只包含了两个float，所以我们可以把他们组合在一起作为一个单一的顶点属性。

片段着色器也比较直观。我们会在这里获取一个纹理和一个颜色向量，它们都会影响片段的最终颜色。我们设置了一个纹理和颜色向量，她们两个都会对像素最后的颜色产生影响。有了这个uniform颜色向量，我们就可以很方便地在游戏代码中改变精灵的颜色了。

```
#version 330 core
in vec2 TexCoords;
out vec4 color;

uniform sampler2D image;
uniform vec3 spriteColor;

void main()
{
    color = vec4(spriteColor, 1.0) * texture(image, TexCoords);
}
```

为了让精灵的渲染更加有条理，我们定义了一个SpriteRenderer类，有了它只需要一个函数就可以渲染精灵了。它的定义如下：

```
class SpriteRenderer
{
public:
    SpriteRenderer(Shader &shader);
    ~SpriteRenderer();

    void DrawSprite(Texture2D &texture, glm::vec2 position,
        glm::vec2 size = glm::vec2(10, 10), GLfloat rotate = 0.0f,
        glm::vec3 color = glm::vec3(1.0f));
private:
    Shader shader;
    GLuint quadVAO;

    void initRenderData();
};
```

SpriteRenderer类封装了一个着色器对象，一个顶点数组对象以及一个渲染和初始化函数。它的构造函数接受一个着色器对象用于之后的渲染。

初始化

首先，让我们深入研究一下负责配置quadVAO的initRenderData函数：

```
void SpriteRenderer::initRenderData()
{
    // 配置 VAO/VBO
    GLuint VBO;
    GLfloat vertices[] = {
        // 位置      // 纹理
        0.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, 0.0f, 1.0f, 0.0f
    };

    glGenVertexArrays(1, &this->quadVAO);
    glGenBuffers(1, &VBO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glBindVertexArray(this->quadVAO);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), (GLvoid*)0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

这里，我们首先定义了一组以四边形的左上角为(0,0)坐标的顶点。这意味着当我们在四边形上应用一个位移或缩放变换的时候，它们会从四边形的左上角开始进行变换。这在2D图形以及/或GUI系统中广为接受，元素的位置定义为元素左上角的位置。

接下来我们简单地向GPU传递顶点数据，并且配置顶点属性，当然在这里仅有一个顶点属性。因为所有的精灵共享着同样的顶点数据，我们只需要为这个精灵渲染器定义一个VAO就行了。

渲染

渲染精灵并不是太难；我们使用精灵渲染器的着色器，配置一个模型矩阵并且设置相关的uniform。这里最重要的就是变换的顺序：

```
void SpriteRenderer::DrawSprite(Texture2D &texture, glm::vec2 position,
    glm::vec2 size, GLfloat rotate, glm::vec3 color)
{
    // 准备变换
    this->shader.Use();
    glm::mat4 model;
    model = glm::translate(model, glm::vec3(position, 0.0f));

    model = glm::translate(model, glm::vec3(0.5f * size.x, 0.5f * size.y, 0.0f));
    model = glm::rotate(model, rotate, glm::vec3(0.0f, 0.0f, 1.0f));
    model = glm::translate(model, glm::vec3(-0.5f * size.x, -0.5f * size.y, 0.0f));

    model = glm::scale(model, glm::vec3(size, 1.0f));

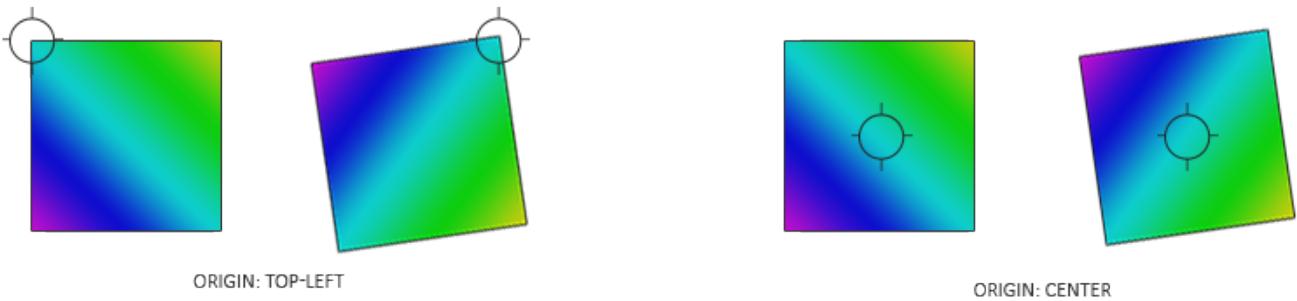
    this->shader.SetMatrix4("model", model);
    this->shader.SetVector3f("spriteColor", color);

    glActiveTexture(GL_TEXTURE0);
    texture.Bind();

    glBindVertexArray(this->quadVA0);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);
}
```

当试图在一个场景中用旋转矩阵和缩放矩阵放置一个对象的时候，建议是首先做缩放变换，再旋转，最后才是位移变换。因为矩阵乘法是从右向左执行的，所以我们变换的矩阵顺序是相反的：移动，旋转，缩放。

旋转变换可能看起来稍微有点让人望而却步。我们从[变换](#)教程里面知道旋转总是围绕原点(0,0)转动的。因为我们指定了四边形的左上角为(0,0)，所有的旋转都会围绕这个(0,0)。简单来说，在四边形左上角的**旋转原点**(Origin of Rotation)会产生不想要的结果。我们想要做的是把旋转原点移到四边形的中心，这样旋转就会围绕四边形中心而不是左上角了。我们会在旋转之前把旋转原点移动到四边形中心来解决这个问题。



因为我们首先会缩放这个四边形，我们在位移精灵的中心时还需要把精灵的大小考虑进来（这也是为什么我们乘以了精灵的size向量）。在旋转变换应用之后，我们会反转之前的平移操作。

把所有变换组合起来我们就能以任何想要的方式放置、缩放并平移每个精灵了。下面你可以找到精灵渲染器完整的源代码：

- 精灵渲染器：[头文件](#), [代码](#)

你好，精灵

有了SpriteRenderer类，我们终于能够渲染实际的图像到屏幕上上了！让我们来在游戏代码里面初始化一个精灵并且加载我们最喜爱的纹理：

```
SpriteRenderer *Renderer;
void Game::Init()
{
    // 加载着色器
    ResourceManager::LoadShader("shaders/sprite.vs", "shaders/sprite.frag", nullptr, "sprite");
    // 配置着色器
    glm::mat4 projection = glm::ortho(0.0f, static_cast<GLfloat>(this->Width),
        static_cast<GLfloat>(this->Height), 0.0f, -1.0f, 1.0f);
    ResourceManager::GetShader("sprite").Use().SetInteger("image", 0);
    ResourceManager::GetShader("sprite").SetMatrix4("projection", projection);
    // 设置专用于渲染的控制
    Renderer = new SpriteRenderer(ResourceManager::GetShader("sprite"));
    // 加载纹理
    ResourceManager::LoadTexture("textures/awesomeface.png", GL_TRUE, "face");
}
```

然后，在渲染函数里面我们就可以渲染一下我们心爱的吉祥物来检测是否一切都正常工作了：

```
void Game::Render()
{
    Renderer->DrawSprite(ResourceManager::GetTexture("face"),
        glm::vec2(200, 200), glm::vec2(300, 400), 45.0f, glm::vec3(0.0f, 1.0f, 0.0f));
}
```

这里我们把精灵放置在靠近屏幕中心的位置，它的高度会比宽度大一点。我们同样也把它旋转了45度并把它设置为绿色。注意，我们设定的精灵的位置是精灵四边形左上角的位置。

如果你一切都做对了你应该可以看到下面的结果：



你可以在[这里](#)找到更新后的游戏类源代码。

现在我们已经让渲染系统正常工作了，我们可以在[下一节](#)教程中用它来构建游戏的关卡。

关卡

原文 [Levels](#)

作者 JoeydeVries

翻译 [包纸](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

Breakout不会只是一个单一的绿色笑脸，而是一些由许多彩色砖块组成的完整关卡。我们希望这些关卡有以下特性：他们足够灵活以便于支持任意数量的行或列、可以拥有不可摧毁的坚固砖块、支持多种类型的砖块且这些信息被存储在外部文件中。

在本教程中，我们将简要介绍用于管理大量砖块的游戏关卡对象的代码，首先我们需要先定义什么是一个砖块。

我们创建一个被称为游戏对象的组件作为一个游戏内物体的基本表示。这样的游戏对象持有一些状态数据，如其位置、大小与速率。它还持有颜色、旋转、是否坚硬(不可被摧毁)、是否被摧毁的属性，除此之外，它还存储了一个Texture2D变量作为其精灵(Sprite)。

游戏中的每个物体都可以被表示为[GameObject](#)或这个类的派生类，你可以在下面找到[GameObject](#)的代码：

- [GameObject](#): [头文件](#), [代码](#)

Breakout中的关卡基本由砖块组成，因此我们可以用一个砖块的集合表示一个关卡。因为砖块需要和游戏对象几乎相同的状态，所以我们将关卡中的每个砖块表示为[GameObject](#)。[GameLevel](#)类的布局如下所示：

```
class GameLevel
{
public:
    std::vector<GameObject> Bricks;

    GameLevel() { }
    // 从文件中加载关卡
    void Load(const GLchar *file, GLuint levelWidth, GLuint levelHeight);
    // 渲染关卡
    void Draw(SpriteRenderer &renderer);
    // 检查一个关卡是否已完成 (所有非坚硬的瓷砖均被摧毁)
    GLboolean IsCompleted();
private:
    // 由砖块数据初始化关卡
    void init(std::vector<std::vector<GLuint>> tileData, GLuint levelWidth, GLuint levelHeight);
};
```

由于关卡数据从外部文本中加载，所以我们需要提出某种关卡的数据结构，以下是关卡数据在文本文件中可能的表示形式的一个例子：

```
1 1 1 1 1
2 2 0 0 2 2
3 3 4 4 3 3
```

在这里一个关卡被存储在一个矩阵结构中，每个数字代表一种类型的砖块，并以空格分隔。在关卡代码中我们可以假定每个数字代表什么：

- 数字0：无砖块，表示关卡中空的区域
- 数字1：一个坚硬的砖块，不可被摧毁
- 大于1的数字：一个可被摧毁的砖块，不同的数字区分砖块的颜色

上面的示例关卡在被[GameLevel](#)处理后，看起来会像这样：



[GameLevel](#)类使用两个函数从文件中生成一个关卡。它首先将所有数字在[Load](#)函数中加载到二维容器(vector)里，然后在[init](#)函数中处理这些数字，以创建所有的游戏对象。

```
void GameLevel::Load(const GLchar *file, GLuint levelWidth, GLuint levelHeight)
{
    // 清空过期数据
    this->Bricks.clear();
    // 从文件中加载
    GLuint tileCode;
    GameLevel level;
    std::string line;
    std::ifstream fstream(file);
    std::vector<std::vector<GLuint>> tileData;
    if (fstream)
    {
        while (std::getline(fstream, line)) // 读取关卡文件的每一行
        {
            std::istringstream sstream(line);
            std::vector<GLuint> row;
            while (sstream >> tileCode) // 读取被空格分隔的每个数字
                row.push_back(tileCode);
            tileData.push_back(row);
        }
        if (tileData.size() > 0)
            this->init(tileData, levelWidth, levelHeight);
    }
}
```

被加载后的`tileData`数据被传递到`GameLevel`的`init`函数：

```
void GameLevel::init(std::vector<std::vector<GLuint>> tileData, GLuint lvlWidth, GLuint lvlHeight)
{
    // 计算每个维度的大小
    GLuint height = tileData.size();
    GLuint width = tileData[0].size();
    GLfloat unit_width = lvlWidth / static_cast<GLfloat>(width);
    GLfloat unit_height = lvlHeight / height;
    // 基于tileData初始化关卡
    for (GLuint y = 0; y < height; ++y)
    {
        for (GLuint x = 0; x < width; ++x)
        {
            // 检查砖块类型
            if (tileData[y][x] == 1)
            {
                glm::vec2 pos(unit_width * x, unit_height * y);
                glm::vec2 size(unit_width, unit_height);
                GameObject obj(pos, size,
                               ResourceManager::GetTexture("block_solid"),
                               glm::vec3(0.8f, 0.8f, 0.7f));
                obj.IsSolid = GL_TRUE;
                this->Bricks.push_back(obj);
            }
            else if (tileData[y][x] > 1)
            {
                glm::vec3 color = glm::vec3(1.0f); // 默认为白色
                if (tileData[y][x] == 2)
                    color = glm::vec3(0.2f, 0.6f, 1.0f);
                else if (tileData[y][x] == 3)
                    color = glm::vec3(0.0f, 0.7f, 0.0f);
                else if (tileData[y][x] == 4)
                    color = glm::vec3(0.8f, 0.8f, 0.4f);
                else if (tileData[y][x] == 5)
                    color = glm::vec3(1.0f, 0.5f, 0.0f);

                glm::vec2 pos(unit_width * x, unit_height * y);
                glm::vec2 size(unit_width, unit_height);
                this->Bricks.push_back(
                    GameObject(pos, size, ResourceManager::GetTexture("block"), color));
            }
        }
    }
}
```

`init`函数遍历每个被加载的数字，处理后将一个相应的`GameObject`添加到关卡的容器中。每个砖块的尺寸(`unit_width`和`unit_height`)根据砖块的总数被自动计算以便于每块砖可以完美地适合屏幕边界。

在这里我们用两个新的纹理加载游戏对象，分别为`block`纹理与`solid block`纹理。



这里有一个很好的小窍门，即这些纹理是完全灰度的。其效果是，我们可以在游戏代码中，通过将灰度值与定义好的颜色矢量相乘来巧妙地操纵它们的颜色，就如同我们在SpriteRenderer中所做的那样。这样一来，自定义的颜色/外观就不会显得怪异或不平衡。

GameLevel类还包含一些其他的功能，比如渲染所有未被破坏的砖块，或验证是否所有的可破坏砖块均被摧毁。你可以在下面找到GameLevel类的源码：

- GameLevel: [头文件](#), [代码](#)

因为支持任意数量的行和列，这个游戏关卡类给我们带来了很大的灵活性，用户可以通过修改关卡文件轻松创建自己的关卡。

在游戏中

我们希望在Breakout游戏中支持多个关卡，因此我们将在Game类中添加一个持有GameLevel变量的容器。同时我们还将存储当前的游戏关卡。

```
class Game
{
    [...]
    std::vector<GameLevel> Levels;
    GLuint             Level;
    [...]
};
```

这个教程的Breakout版本共有4个游戏关卡：

- [Standard](#)
- [A few small gaps](#)
- [Space invader](#)
- [Bounce galore](#)

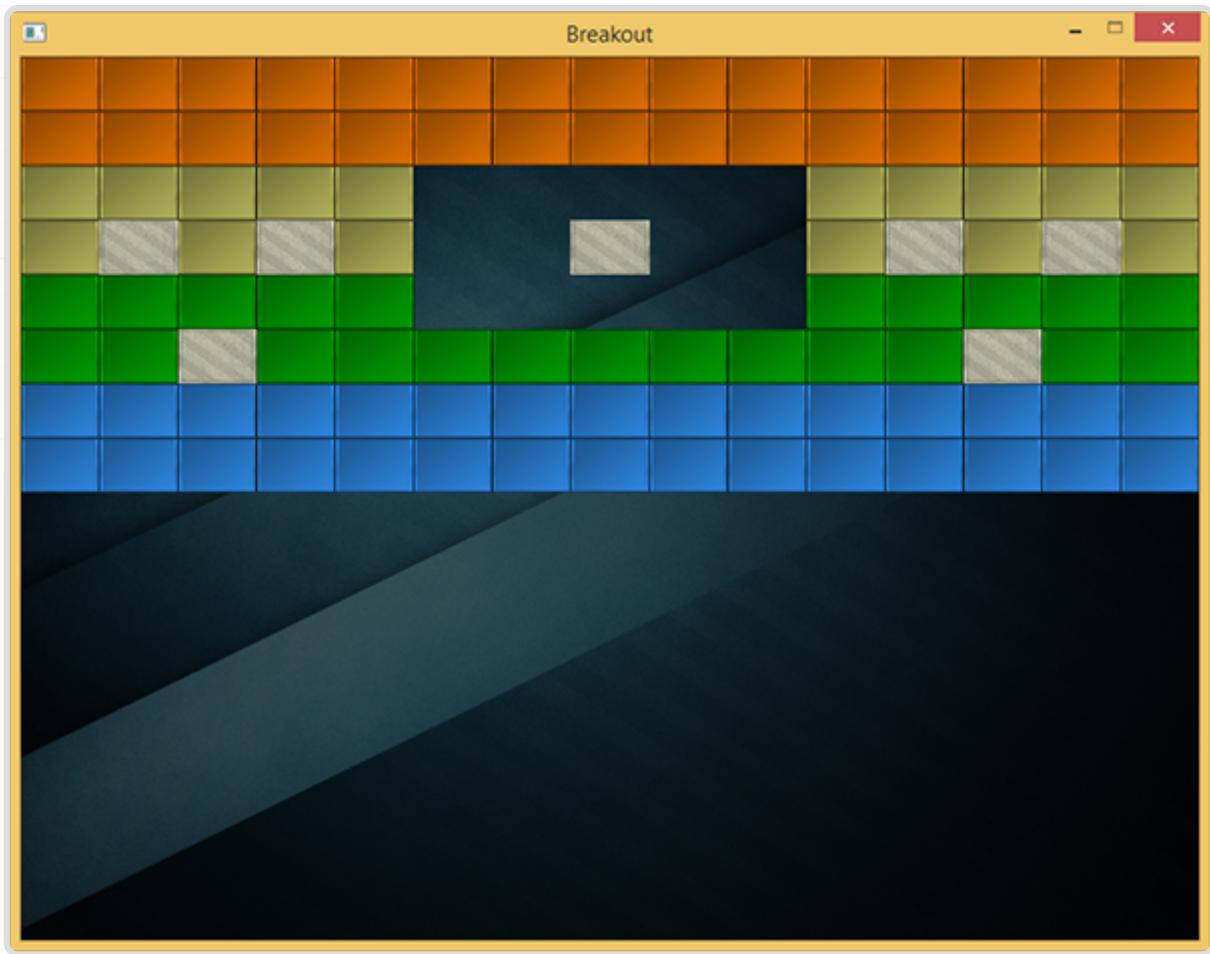
然后Game类的init函数初始化每个纹理和关卡：

```
void Game::Init()
{
    [...]
    // 加载纹理
    ResourceManager::LoadTexture("textures/background.jpg", GL_FALSE, "background");
    ResourceManager::LoadTexture("textures/awesomeface.png", GL_TRUE, "face");
    ResourceManager::LoadTexture("textures/block.png", GL_FALSE, "block");
    ResourceManager::LoadTexture("textures/block_solid.png", GL_FALSE, "block_solid");
    // 加载关卡
    GameLevel one; one.Load("levels/one.lvl", this->Width, this->Height * 0.5);
    GameLevel two; two.Load("levels/two.lvl", this->Width, this->Height * 0.5);
    GameLevel three; three.Load("levels/three.lvl", this->Width, this->Height * 0.5);
    GameLevel four; four.Load("levels/four.lvl", this->Width, this->Height * 0.5);
    this->Levels.push_back(one);
    this->Levels.push_back(two);
    this->Levels.push_back(three);
    this->Levels.push_back(four);
    this->Level = 1;
}
```

现在剩下要做的就是通过调用当前关卡的Draw函数来渲染我们完成的关卡，然后使用给定的sprite渲染器调用每个GameObject的Draw函数。除了关卡之外，我们还会用一个很好的[背景图片](#)来渲染这个场景：

```
void Game::Render()
{
    if(this->State == GAME_ACTIVE)
    {
        // 绘制背景
        Renderer->DrawSprite(ResourceManager::GetTexture("background"),
            glm::vec2(0, 0), glm::vec2(this->Width, this->Height), 0.0f
        );
        // 绘制关卡
        this->Levels[this->Level].Draw(*Renderer);
    }
}
```

结果便是如下这个被呈现的关卡，它使我们的游戏变得开始生动起来：



玩家挡板

此时我们在场景底部引入一个由玩家控制的挡板，挡板只允许水平移动，并且在它接触任意场景边缘时停止。对于玩家挡板，我们将使用[以下纹理](#)：



一个挡板对象拥有位置、大小、渲染纹理等属性，所以我们理所当然地将其定义为一个**GameObject**。

```
// 初始化挡板的大小
const glm::vec2 PLAYER_SIZE(100, 20);
// 初始化当班的速率
const GLfloat PLAYER_VELOCITY(500.0f);

GameObject *Player;

void Game::Init()
{
    [...]
    ResourceManager::LoadTexture("textures/paddle.png", true, "paddle");
    [...]
    glm::vec2 playerPos = glm::vec2(
        this->Width / 2 - PLAYER_SIZE.x / 2,
        this->Height - PLAYER_SIZE.y
    );
    Player = new GameObject(playerPos, PLAYER_SIZE, ResourceManager::GetTexture("paddle"));
}
```

这里我们定义了几个常量来初始化挡板的大小与速率。在**Game**的**Init**函数中我们计算挡板的初始位置，使其中心与场景的水平中心对齐。

除此之外我们还需要在**Game**的**Render**函数中添加：

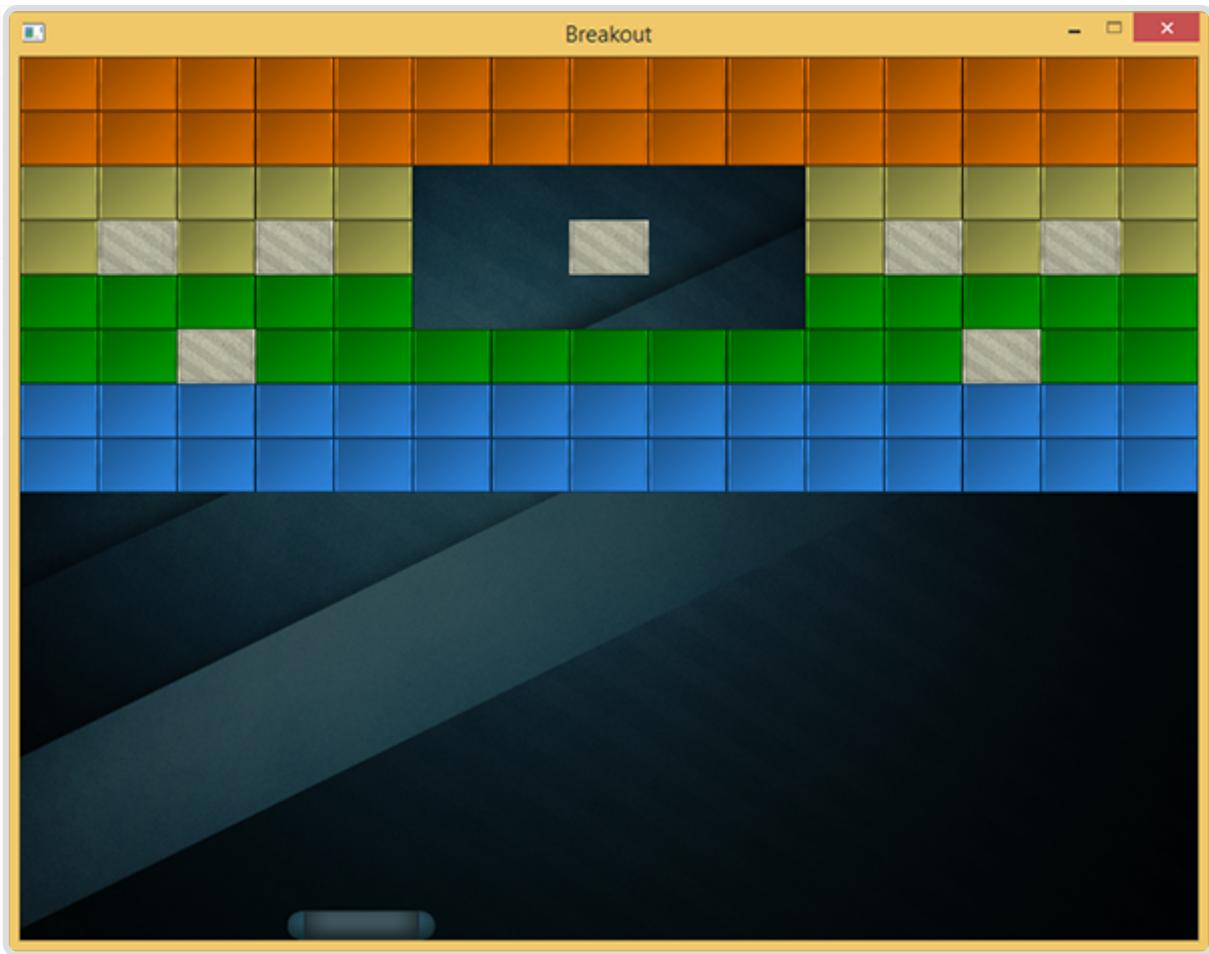
```
Player->Draw(*Renderer);
```

如果你现在启动游戏，你不仅会看到关卡画面，还会有一个在场景底部边缘的奇特的挡板。到目前为止，它除了静态地放置在那以外不会发生任何事情，因此我们需要进入游戏的 `ProcessInput` 函数，使得当玩家按下A和D时，挡板可以水平移动。

```
void Game::ProcessInput(GLfloat dt)
{
    if (this->State == GAME_ACTIVE)
    {
        GLfloat velocity = PLAYER_VELOCITY * dt;
        // 移动挡板
        if (this->Keys[GLFW_KEY_A])
        {
            if (Player->Position.x >= 0)
                Player->Position.x -= velocity;
        }
        if (this->Keys[GLFW_KEY_D])
        {
            if (Player->Position.x <= this->Width - Player->Size.x)
                Player->Position.x += velocity;
        }
    }
}
```

在这里，我们根据用户按下的键，向左或向右移动挡板(注意我们将速率与deltaTime相乘)。当挡板的x值小于0，它将移动出游戏场景的最左侧，所以我们只允许挡板的x值大于0时向左移动。对于右侧边缘我们做相同的处理，但我们必须比较场景的右侧边缘与挡板的右侧边缘，即场景宽度减去挡板宽度。

现在启动游戏，将呈现一个玩家可控制在整个场景底部自由移动的挡板。



你可以在下面找到更新后的Game类代码：

- Game: [头文件](#), [代码](#)

碰撞

球

原文 [Ball](#)

作者 JoeyDeVries

翻译 [aillieo](#)

校对 暂未校对

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

此时我们已经有了一个包含有很多砖块和玩家的一个挡板的关卡。与经典的Breakout内容相比还差一个球。游戏的目的是让球撞击所有的砖块，直到所有的可销毁砖块都被销毁，但同时也要满足条件：球不能碰触屏幕的下边缘。

除了通用的游戏对象组件，球还需要有半径和一个布尔值，该布尔值用于指示球被固定(stuck)在玩家挡板上还是被允许自由运动的状态。当游戏开始时，球被初始固定在玩家挡板上，直到玩家按下任意键开始游戏。

由于球只是一个附带了一些额外属性的GameObject，所以按照常规需要创建BallObject类作为GameObject的子类。

```
class BallObject : public GameObject
{
public:
    // 球的状态
    GLfloat Radius;
    GLboolean Stuck;

    BallObject();
    BallObject(glm::vec2 pos, GLfloat radius, glm::vec2 velocity, Texture2D sprite);

    glm::vec2 Move(GLfloat dt, GLuint window_width);
    void Reset(glm::vec2 position, glm::vec2 velocity);
};
```

`BallObject`的构造函数不但初始化了其自身的值，而且实际上也潜在地初始化了`GameObject`。`BallObject`类拥有一个`Move`函数，该函数用于根据球的速度来移动球，并检查它是否碰到了场景的任何边界，如果碰到的话就会反转球的速度：

```
glm::vec2 BallObject::Move(GLfloat dt, GLuint window_width)
{
    // 如果没有被固定在挡板上
    if (!this->Stuck)
    {
        // 移动球
        this->Position += this->Velocity * dt;
        // 检查是否在窗口边界以外，如果是的话反转速度并恢复到正确的位置
        if (this->Position.x <= 0.0f)
        {
            this->Velocity.x = -this->Velocity.x;
            this->Position.x = 0.0f;
        }
        else if (this->Position.x + this->Size.x >= window_width)
        {
            this->Velocity.x = -this->Velocity.x;
            this->Position.x = window_width - this->Size.x;
        }
        if (this->Position.y <= 0.0f)
        {
            this->Velocity.y = -this->Velocity.y;
            this->Position.y = 0.0f;
        }
    }
    return this->Position;
}
```

除了反转球的速度之外，我们还需要把球沿着边界重新放置回来。只有在没有被固定时球才能够移动。

因为如果球碰触到底部边界时玩家会结束游戏（或失去一条命），所以在底部边界没有代码来控制球反弹。我们稍后需要在代码中某些地方实现这一逻辑。

你可以在下边看到`BallObject`的代码：

- `BallObject`: [header](#), [code](#)

首先我们在游戏中添加球。与玩家挡板相似，我们创建一个球对象并且定义两个用来初始化球的常量。对于球的纹理，我们会使用在LearnOpenGL Breakout游戏中完美适用的一张图片：[球纹理](#)。

```
// 初始化球的速度
const glm::vec2 INITIAL_BALL_VELOCITY(100.0f, -350.0f);
// 球的半径
const GLfloat BALL_RADIUS = 12.5f;

BallObject *Ball;

void Game::Init()
{
    [...]
    glm::vec2 ballPos = playerPos + glm::vec2(PLAYER_SIZE.x / 2 - BALL_RADIUS, -BALL_RADIUS * 2);
    Ball = new BallObject(ballPos, BALL_RADIUS, INITIAL_BALL_VELOCITY,
        ResourceManager::GetTexture("face"));
}
```

然后我们在每帧中调用游戏代码中[Update](#)函数里的[Move](#)函数来更新球的位置：

```
void Game::Update(GLfloat dt)
{
    Ball->Move(dt, this->Width);
}
```

除此之外，由于球初始是固定在挡板上的，我们必须让玩家能够从固定的位置重新移动它。我们选择使用空格键来从挡板释放球。这意味着我们必须稍微修改[ProcessInput](#)函数：

```
void Game::ProcessInput(GLfloat dt)
{
    if (this->State == GAME_ACTIVE)
    {
        GLfloat velocity = PLAYER_VELOCITY * dt;
        // 移动玩家挡板
        if (this->Keys[GLFW_KEY_A])
        {
            if (Player->Position.x >= 0)
            {
                Player->Position.x -= velocity;
                if (Ball->Stuck)
                    Ball->Position.x -= velocity;
            }
        }
        if (this->Keys[GLFW_KEY_D])
        {
            if (Player->Position.x <= this->Width - Player->Size.x)
            {
                Player->Position.x += velocity;
                if (Ball->Stuck)
                    Ball->Position.x += velocity;
            }
        }
        if (this->Keys[GLFW_KEY_SPACE])
            Ball->Stuck = false;
    }
}
```

现在如果玩家按下了空格键，球的`stuck`值会设置为`false`。我们还需要更新`ProcessInput`函数，当球被固定的时候，会跟随挡板的位置来移动球。

最后我们需要渲染球，此时这应该很显而易见了：

```
void Game::Render()
{
    if (this->State == GAME_ACTIVE)
    {
        [...]
        Ball->Draw(*Renderer);
    }
}
```

结果就是球会跟随着挡板，并且当我们按下空格键时球开始自由运动。球会在左侧、右侧和顶部边界合理地反弹，但看起来不会撞击任何的砖块，就像我们可以在下边的视频中看到的那样：

我们要做的就是创建一个或多个函数用于检查球对象是否撞击关卡中的任何砖块，如果撞击的话就销毁砖块。这些所谓的碰撞检测(`collision detection`)功能将是我们[下一个](#)教程的重点。

碰撞检测

原文 [Collision detection](#)

作者 JoeyDeVries

翻译 [aillieo](#)

校对 暂未校对

Note

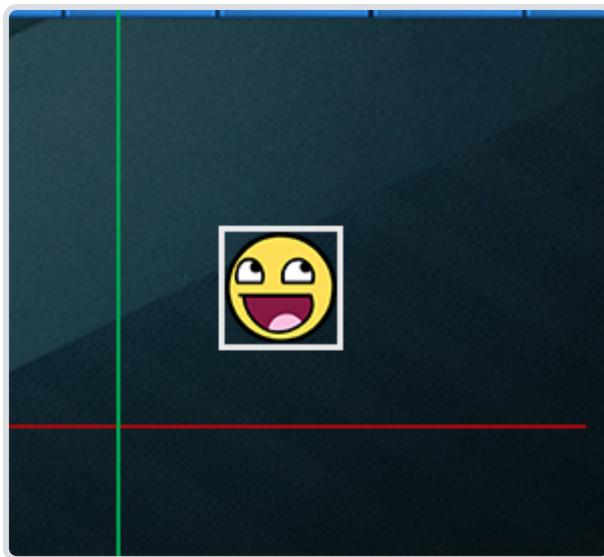
本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

当试图判断两个物体之间是否有碰撞发生时，我们通常不使用物体本身的数据，因为这些物体常常会很复杂，这将导致碰撞检测变得很复杂。正因这一点，使用重叠在物体上的更简单的外形（通常有较简单明确的数学定义）来进行碰撞检测成为常用的方法。我们基于这些简单的外形来检测碰撞，这样代码会变得更简单且节省了很多性能。这些**碰撞外形**例如圆、球体、长方形和立方体等，与拥有上百个三角形的网格相比简单了很多。

虽然它们确实提供了更简单更高效的碰撞检测算法，但这些简单的碰撞外形拥有一个共同的缺点，这些外形通常无法完全包裹物体。产生的影响就是当检测到碰撞时，实际的物体并没有真正的碰撞。必须记住的是这些外形仅仅是真实外形的近似。

AABB - AABB 碰撞

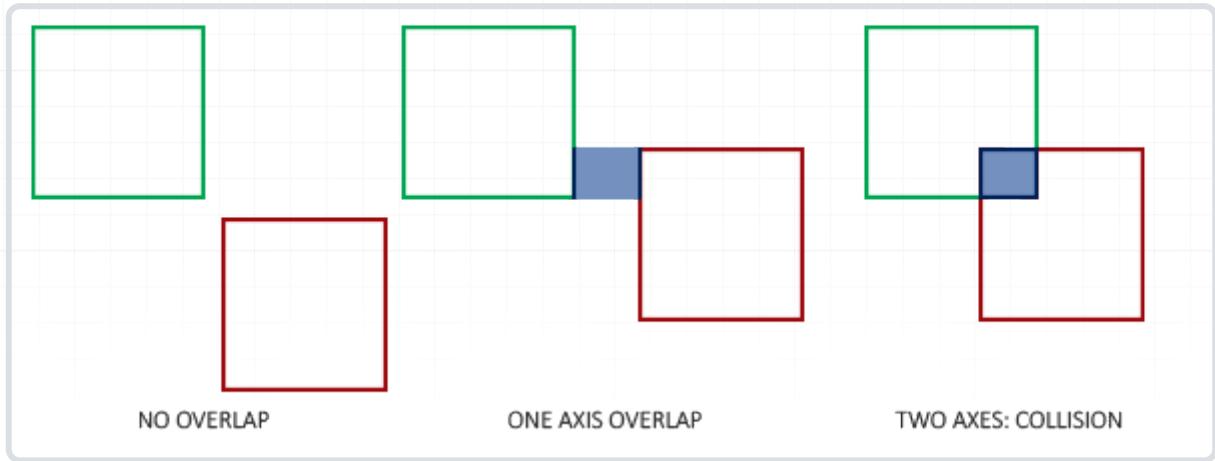
AABB代表的是**轴对齐碰撞箱**(Axis-aligned Bounding Box)，碰撞箱是指与场景基础坐标轴（2D中的是x和y轴）对齐的长方形的碰撞外形。与坐标轴对齐意味着这个长方形没有经过旋转并且它的边线和场景中基础坐标轴平行（例如，左右边线和y轴平行）。这些碰撞箱总是和场景的坐标轴平行，这使得所有的计算都变得更简单。下边是我们用一个AABB包裹一个球对象（物体）：



Breakout中几乎所有的物体都是基于长方形的物体，因此很理所当地使用轴对齐碰撞箱来进行碰撞检测。这就是我们接下来要做的。

有多种方式来定义与坐标轴对齐的碰撞箱。其中一种定义AABB的方式是获取左上角点和右下角点的位置。我们定义的`GameObject`类已经包含了一个左上角点位置（它的Position vector）并且我们可以通过把左上角点的矢量加上它的尺寸（`Position + Size`）很容易地计算出右下角点。每个`GameObject`都包含一个AABB我们可以高效地使用它们碰撞。

那么我们如何判断碰撞呢？当两个碰撞外形进入对方的区域时就会发生碰撞，例如定义了第一个物体的碰撞外形以某种形式进入了第二个物体的碰撞外形。对于AABB来说很容易判断，因为它们是与坐标轴对齐的：对于每个轴我们要检测两个物体的边界在此轴向是否有重叠。因此我们只是简单地检查两个物体的水平边界是否重合以及垂直边界是否重合。如果水平边界和垂直边界都有重叠那么我们就检测到一次碰撞。



将这一概念转化为代码也是很直白的。我们对两个轴都检测是否重叠，如果都重叠就返回碰撞：

```
GLboolean CheckCollision(GameObject &one, GameObject &two) // AABB - AABB collision
{
    // x轴方向碰撞?
    bool collisionX = one.Position.x + one.Size.x >= two.Position.x &&
        two.Position.x + two.Size.x >= one.Position.x;
    // y轴方向碰撞?
    bool collisionY = one.Position.y + one.Size.y >= two.Position.y &&
        two.Position.y + two.Size.y >= one.Position.y;
    // 只有两个轴向都有碰撞时才碰撞
    return collisionX && collisionY;
}
```

我们检查第一个物体的最右侧是否大于第二个物体的最左侧并且第二个物体的最右侧是否大于第一个物体的最左侧；垂直的轴向与此相似。如果您无法顺利地将这一过程可视化，可以尝试在纸上画边界线/长方形来自行判断。

为更好地组织碰撞的代码，我们在**Game**类中加入一个额外的函数：

```
class Game
{
public:
    [...]
    void DoCollisions();
};
```

我们可以使用`DoCollisions`来检查球与关卡中的砖块是否发生碰撞。如果检测到碰撞，就将砖块的`Destroyed`属性设为`true`，此举会停止关卡中对此砖块的渲染。

```
void Game::DoCollisions()
{
    for (GameObject &box : this->Levels[this->Level].Bricks)
    {
        if (!box.Destroyed)
        {
            if (CheckCollision(*Ball, box))
            {
                if (!box.IsSolid)
                    box.Destroyed = GL_TRUE;
            }
        }
    }
}
```

接下来我们需要更新`Game`的`Update`函数：

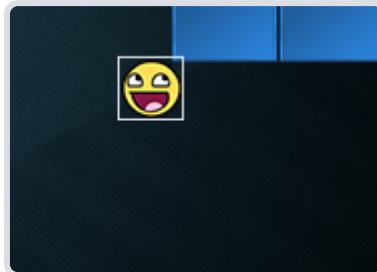
```
void Game::Update(GLfloat dt)
{
    // 更新对象
    Ball->Move(dt, this->Width);
    // 检测碰撞
    this->DoCollisions();
}
```

此时如果我们运行代码，球会与每个砖块进行碰撞检测，如果砖块不是实心的，则表示砖块被销毁。如果运行游戏以下是你会看到的：

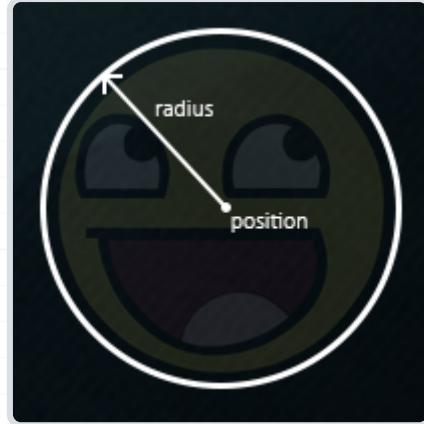
虽然碰撞检测确实生效，但并不是非常准确，因为球会在不直接接触到大多数砖块时与它们发生碰撞。我们来实现另一种碰撞检测技术。

AABB - 圆碰撞检测

由于球是一个圆形的物体，AABB或许不是球的最佳碰撞外形。碰撞的代码中将球视为长方形框，因此常常会出现球碰撞了砖块但此时球精灵还没有接触到砖块。

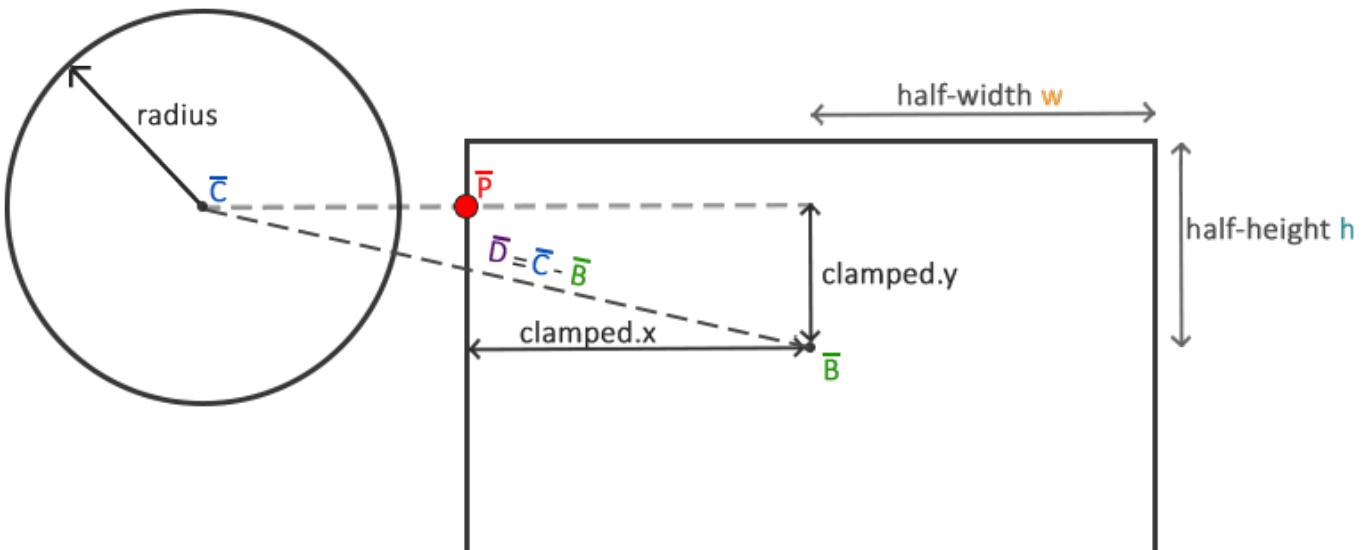


使用圆形碰撞外形而不是AABB来代表球会更合乎常理。因此我们在球对象中包含了`Radius`变量，为了定义圆形碰撞外形，我们需要的是一个位置矢量和一个半径。



这意味着我们不得不修改检测算法，因为当前的算法只适用于两个AABB的碰撞。检测圆和AABB碰撞的算法会稍稍复杂，关键点如下：我们会找到AABB上距离圆最近的一个点，如果圆到这一点的距离小于它的半径，那么就产生了碰撞。

难点在于获取AABB上的最近点。下图展示了对于任意的AABB和圆我们如何计算该点：



首先我们要获取球心与AABB中心的矢量差。接下来用AABB的半边长(half-extents)和来限制(clamp)矢量。长方形的半边长是指长方形的中心到它的边的距离；简单的说就是它的尺寸除以2。这一过程返回的是一个总是位于AABB的边上的位置矢量（除非圆心在AABB内部）。

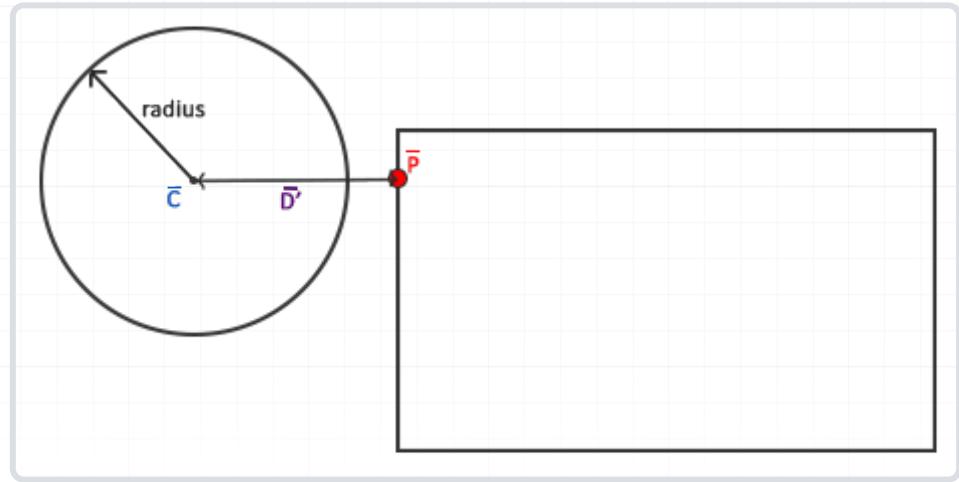
限制运算把一个值限制在给定范围内，并返回限制后的值。通常可以表示为：

```
float clamp(float value, float min, float max) {
    return std::max(min, std::min(max, value));
}
```

例如，值 $42.0f$ 被限制到 $6.0f$ 和 $3.0f$ 之间会得到 $6.0f$ ；而 $4.20f$ 会被限制为 $4.20f$ 。

限制一个2D的矢量表示将其 x 和 y 分量都限制在给定的范围内。

这个限制后矢量就是AABB上距离圆最近的点。接下来我们需要做的就是计算一个新的差矢量，它是圆心和的差矢量。



既然我们已经有了矢量，我们就可以比较它的长度和圆的半径以判断是否发生了碰撞。

这一过程通过下边的代码来表示：

```
GLboolean CheckCollision(BallObject &one, GameObject &two) // AABB - Circle collision
{
    // 获取圆的中心
    glm::vec2 center(one.Position + one.Radius);
    // 计算AABB的信息（中心、半边长）
    glm::vec2 aabb_half_extents(two.Size.x / 2, two.Size.y / 2);
    glm::vec2 aabb_center(
        two.Position.x + aabb_half_extents.x,
        two.Position.y + aabb_half_extents.y
    );
    // 获取两个中心的差矢量
    glm::vec2 difference = center - aabb_center;
    glm::vec2 clamped = glm::clamp(difference, -aabb_half_extents, aabb_half_extents);
    // AABB_center加上clamped这样就得到了碰撞箱上距离圆最近的点closest
    glm::vec2 closest = aabb_center + clamped;
    // 获得圆心center和最近点closest的矢量并判断是否 length <= radius
    difference = closest - center;
    return glm::length(difference) < one.Radius;
}
```

我们创建了`CheckCollision`的一个重载函数用于专门处理一个`BallObject`和一个`GameObject`的情况。因为我们并没有在对象中保存碰撞外形的信息，因此我们必须为其计算：首先计算球心，然后是AABB的半边长及中心。

使用这些碰撞外形的参数，我们计算出`difference`然后得到限制后的值`clamped`，并与AABB中心相加得到`closest`。然后计算出`center`和`closest`的矢量差并返回两个外形是否碰撞。

之前我们调用`CheckCollision`时将球对象作为其第一个参数，因此现在`CheckCollision`的重载变量会自动生效，我们无需修改任何代码。现在的结果会比之前的碰撞检测算法更准确。

看起来生效了，但仍缺少一些东西。我们准确地检测了所有碰撞，但碰撞并没有对球产生任何反作用。我们需要在碰撞时产生一些反作用，例如当碰撞发生时，更新球的位置和/或速度。这将是[下一个](#)教程的主题。

碰撞处理

原文 [Collision resolution](#)

作者 JoeyDeVries

翻译 [aillieo](#)

校对 暂未校对

Note

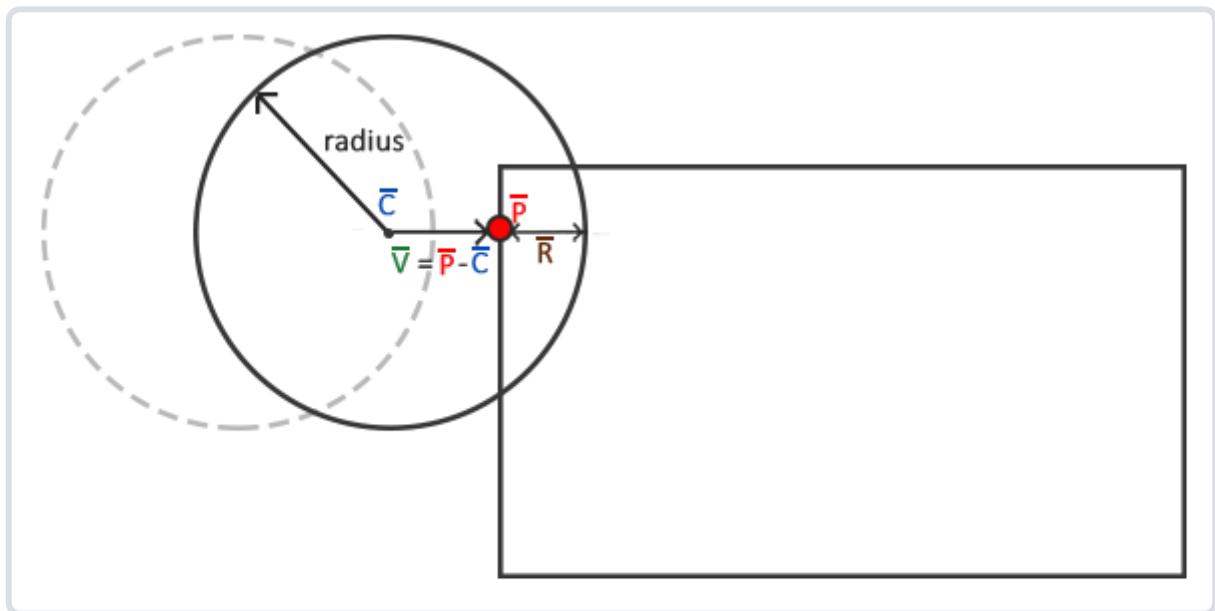
本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

上个教程的最后，我们得到了一种有效的碰撞检测方案。但是球对检测到的碰撞不会有反作用；它仅仅是径直穿过所有的砖块。我们希望球会从撞击到的砖块反弹。此教程将讨论如何使用AABB-圆碰撞方案实现这项称为**碰撞处理**(Collision Resolution)的功能。

当碰撞发生时，我们希望出现两个现象：重新定位球，以免它进入另一个物体，其次是改变球的速度方向，使它看起来像是物体的反弹。

碰撞重定位

为了把球对象定位到碰撞的AABB的外部，我们必须明确球侵入碰撞框的距离。为此我们要回顾上一节教程中的示意图：



此时球少量进入了AABB，所以检测到了碰撞。我们现在希望将球从移出AABB的外形使其仅仅碰触到AABB，像是没有碰撞一样。为了确定需要将球从AABB中移出多少距离，我们需要找回矢量，它代表的是侵入AABB的程度。为得到我们用球的半径减去。矢量是最远点和球心的差矢量。

有了之后我们将球的位置偏移就将球直接放置在与AABB紧邻的位置；此时球已经被重定位到合适的位置。

碰撞方向

下一步我们需要确定碰撞之后如何更新球的速度。对于Breakout我们使用以下规则来改变球的速度：

1. 如果球撞击AABB的右侧或左侧，它的水平速度（ x ）将会反转。
2. 如果球撞击AABB的上侧或下侧，它的垂直速度（ y ）将会反转。

但是如何判断球撞击AABB的方向呢？解决这一问题有很多种方法，其中之一是对于每个砖块使用4个AABB而不是1个AABB，并把它们放置到砖块的每个边上。使用这种方法我们可以确定被碰撞的是哪个AABB和哪个边。但是有一种使用点乘(dot product)的更简单的方法。

您或许还记得[变换](#)教程中点乘可以得到两个正交化的矢量的夹角。如果我们定义指向北、南、西和东的四个矢量，然后计算它们和给定矢量的夹角会怎么样？由这四个方向矢量和给定的矢量点乘积的结果中的最高值（点乘积的最大值为`1.0f`，代表`0`度角）即是矢量的方向。

这一过程如下代码所示：

```
Direction VectorDirection(glm::vec2 target)
{
    glm::vec2 compass[] = {
        glm::vec2(0.0f, 1.0f), // 上
        glm::vec2(1.0f, 0.0f), // 右
        glm::vec2(0.0f, -1.0f), // 下
        glm::vec2(-1.0f, 0.0f) // 左
    };
    GLfloat max = 0.0f;
    GLuint best_match = -1;
    for (GLuint i = 0; i < 4; i++)
    {
        GLfloat dot_product = glm::dot(glm::normalize(target), compass[i]);
        if (dot_product > max)
        {
            max = dot_product;
            best_match = i;
        }
    }
    return (Direction)best_match;
}
```

此函数比较了`target`矢量和`compass`数组中各方向矢量。`compass`数组中与`target`角度最接近的矢量，即是返回给函数调用者的`Direction`。这里的`Direction`是一个[Game](#)类的头文件中定义的枚举类型：

```
enum Direction {
    UP,
    RIGHT,
    DOWN,
    LEFT
};
```

既然我们已经知道了如何获得以及如何判断球撞击AABB的方向，我们开始编写碰撞处理的代码。

AABB - 圆碰撞检测

为了计算碰撞处理所需的数值我们要从碰撞的函数中获取更多的信息而不只是一个`true`或`false`，因此我们要返回一个包含更多信息的[tuple](#)，这些信息即是碰撞发生时的方向及差矢量（）。你可以在头文件中找到[tuple](#)。

为了更好组织代码，我们把碰撞相关的数据使用typedef定义为Collision：

```
typedef std::tuple<GLboolean, Direction, glm::vec2> Collision;
```

接下来我们还需要修改CheckCollision函数的代码，使其不仅仅返回true或false而是还包含方向和差矢量：

```
Collision CheckCollision(BallObject &one, GameObject &two) // AABB - AABB 碰撞
{
    [...]
    if (glm::length(difference) <= one.Radius)
        return std::make_tuple(GL_TRUE, VectorDirection(difference), difference);
    else
        return std::make_tuple(GL_FALSE, UP, glm::vec2(0, 0));
}
```

Game类的DoCollision函数现在不仅仅只检测是否出现了碰撞，而且在碰撞发生时会有适当的动作。此函数现在会计算碰撞侵入的程度（如本教程一开始计时的示意图中所示）并且基于碰撞方向使球的位置矢量与其相加或相减。

```
void Game::DoCollisions()
{
    for (GameObject &box : this->Levels[this->Level].Bricks)
    {
        if (!box.Destroyed)
        {
            Collision collision = CheckCollision(*Ball, box);
            if (std::get<0>(collision)) // 如果collision 是 true
            {
                // 如果砖块不是实心就销毁砖块
                if (!box.IsSolid)
                    box.Destroyed = GL_TRUE;
                // 碰撞处理
                Direction dir = std::get<1>(collision);
                glm::vec2 diff_vector = std::get<2>(collision);
                if (dir == LEFT || dir == RIGHT) // 水平方向碰撞
                {
                    Ball->Velocity.x = -Ball->Velocity.x; // 反转水平速度
                    // 重定位
                    GLfloat penetration = Ball->Radius - std::abs(diff_vector.x);
                    if (dir == LEFT)
                        Ball->Position.x += penetration; // 将球右移
                    else
                        Ball->Position.x -= penetration; // 将球左移
                }
                else // 垂直方向碰撞
                {
                    Ball->Velocity.y = -Ball->Velocity.y; // 反转垂直速度
                    // 重定位
                    GLfloat penetration = Ball->Radius - std::abs(diff_vector.y);
                    if (dir == UP)
                        Ball->Position.y -= penetration; // 将球上移
                    else
                        Ball->Position.y += penetration; // 将球下移
                }
            }
        }
    }
}
```

不要被函数的复杂度给吓到，因为它仅仅是我们目前为止的概念的直接转化。首先我们会检测碰撞如果发生了碰撞且砖块不是实心的那么就销毁砖块。然后我们从tuple中获取到了碰撞的方向`dir`以及表示的差矢量`diff_vector`，最终完成碰撞处理。

我们首先检查碰撞方向是水平还是垂直，并据此反转速度。如果是水平方向，我们从`diff_vector`的x分量计算侵入量RR并根据碰撞方向用球的位置矢量加上或减去它。垂直方向的碰撞也是如此，但是我们要操作各矢量的y分量。

现在运行你的应用程序，应该会向你展示一套奏效的碰撞方案，但可能会很难真正看到它的效果，因为一旦球碰撞到了一个砖块就会弹向底部并永远丢失。我们可以通过处理玩家挡板的碰撞来修复这一问题。

玩家 - 球碰撞

球和玩家之间的碰撞与我们之前讨论的碰撞稍有不同，因为这里应当基于撞击挡板的点与（挡板）中心的距离来改变球的水平速度。撞击点距离挡板的中心点越远，则水平方向的速度就会越大。

```
void Game::DoCollisions()
{
    [...]
    Collision result = CheckCollision(*Ball, *Player);
    if (!Ball->Stuck && std::get<0>(result))
    {
        // 检查碰到了挡板的哪个位置，并根据碰到哪个位置来改变速度
        GLfloat centerBoard = Player->Position.x + Player->Size.x / 2;
        GLfloat distance = (Ball->Position.x + Ball->Radius) - centerBoard;
        GLfloat percentage = distance / (Player->Size.x / 2);
        // 依据结果移动
        GLfloat strength = 2.0f;
        glm::vec2 oldVelocity = Ball->Velocity;
        Ball->Velocity.x = INITIAL_BALL_VELOCITY.x * percentage * strength;
        Ball->Velocity.y = -Ball->Velocity.y;
        Ball->Velocity = glm::normalize(Ball->Velocity) * glm::length(oldVelocity);
    }
}
```

在我们完成了球和各砖块的碰撞检测之后，我们来检测球和玩家挡板是否发生碰撞。如果有碰撞（并且球不是被固定在挡板上）我们要计算球的中心与挡板中心的距离和挡板的半边长的百分比。之后球的水平速度会依据它撞击挡板的点到挡板中心的距离来更新。除了更新水平速度之外我们还需要反转它的y方向速度。

注意旧的速度被存储为`oldVelocity`。之所以要存储旧的速度是因为我们只更新球的速度矢量中水平方向的速度并保持它的y速度不变。这将意味着矢量的长度会持续变化，其产生的影响是如果球撞击到挡板的边缘则会比撞击到挡板中心有更大（也因此更强）的速度矢量。为此新的速度矢量会正交化然后乘以旧速度矢量的长度。这样一来，球的力量和速度将总是一致的，无论它撞击到挡板的那个地方。

粘板

无论你有没有注意到，但当运行代码时，球和玩家挡板的碰撞处理仍旧有一个大问题。以下的视频清楚地展示了将会出现的现象：

这种问题称为粘板问题(Sticky Paddle Issue)，出现的原因是玩家挡板以较高的速度移向球，导致球的中心进入玩家挡板。由于我们没有考虑球的中心在AABB内部的情况，游戏会持续试图对所有的碰撞做出响应，当球最终脱离时，已经对`y`向速度翻转了多次，以至于无法确定球在脱离后是向上还是向下运动。

我们可以引入一个小的特殊处理来很容易地修复这种行为，这个处理之所以成为可能是基于我们可以假设碰撞总是发生在挡板顶部的事实。我们总是简单地返回正的`y`速度而不是反转`y`速度，这样当它被卡住时也可以立即脱离。

```
//Ball->Velocity.y = -Ball->Velocity.y;
Ball->Velocity.y = -1 * abs(Ball->Velocity.y);
```

如果你足够仔细就会觉得这一影响仍然是可以被注意到的，但是我个人将此方法当作一种可接受的折衷处理。

底部边界

与经典的Breakout内容相比唯一缺少的就是失败条件了，失败会重置关卡和玩家。在Game类的Update函数中，我们要检查球是否接触到了底部边界，如果接触到就重置游戏。

```
void Game::Update(GLfloat dt)
{
    [...]
    if (Ball->Position.y >= this->Height) // 球是否接触底部边界?
    {
        this->ResetLevel();
        this->ResetPlayer();
    }
}
```

ResetLevel和ResetPlayer函数直接重新加载关卡并重置对象的各变量值为原始的值。现在游戏看起来应该是这样的：

就是这样了，我们创建完成了一个有相似机制的经典Breakout游戏的复制版。这里你可以找到Game类的源代码：[header](#), [code](#)。

一些注意事项

在视频游戏的发展过程中，碰撞检测是一个困难的话题甚至可能是最大的挑战。大多数的碰撞检测和处理方案是和物理引擎合并在一起的，正如多数现代的游戏中看到的那样。我们在Breakout游戏中使用的碰撞方案是一个非常简单的方案并且是专门给这类游戏所专用的。

需要强调的是这类碰撞检测和处理方式是不完美的。它只能计算每帧内可能发生的碰撞并且只能计算在该时间步时物体所在的位置；这意味着如果一个物体拥有一个很大的速度以致于在一帧内穿过了另一个物体，它将看起来像是从来没有与另一个物体碰撞过。因此如果出现掉帧或出现了足够高的速度，这一碰撞检测方案将无法应对。

(我们使用的碰撞方案) 仍然会出现这几个问题：

- 如果球运动得足够快，它可能在一帧内完整地穿过一个物体，而不会检测到碰撞。
- 如果球在一帧内同时撞击了一个以上的物体，它将会检测到两次碰撞并两次反转速度；这样不改变它的原始速度。
- 撞击到砖块的角时会在错误的方向反转速度，这是因为它在一帧内穿过的距离会引发VectorDirection返回水平方向还是垂直方向的差别。

但是，本教程目的在于教会读者们图形学和游戏开发的基础知识。因此，这里的碰撞方案可以服务于此目的；它更容易理解且在正常的场景中可以较好地运作。需要记住的是存在有更好的（更复杂）碰撞方案，在几乎所有的场景中都可以很好地运作（包括可移动的物体）如分离轴定理(Separating Axis Theorem)。

值得庆幸的是，有大量实用并且常常很高效的物理引擎（使用时间步无关的碰撞方案）可供您在游戏中使用。如果您希望在这一系统中有更深入的探索或需要更高级的物理系统又不理解其中的数学机理，[Box2D](#)是一个实现了物理系统和碰撞检测的可以在您的应用程序中的完美的2D物理库。

粒子

原文 [Particles](#)

作者 JoeydeVries

翻译 [ZMANT](#)

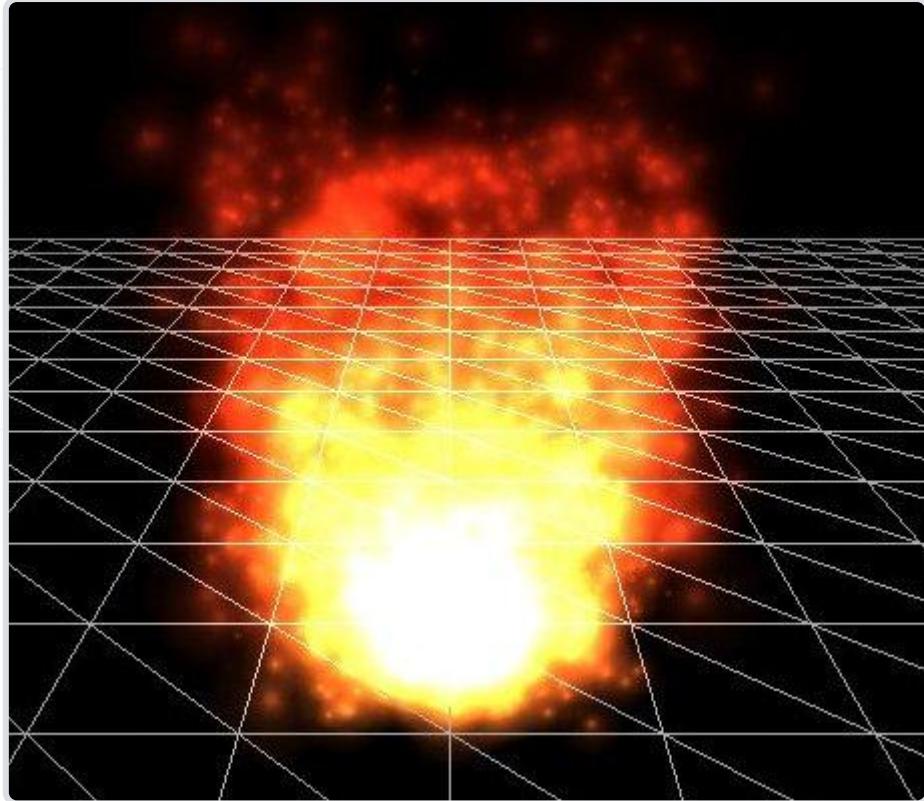
校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

一个微粒,从OpenGL的角度看就是一个总是面向摄像机方向且(通常)包含一个大部分区域是透明的纹理的小四边形。一个微粒本身主要就是一个精灵(sprite),前面我们已经早就使用过了,但是当你把成千上万个这些微粒放在一起的时候,就可以创造出令人疯狂的效果.

当处理这些微粒的时候,通常是由一个叫做粒子发射器或粒子生成器的东西完成的,从这个地方,持续不断的产生新的微粒并且旧的微粒随着时间逐渐消亡。如果这个粒子发射器产生一个带着类似烟雾纹理的微粒的时候,它的颜色亮度同时又随着与发射器距离的增加而变暗,那么就会产生出灼热的火焰的效果:



一个单一的微粒通常有一个生命值变量，并且从它产生开始就一直在缓慢的减少。一旦它的生命值少于某个极限值（通常是0）我们就会杀掉这个粒子，这样下一个粒子产生时就可以让它来替换那个被杀掉的粒子。一个粒子发射器控制它产生的所有粒子并且根据它们的属性来改变它们的行为。一个粒子通常有下面的属性：

```
struct Particle {
    glm::vec2 Position, Velocity;
    glm::vec4 Color;
    GLfloat Life;

    Particle()
        : Position(0.0f), Velocity(0.0f), Color(1.0f), Life(0.0f) { }
};
```

看上面那个火焰的例子，那个粒子发射器可能在靠近发射器的地方产生每一个粒子，并且有一个向上的速度，这样每个粒子都是朝着正\$y\$轴方向移动。那似乎有3个不同区域，只是可能相比其他的区域，给了某个区域内的粒子更快的速度。我们也可以看到，\$y\$轴方向越高的粒子，它们的黄色或者说亮度就越低。一旦某个粒子到达某个高度的时候，它的生命值就会耗尽然后被杀掉；绝不可能直冲云霄。

你可以想象到用这样一个系统，我们就可以创造一些有趣的效果比如火焰，青烟，烟雾，魔法效果，炮火残渣等等。在Breakout游戏里，我们将会使用下面那个小球来创建一个简单的粒子生成器来制作一些有趣的效果，结果看起来就像这样：

上面那个粒子生成器在这个球的位置产生无数的粒子，根据球移动的速度给了粒子相应的速度，并且根据它们的生命值来改变他们的颜色亮度。

为了渲染这些粒子，我们将会用到有不同实现的着色器：

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 position, vec2 texCoords>

out vec2 TexCoords;
out vec4 ParticleColor;

uniform mat4 projection;
uniform vec2 offset;
uniform vec4 color;

void main()
{
    float scale = 10.0f;
    TexCoords = vertex.zw;
    ParticleColor = color;
    gl_Position = projection * vec4((vertex.xy * scale) + offset, 0.0, 1.0);
}
```

以及像素着色器：

```
#version 330 core
in vec2 TexCoords;
in vec4 ParticleColor;
out vec4 color;

uniform sampler2D sprite;

void main()
{
    color = (texture(sprite, TexCoords) * ParticleColor);
}
```

我们获取每个粒子的位置和纹理属性并且设置两个uniform变量：\$offset\$和\$color\$来改变每个粒子的输出状态。注意到，在顶点着色器里，我们把这个四边形的粒子缩小了10倍；你也可以把这个缩放变量设置成uniform类型的变量从而控制一些个别的粒子。

首先，我们需要一个粒子数组，然后用Particle结构体的默认构造函数来实例化。

```
GLuint nr_particles = 500;
std::vector<Particle> particles;

for (GLuint i = 0; i < nr_particles; ++i)
    particles.push_back(Particle());
```

然后在每一帧里面，我们都会用一个起始变量来产生一些新的粒子并且对每个粒子（还活着的）更新它们的值。

```
GLuint nr_new_particles = 2;
// Add new particles
for (GLuint i = 0; i < nr_new_particles; ++i)
{
    int unusedParticle = FirstUnusedParticle();
    RespawnParticle(particles[unusedParticle], object, offset);
}
// Update all particles
for (GLuint i = 0; i < nr_particles; ++i)
{
    Particle &p = particles[i];
    p.Life -= dt; // reduce life
    if (p.Life > 0.0f)
    {
        // particle is alive, thus update
        p.Position -= p.Velocity * dt;
        p.Color.a -= dt * 2.5;
    }
}
```

第一个循环看起来可能有点吓人。因为这些粒子会随着时间消亡，我们就想在每一帧里面产生 `nr_new_particles` 个新粒子。但是一开始我们就知道了总的粒子数量是 `nr_particles`，所以我们不能简单的往粒子数组里面添加新的粒子。否则的话我们很快就会得到一个装满成千上万个粒子的数组，考虑到这个粒子数组里面其实只有一小部分粒子是存活的，这样就太浪费效率了。

我们要做的就是找到第一个消亡的粒子然后用一个新产生的粒子来更新它。函数 `FirstUnusedParticle` 就是试图找到第一个消亡的粒子并且返回它的索引值给调用者。

```
GLuint lastUsedParticle = 0;
GLuint FirstUnusedParticle()
{
    // Search from last used particle, this will usually return almost instantly
    for (GLuint i = lastUsedParticle; i < nr_particles; ++i){
        if (particles[i].Life <= 0.0f){
            lastUsedParticle = i;
            return i;
        }
    }
    // Otherwise, do a linear search
    for (GLuint i = 0; i < lastUsedParticle; ++i){
        if (particles[i].Life <= 0.0f){
            lastUsedParticle = i;
            return i;
        }
    }
    // Override first particle if all others are alive
    lastUsedParticle = 0;
    return 0;
}
```

这个函数存储了它找到的上一个消亡的粒子的索引值，由于下一个消亡的粒子索引值总是在上一个消亡的粒子索引值的右边，所以我们首先从它存储的上一个消亡的粒子索引位置开始查找，如果我们没有任何消亡的粒子，我们就简单的做一个线性查找，如果没有粒子消亡就返回索引值 `0`，结果就是第一个粒子被覆盖，需要注意的是，如果是最后一种情况，就意味着你粒子的生命值太长了，在每一帧里面需要产生更少的粒子，或者你只是没有保留足够的粒子，

之后，一旦粒子数组中第一个消亡的粒子被发现的时候，我们就通过调用 `RespawnParticle` 函数更新它的值，函数接受一个 `Particle` 对象，一个 `GameObject` 对象和一个 `offset` 向量：

```
void RespawnParticle(Particle &particle, GameObject &object, glm::vec2 offset)
{
    GLfloat random = ((rand() % 100) - 50) / 10.0f;
    GLfloat rColor = 0.5 + ((rand() % 100) / 100.0f);
    particle.Position = object.Position + random + offset;
    particle.Color = glm::vec4(rColor, rColor, rColor, 1.0f);
    particle.Life = 1.0f;
    particle.Velocity = object.Velocity * 0.1f;
}
```

这个函数简单的重置这个粒子的生命值为1.0f，随机的给一个大于 `0.5` 的颜色值(经过颜色向量)并且(在物体周围)分配一个位置和速度基于游戏里的物体。

对于更新函数里的第二个循环遍历了所有粒子，并且对于每个粒子的生命值都减去一个时间差；这样每个粒子的生命值就精确到了秒。然后再检查这个粒子是否是还活着的，若是，则更新它的位置和颜色属性。这里我们缓慢的减少粒子颜色值的 `alpha` 值，以至于它看起来就是随着时间而缓慢的消亡。

最后保留下来就是实际需要渲染的粒子：

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
particleShader.Use();
for (Particle particle : particles)
{
    if (particle.Life > 0.0f)
    {
        particleShader.SetVector2f("offset", particle.Position);
        particleShader.SetVector4f("color", particle.Color);
        particleTexture.Bind();
        glBindVertexArray(particleVA0);
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindVertexArray(0);
    }
}
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

在这，对于每个粒子，我们一一设置他们的 `uniform` 变量 `offse` 和 `color`，绑定纹理，然后渲染 `2D` 四边形的粒子。有趣的是我们在这看到了两次调用函数 `glBlendFunc`。当要渲染这些粒子的时候，我们使用 `GL_ONE` 替换默认的目的因子模式 `GL_ONE_MINUS_SRC_ALPHA`，这样，这些粒子叠加在一起的时候就会产生一些平滑的发热效果，就像在这个教程前面那样使用混合模式来渲染出火焰的效果也是可以的，这样在有大多数粒子的中心就会产生更加灼热的效果。

因为我们(就像这个系列教程的其他部分一样)喜欢让事情变得有条理，所以我们就创建了另一个类 `ParticleGenerator` 来封装我们刚刚谈到的所有功能。你可以在下面的链接里找到源码：

- [header](#), [code](#)

然后在游戏代码里,我们创建这样一个粒子发射器并且用[这个](#)纹理初始化。

```
ParticleGenerator *Particles;

void Game::Init()
{
    [...]
    ResourceManager::LoadShader("shaders/particle.vs", "shaders/particle.frag", nullptr, "particle");
    [...]
    ResourceManager::LoadTexture("textures/particle.png", GL_TRUE, "particle");
    [...]
    Particles = new ParticleGenerator(
        ResourceManager::GetShader("particle"),
        ResourceManager::GetTexture("particle"),
        500
    );
}
```

然后我们在 `Game` 类的 `Update` 函数里为粒子生成器添加一条更新语句:

```
void Game::Update(GLfloat dt)
{
    [...]
    // Update particles
    Particles->Update(dt, *Ball, 2, glm::vec2(Ball->Radius / 2));
    [...]
}
```

每个粒子都将使用球的游戏对象属性对象, 每帧产生两个粒子并且他们都是偏向球得中心, 最后是渲染粒子:

```
void Game::Render()
{
    if (this->State == GAME_ACTIVE)
    {
        [...]
        // Draw player
        Player->Draw(*Renderer);
        // Draw particles
        Particles->Draw();
        // Draw ball
        Ball->Draw(*Renderer);
    }
}
```

注意到, 我们是在渲染球体之前且在渲染其他物体之后渲染粒子的, 这样, 粒子就会在所有其他物体面前, 但报纸在球体之后, 你可以在[这里](#)找到更新的 `game` 类的源码。

如果你现在编译并运行你的程序, 你可能会看到在球体之后有一条小尾巴。就像这个教程开始的那样, 给了这个游戏更加现代化的面貌。这个系统还可以很容易的扩展到更高级效果的主体上, 就用这个粒子生成器自由的去实验吧, 看看你是否可以创建出你自己的特效。

后期处理

原文 [Postprocessing](#)

作者 JoeydeVries

翻译 [包纸](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

如果我们可以通过几个后期处理(Postprocess)特效丰富Breakout游戏的视觉效果的话，会不会是一件很有趣的事情？利用OpenGL的帧缓冲，我们可以相对容易地创造出模糊的抖动效果、反转场景里的所有颜色、做一些“疯狂”的顶点运动、或是使用一些其他有趣的特效。

这章教程广泛运用了之前[帧缓冲与抗锯齿](#)章节的概念。

在教程的帧缓冲章节里，我们演示了如何使用单个纹理，通过后期处理特效实现有趣的效果（反相、灰度、模糊、锐化、边缘检测）。在Breakout中我们将做一些类似的事情：我们会创建一个帧缓冲对象，并附带一个多重采样的渲染缓冲对象作为其颜色附件。游戏中所有的渲染相关代码都应该渲染至这个多重采样的帧缓冲，然后将其内容传输([Bit blit](#))至一个不同的帧缓冲中，该帧缓冲用一个纹理作为其颜色附件。这个纹理会包含游戏的渲染后的抗锯齿图像，我们对它应用零或多个后期处理特效后渲染至一个大的2D四边形。（译注：这段表述的复杂流程与教程帧缓冲章节的内容相似，原文中包含大量易混淆的名词与代词，建议读者先仔细理解帧缓冲章节的内容与流程）。

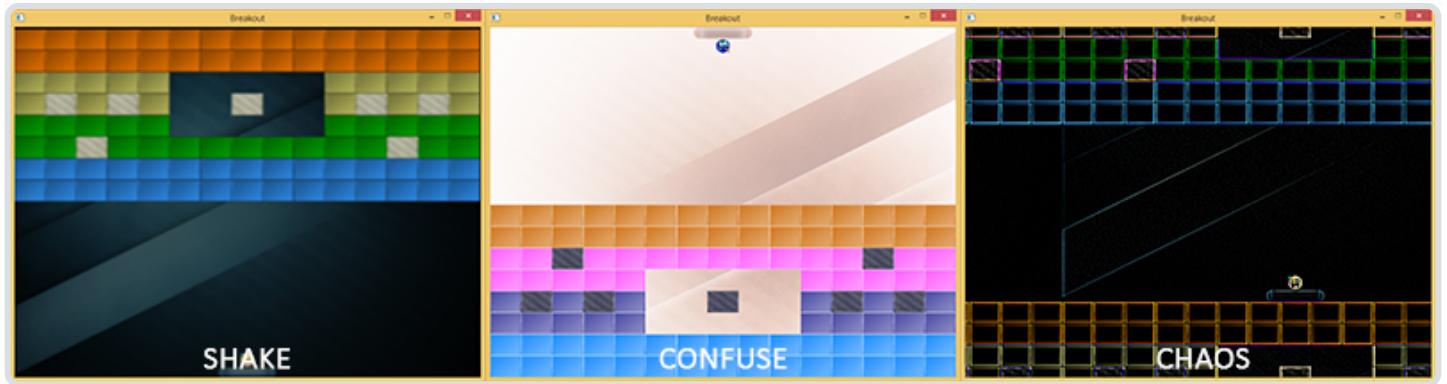
总结一下这些渲染步骤：

1. 绑定至多重采样的帧缓冲
2. 和往常一样渲染游戏
3. 将多重采样的帧缓冲内容传输至一个普通的帧缓冲中（这个帧缓冲使用了一个纹理作为其颜色缓冲附件）
4. 解除绑定（绑定回默认的帧缓冲）
5. 在后期处理着色器中使用来自普通帧缓冲的颜色缓冲纹理
6. 渲染屏幕大小的四边形作为后期处理着色器的输出

我们的后期处理着色器允许使用三种特效：**shake**, **confuse**和**chaos**。

- **shake**: 轻微晃动场景并附加一个微小的模糊效果。
- **shake**: 反转场景中的颜色并颠倒x轴和y轴。
- **chaos**: 利用边缘检测卷积核创造有趣的视觉效果，并以圆形旋转动画的形式移动纹理图片，实现“混沌”特效。

以下是这些效果的示例：



在2D四边形上操作的顶点着色器如下所示：

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 position, vec2 texCoords>

out vec2 TexCoords;

uniform bool chaos;
uniform bool confuse;
uniform bool shake;
uniform float time;

void main()
{
    gl_Position = vec4(vertex.xy, 0.0f, 1.0f);
    vec2 texture = vertex.zw;
    if(chaos)
    {
        float strength = 0.3;
        vec2 pos = vec2(texture.x + sin(time) * strength, texture.y + cos(time) * strength);
        TexCoords = pos;
    }
    else if(confuse)
    {
        TexCoords = vec2(1.0 - texture.x, 1.0 - texture.y);
    }
    else
    {
        TexCoords = texture;
    }
    if (shake)
    {
        float strength = 0.01;
        gl_Position.x += cos(time * 10) * strength;
        gl_Position.y += cos(time * 15) * strength;
    }
}
```

基于uniform是否被设置为true，顶点着色器可以执行不同的分支。如果`chaos`或`confuse`被设置为true，顶点着色器将操纵纹理坐标来移动场景（以圆形动画变换纹理坐标或反转纹理坐标）。因为我们将纹理环绕方式设置为了`GL_REPEAT`，所以`chaos`特效会导致场景在四边形的各个部分重复。除此之外，如果`shake`被设置为true，它将微量移动顶点位置。需要注意的是，`chaos`与`confuse`不应同时为true，而`shake`则可以与其他特效一起生效。

当任意特效被激活时，除了偏移顶点的位置和纹理坐标，我们也希望创造显著的视觉效果。我们可以在片段着色器中实现这一点：

```
#version 330 core
in vec2 TexCoords;
out vec4 color;

uniform sampler2D scene;
uniform vec2      offsets[9];
uniform int       edge_kernel[9];
uniform float     blur_kernel[9];

uniform bool chaos;
uniform bool confuse;
uniform bool shake;

void main()
{
    color = vec4(0.0f);
    vec3 sample[9];
    // 如果使用卷积矩阵，则对纹理的偏移像素进行采样
    if(chaos || shake)
        for(int i = 0; i < 9; i++)
            sample[i] = vec3(texture(scene, TexCoords.st + offsets[i]));

    // 处理特效
    if(chaos)
    {
        for(int i = 0; i < 9; i++)
            color += vec4(sample[i] * edge_kernel[i], 0.0f);
        color.a = 1.0f;
    }
    else if(confuse)
    {
        color = vec4(1.0 - texture(scene, TexCoords).rgb, 1.0);
    }
    else if(shake)
    {
        for(int i = 0; i < 9; i++)
            color += vec4(sample[i] * blur_kernel[i], 0.0f);
        color.a = 1.0f;
    }
    else
    {
        color = texture(scene, TexCoords);
    }
}
```

这个着色器几乎直接构建自帧缓冲教程的片段着色器，并根据被激活的特效类型进行相应的后期处理。这一次，偏移矩阵(offset matrix)和卷积核作为uniform变量，由应用程序中的代码定义。好处是我们只需要设置这些内容一次，而不必在每个片段着色器执行时重新计算这些矩阵。例如，偏移矩阵的配置如下所示：

```
GLfloat offset = 1.0f / 300.0f;
GLfloat offsets[9][2] = {
    { -offset,  offset }, // 左上
    {  0.0f,   offset }, // 中上
    {  offset,  offset }, // 右上
    { -offset,  0.0f }, // 左中
    {  0.0f,   0.0f }, // 正中
    {  offset,  0.0f }, // 右中
    { -offset, -offset }, // 左下
    {  0.0f,  -offset }, // 中下
    {  offset, -offset } // 右下
};
glUniform2fv(glGetUniformLocation(shader.ID, "offsets"), 9, (GLfloat*)offsets);
```

由于所有管理帧缓冲器的概念已经在之前的教程中有过广泛的讨论，所以这次我不会深入其细节。下面是PostProcessor类的代码，它管理初始化，读写帧缓冲并将一个四边形渲染至屏幕。如果你理解了帧缓冲与反锯齿章节的教程，你应该可以完全它的代码。

- PostProcessor: [头文件](#), [代码](#)

有趣的是BeginRender和EndRender函数。由于我们必须将整个游戏场景渲染至帧缓冲中，因此我们可以在场景的渲染代码之前和之后分别调用BeginRender和EndRender。接着，这个类将处理幕后的帧缓冲操作。在游戏的渲染函数中使用PostProcessor类如下所示：

```
PostProcessor *Effects;

void Game::Render()
{
    if (this->State == GAME_ACTIVE)
    {
        Effects->BeginRender();
        // 绘制背景
        // 绘制关卡
        // 绘制挡板
        // 绘制粒子
        // 绘制小球
        Effects->EndRender();
        Effects->Render(glfwGetTime());
    }
}
```

无论我们需要什么，我们只需要将需要的PostProcessor类中的特效属性设置为true，其效果就可以立即可见。

Shake it

作为这些效果的演示，我们将模拟球击中坚固的混凝土块时的视觉冲击。无论在哪里发生碰撞，只要在短时间内实现晃动(shake)效果，便能增强撞击的冲击感。

我们只想允许晃动效果持续一小段时间。我们可以通过声明一个持有晃动效果持续时间的变量 `ShakeTime` 来实现这个功能。无论碰撞何时发生，我们将这个变量重置为一个特定的持续时间：

```
GLfloat ShakeTime = 0.0f;

void Game::DoCollisions()
{
    for (GameObject &box : this->Levels[this->Level].Bricks)
    {
        if (!box.Destroyed)
        {
            Collision collision = CheckCollision(*Ball, box);
            if (std::get<0>(collision)) // 如果发生了碰撞
            {
                // 如果不是实心的砖块则摧毁
                if (!box.IsSolid)
                    box.Destroyed = GL_TRUE;
                else
                { // 如果是实心的砖块则激活shake特效
                    ShakeTime = 0.05f;
                    Effects->Shake = true;
                }
                [...]
            }
        }
    }
    [...]
}
```

然后在游戏的 `Update` 函数中，我们减少 `ShakeTime` 变量的值直到其为 0.0，并停用 shake 特效。

```
void Game::Update(GLfloat dt)
{
    [...]
    if (ShakeTime > 0.0f)
    {
        ShakeTime -= dt;
        if (ShakeTime <= 0.0f)
            Effects->Shake = false;
    }
}
```

这样，每当我们碰到一个实心砖块时，屏幕会短暂地抖动与模糊，给玩家一些小球与坚固物体碰撞的视觉反馈。

你可以在[这里](#)找到更新后的 `Game` 类。

在[下一章](#)关于“道具”的教程中我们将带来另外两种的特效的使用。

道具

原文 [Powerups](#)

作者 JoeydeVries

翻译 [包纸](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

Breakout已经接近完成了，但我们可以至少再增加一种游戏机制让它变得更酷。“充电”（译注：Powerups，很多游戏中都会用这个单词指代可以提升能力的道具，本文之后也会用道具一词作为其翻译）怎么样？

这个想法的含义是，无论一个砖块何时被摧毁，它都有一定几率产生一个道具块。这样的道具块会缓慢降落，而且当它与玩家挡板发生接触时，会发生基于道具类型的有趣效果。例如，某一种道具可以让玩家挡板变长，另一种道具则可以让小球穿过物体。我们还可以添加一些可以给玩家造成负面影响的负面道具。

我们可以将道具建模为具有一些额外属性的`GameObject`，这也是为什么我们定义一个继承自`GameObject`的`PowerUp`类并在其中增添了一些额外的成员属性。

```
const glm::vec2 SIZE(60, 20);
const glm::vec2 VELOCITY(0.0f, 150.0f);

class PowerUp : public GameObject
{
public:
    // 道具类型
    std::string Type;
    GLfloat Duration;
    GLboolean Activated;
    // 构造函数
    PowerUp(std::string type, glm::vec3 color, GLfloat duration,
            glm::vec2 position, Texture2D texture)
        : GameObject(position, SIZE, texture, color, VELOCITY),
          Type(type), Duration(duration), Activated()
    {}
};
```

`PowerUp`类仅仅是一个有一些额外状态的`GameObject`，所以我们简单地将它定义为一个头文件，你可以在[这里](#)找到它。

每个道具以字符串的形式定义它的类型，持有表示它有效时长的持续时间与表示当前是否被激活的属性。在Breakout中，我们将添加4种增益道具与2种负面道具：



- **Speed:** 增加小球20%的速度
- **Sticky:** 当小球与玩家挡板接触时，小球会保持粘在挡板上的状态直到再次按下空格键，这可以让玩家在释放小球前找到更合适的位置
- **Pass-Through:** 非实心砖块的碰撞处理被禁用，使小球可以穿过并摧毁多个砖块
- **Pad-Size-Increase:** 增加玩家挡板50像素的宽度
- **Confuse:** 短时间内激活confuse后期特效，迷惑玩家
- **Chaos:** 短时间内激活chaos后期特效，使玩家迷失方向

你可以在下面找到道具的高质量纹理：

- **Texture:** [Speed](#), [Sticky](#), [Pass-Through](#), [Pad-Size-Increase](#), [Confuse](#), [Chaos](#).

与关卡中的砖块纹理类似，每个道具纹理都是完全灰度的，这使得我们在将其与颜色向量相乘时可以保持色彩的平衡。

因为我们需要跟踪游戏中被激活的道具的类型、持续时间、相关效果等状态，所以我们将它们存储在一个容器内：

```
class Game {
public:
    [...]
    std::vector<PowerUp> PowerUps;
    [...]
    void SpawnPowerUps(GameObject &block);
    void UpdatePowerUps(GLfloat dt);
};
```

我们还定义了两个管理道具的函数，`SpawnPowerUps`在给定的砖块位置生成一个道具，`UpdatePowerUps`管理所有当前被激活的道具。

SpawnPowerUps

每次砖块被摧毁时我们希望以一定几率生成一个道具，这个功能可以在Game的SpawnPowerUps函数中找到：

```
GLboolean ShouldSpawn(GLuint chance)
{
    GLuint random = rand() % chance;
    return random == 0;
}
void Game::SpawnPowerUps(GameObject &block)
{
    if (ShouldSpawn(75)) // 1/75的几率
        this->PowerUps.push_back(
            PowerUp("speed", glm::vec3(0.5f, 0.5f, 1.0f), 0.0f, block.Position, tex_speed
        ));
    if (ShouldSpawn(75))
        this->PowerUps.push_back(
            PowerUp("sticky", glm::vec3(1.0f, 0.5f, 1.0f), 20.0f, block.Position, tex_sticky
        );
    if (ShouldSpawn(75))
        this->PowerUps.push_back(
            PowerUp("pass-through", glm::vec3(0.5f, 1.0f, 0.5f), 10.0f, block.Position, tex_pass
        ));
    if (ShouldSpawn(75))
        this->PowerUps.push_back(
            PowerUp("pad-size-increase", glm::vec3(1.0f, 0.6f, 0.4), 0.0f, block.Position, tex_size
        ));
    if (ShouldSpawn(15)) // 负面道具被更频繁地生成
        this->PowerUps.push_back(
            PowerUp("confuse", glm::vec3(1.0f, 0.3f, 0.3f), 15.0f, block.Position, tex_confuse
        );
    if (ShouldSpawn(15))
        this->PowerUps.push_back(
            PowerUp("chaos", glm::vec3(0.9f, 0.25f, 0.25f), 15.0f, block.Position, tex_chaos
        );
}
```

这样的SpawnPowerUps函数以一定几率（1/75普通道具，1/15负面道具）生成一个新的PowerUp对象，并设置其属性。每种道具具有特殊的颜色使它们更具有辨识度，同时根据类型决定其持续时间的秒数，若值为0.0f则表示它持续无限长的时间。除此之外，每个道具初始化时传入被摧毁砖块的位置与上一小节给出的对应纹理。

激活道具

接下来我们更新游戏的DoCollisions函数使它不只检查小球与砖块和挡板的碰撞，还检查挡板与所有未被销毁的道具的碰撞。注意我们在砖块被摧毁的同时调用SpawnPowerUps函数。

```
void Game::DoCollisions()
{
    for (GameObject &box : this->Levels[this->Level].Bricks)
    {
        if (!box.Destroyed)
        {
            Collision collision = CheckCollision(*Ball, box);
            if (std::get<0>(collision))
            {
                if (!box.IsSolid)
                {
                    box.Destroyed = GL_TRUE;
                    this->SpawnPowerUps(box);
                }
                [...]
            }
        }
    }
    [...]
    for (PowerUp &powerUp : this->PowerUps)
    {
        if (!powerUp.Destroyed)
        {
            if (powerUp.Position.y >= this->Height)
                powerUp.Destroyed = GL_TRUE;
            if (CheckCollision(*Player, powerUp))
            {
                // 道具与挡板接触，激活它！
                ActivatePowerUp(powerUp);
                powerUp.Destroyed = GL_TRUE;
                powerUp.Activated = GL_TRUE;
            }
        }
    }
}
```

对所有未被销毁的道具，我们检查它是否接触到了屏幕底部或玩家挡板，无论哪种情况我们都销毁它，但当道具与玩家挡板接触时，激活这个道具。

激活道具的操作可以通过将其`Activated`属性设为true来完成，实现其效果则需要将它传给`ActivatePowerUp`函数：

```
void ActivatePowerUp(PowerUp &powerUp)
{
    // 根据道具类型发动道具
    if (powerUp.Type == "speed")
    {
        Ball->Velocity *= 1.2;
    }
    else if (powerUp.Type == "sticky")
    {
        Ball->Sticky = GL_TRUE;
        Player->Color = glm::vec3(1.0f, 0.5f, 1.0f);
    }
    else if (powerUp.Type == "pass-through")
    {
        Ball->PassThrough = GL_TRUE;
        Ball->Color = glm::vec3(1.0f, 0.5f, 0.5f);
    }
    else if (powerUp.Type == "pad-size-increase")
    {
        Player->Size.x += 50;
    }
    else if (powerUp.Type == "confuse")
    {
        if (!Effects->Chaos)
            Effects->Confuse = GL_TRUE; // 只在chaos未激活时生效, chaos同理
    }
    else if (powerUp.Type == "chaos")
    {
        if (!Effects->Confuse)
            Effects->Chaos = GL_TRUE;
    }
}
```

`ActivatePowerUp`的目的正如其名称，它按本章教程之前所预设的那样激活了一个道具的效果。我们检查道具的类型并相应地改变游戏状态。对于Sticky和Pass-through效果，我们也相应地改变了挡板和小球的颜色来给玩家一些当前被激活了哪种效果的反馈。

因为Sticky和Pass-through效果稍微改变了一些原有的游戏逻辑，所以我们将这些效果作为属性存储在小球对象中，这样我们可以根据小球当前激活了什么效果而改变游戏逻辑。我们只在`BallObject`的头文件中增加了两个属性，但为了完整性下面给出了更新后的代码：

- `GameObject`: [头文件](#), [代码](#)

这样我们可以通过改动`DoCollisions`函数中小球与挡板碰撞的代码便捷地实现Sticky效果。

```
if (!Ball->Stuck && std::get<0>(result))
{
    [...]
    Ball->Stuck = Ball->Sticky;
}
```

在这里我们将小球的`Stuck`属性设置为它自己的`Sticky`属性，若`Stikcy`效果被激活，那么小球则会在与挡板接触时粘在上面，玩家不得不再次按下空格键才能释放它。

在同样的DoCollisions函数中还有个为了实现Pass-through效果的类似小改动。当小球的PassThrough属性被设置为true时，我们不对非实习砖块做碰撞处理操作。

```
Direction dir = std::get<1>(collision);
glm::vec2 diff_vector = std::get<2>(collision);
if (!(Ball->PassThrough && !box.IsSolid))
{
    if (dir == LEFT || dir == RIGHT) // 水平碰撞
    {
        [...]
    }
    else
    {
        [...]
    }
}
```

其他效果可以通过简单的更改游戏的状态来实现，如小球的速度、挡板的尺寸、PostProcessor对象的效果。

更新道具

现在剩下要做的就是保证道具生成后可以移动，并且在它们的持续时间用尽后失效，否则道具将永远保持激活状态。

在游戏的`UpdatePowerUps`函数中，我们根据道具的速度移动它，并减少已激活道具的持续时间，每当时间减少至小于0时，我们令其失效，并恢复相关变量的状态。

```
void Game::UpdatePowerUps(GLfloat dt)
{
    for (PowerUp &powerUp : this->PowerUps)
    {
        powerUp.Position += powerUp.Velocity * dt;
        if (powerUp.Activated)
        {
            powerUp.Duration -= dt;

            if (powerUp.Duration <= 0.0f)
            {
                // 之后会将这个道具移除
                powerUp.Activated = GL_FALSE;
                // 停用效果
                if (powerUp.Type == "sticky")
                {
                    if (!isOtherPowerUpActive(this->PowerUps, "sticky"))
                    {
                        // 仅当没有其他sticky效果处于激活状态时重置，以下同理
                        Ball->Sticky = GL_FALSE;
                        Player->Color = glm::vec3(1.0f);
                    }
                }
                else if (powerUp.Type == "pass-through")
                {
                    if (!isOtherPowerUpActive(this->PowerUps, "pass-through"))
                    {
                        Ball->PassThrough = GL_FALSE;
                        Ball->Color = glm::vec3(1.0f);
                    }
                }
                else if (powerUp.Type == "confuse")
                {
                    if (!isOtherPowerUpActive(this->PowerUps, "confuse"))
                    {
                        Effects->Confuse = GL_FALSE;
                    }
                }
                else if (powerUp.Type == "chaos")
                {
                    if (!isOtherPowerUpActive(this->PowerUps, "chaos"))
                    {
                        Effects->Chaos = GL_FALSE;
                    }
                }
            }
        }
    }

    this->PowerUps.erase(std::remove_if(this->PowerUps.begin(), this->PowerUps.end(),
        [] (const PowerUp &powerUp) { return powerUp.Destroyed && !powerUp.Activated; })
    , this->PowerUps.end());
}
```

你可以看到对于每个效果，我们通过将相关元素重置来停用它。我们还将`PowerUp`的`Activated`属性设为`false`，在`UpdatePowerUps`结束时，我们通过循环`PowerUps`容器，若一个道具被销毁则被停用，则移除它。我们在算法开头使用`remove_if`函数，通过给定的lambda表达式消除这些对象。

`remove_if`函数将lambda表达式为true的元素移动至容器的末尾并返回一个迭代器指向应被移除的元素范围的开始部分。容器的`erase`函数接着擦除这个迭代器指向的元素与容器末尾元素之间的所有元素。

可能会发生这样的情况：当一个道具在激活状态时，另一个道具与挡板发生了接触。在这种情况下我们有超过1个在当前`PowerUps`容器中处于激活状态的道具。然后，当这些道具中的一个被停用时，我们不应使其效果失效因为另一个相同类型的道具仍处于激活状态。出于这个原因，我们使用`isOtherPowerUpActive`检查是否有同类道具处于激活状态。只有当它返回false时，我们才停用这个道具的效果。这样，给定类型的道具的持续时间就可以延长至最近一次被激活后的持续时间。

```
GLboolean IsOtherPowerUpActive(std::vector<PowerUp> &powerUps, std::string type)
{
    for (const PowerUp &powerUp : powerUps)
    {
        if (powerUp.Activated)
            if (powerUp.Type == type)
                return GL_TRUE;
    }
    return GL_FALSE;
}
```

这个函数简单地检查是否有同类道具处于激活状态，如果有则返回`GL_TRUE`。

最后剩下的一件事便是渲染道具：

```
void Game::Render()
{
    if (this->State == GAME_ACTIVE)
    {
        [...]
        for (PowerUp &powerUp : this->PowerUps)
            if (!powerUp.Destroyed)
                powerUp.Draw(*Renderer);
        [...]
    }
}
```

结合所有的这些功能，我们有了一个可以运作的道具系统，它不仅使游戏更有趣，还使游戏更具有挑战性。它看上去会像这样：

你可以在下面找到所有更新后的代码（当关卡重置时我们同时重置所有道具效果）：

- Game: [头文件](#), [代码](#)

音效

原文 [Audio](#)

作者 JoeydeVries

翻译 [包纸](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

无论我们将游戏音量调到多大，我们都不会听到来自游戏的任何音效。我们已经展示了这么多内容，但没有任何音频，游戏仍显得有些空洞。在本节教程中，我们将解决这个问题。

OpenGL不提供关于音频的任何支持。我们不得不手动将音频加载为字节格式，处理并将其转化为音频流，并适当地管理多个音频流以供我们的游戏使用。然而这有一些复杂，并且需要一些底层的音频工程知识。

如果你乐意，你可以手动加载来自多种扩展名的音频文件的音频流。然而，我们将使用被称为irrKlang的音频管理库。

irrKlang

IrrKlang是一个可以播放WAV，MP3，OGG和FLAC文件的高级二维和三维（Windows，Mac OS X，Linux）声音引擎和音频库。它还有一些可以自由调整的音频效果，如混响、延迟和失真。



3D音频意味着音频源可以有一个3D位置，然后根据相机到音频源的位置衰减音量，使其在一个3D世界里显得自然（想想3D世界中的枪声，通常你可以从音效中听出它来自什么方向/位置）。

IrrKlang是一个易于使用的音频库，只需几行代码便可播放大多数音频文件，这使它成为我们Breakout游戏的完美选择。请注意，irrKlang有一个有一定限制的证书：允许你将irrKlang用于非商业目的，但是如果你想使用irrKlang商业版，就必须支付购买他们的专业版。由于Breakout和本教程系列是非商业性的，所以我们可以自由地使用他们的标准库。

你可以从他们的[下载页面](#)下载irrKlang，我们将使用1.5版本。由于irrKlang是非开源的代码，因此我们不得不使用irrKlang为我们提供的任何东西。幸运的是，他们有大量的预编译库文件，所以你们大多数人应该可以很好地使用它。

你需要引入了irrKlang的头文件，将他们的库文件（irrKlang.lib）添加到链接器设置中，并将他们的dll文件复制到适当的目录下（通常和.exe在同一目录下）。需要注意的是，如果你想要加载MP3文件，则还需要引入ikpMP3.dll文件。

添加音乐

为了这个游戏我特制了一个小小的音轨，让游戏更富有活力。在[这里](#)你可以找到我们将要用作游戏背景音乐的音轨。这个音轨会在游戏开始时播放并不断循环直到游戏结束。你可以随意用自己的音频替换它，或者用喜欢的方式使用它。

利用IrrKlang库将其添加到Breakout游戏里非常简单。我们引入相应的头文件，创建`irrklang::ISoundEngine`，用`createIrrKlangDevice`初始化它并使用这个引擎加载、播放音频：

```
#include <irrklang/irrklang.h>
using namespace irrklang;

ISoundEngine *SoundEngine = createIrrKlangDevice();

void Game::Init()
{
    [...]
    SoundEngine->play2D("audio/breakout.mp3", GL_TRUE);
}
```

在这里，我们创建了一个`SoundEngine`，用于管理所有与音频相关的代码。一旦我们初始化了引擎，便可以调用`play2D`函数播放音频。第一个参数为文件名，第二个参数为是否循环播放。

这就是全部了！现在运行游戏会使你的耳机或扬声器迸发出声波。

添加音效

我们还没有结束，因为音乐本身并不能使游戏完全充满活力。我们希望在游戏发生一些有趣事件时播放音效，作为给玩家的额外反馈，如我们撞击砖块、获得道具时。下面你可以找到我们需要的所有音效（来自freesound.org）：

[bleep.mp3](#): 小球撞击非实心砖块时的音效

[solid.wav](#): 小球撞击实心砖块时的音效

[powerup.wav](#): 获得道具时的音效

[bleep.wav](#): 小球在挡板上反弹时的音效

无论在哪里发生碰撞，我们都会播放相应的音效。我不会详细阐述每一行的代码，并把更新后的代码放在了[这里](#)，你应该可以轻松地找到相应的添加音效的地方。

把这些集成在一起后我们的游戏显得更完整了，就像这样：

IrrKlang允许一些更精细的音频管理功能，如内存管理、声音特效和声音事件回调。看看他们的C++[教程](#)并尝试一些功能吧。

渲染文本

原文 [Render text](#)

作者 JoeydeVries

翻译 [aillieo](#)

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

本教程中将通过增加生命值系统、获胜条件和渲染文本形式的反馈来对游戏做最后的完善。本教程很大程度上是建立在之前的教程[文本渲染](#)基础之上，因此如果没有看过的话，强烈建议您先一步一步学习之前的教程。

在Breakout中，所有的文本渲染代码都封装在一个名为[TextRendererer](#)的类中，其中包含FreeType库的初始化、渲染配置和实际渲染代码等重要组成部分。以下是[TextRendererer](#)类的代码：

- [TextRendererer](#): [header](#), [code](#).
- [Text shaders](#): [vertex](#), [fragment](#).

文本渲染器中函数的内容几乎与文本渲染教程中的代码完全一样。但用于向屏幕渲染字形的代码稍有不同：

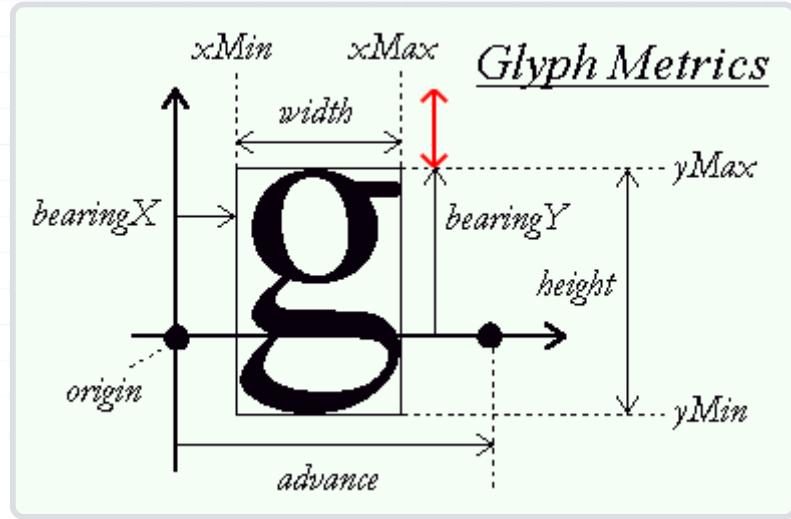
```
void TextRenderer::RenderText(std::string text, GLfloat x, GLfloat y, GLfloat scale,glm::vec3 color)
{
    [...]
    for (c = text.begin(); c != text.end(); c++)
    {
        GLfloat xpos = x + ch.Bearing.x * scale;
        GLfloat ypos = y + (this->Characters['H'].Bearing.y - ch.Bearing.y) * scale;

        GLfloat w = ch.Size.x * scale;
        GLfloat h = ch.Size.y * scale;
        // 为每个字符更新VBO
        GLfloat vertices[6][4] = {
            { xpos,      ypos + h,      0.0, 1.0 },
            { xpos + w,   ypos,      1.0, 0.0 },
            { xpos,      ypos,      0.0, 0.0 },

            { xpos,      ypos + h,      0.0, 1.0 },
            { xpos + w,   ypos + h,      1.0, 1.0 },
            { xpos + w,   ypos,      1.0, 0.0 }
        };
        [...]
    }
}
```

之所以会稍有不同，是因为相比于文本渲染教程我们此处使用了一个不同的正交投影矩阵。在文本渲染教程中，所有的`y`值取值从底部向顶部递增，但在游戏Breakout中，`y`值取值从顶部到底部递增，值为`0.0`的`y`值对应屏幕顶端。这意味着我们需要稍微改变计算垂直方向偏移的方法。

由于现在我们将`RenderText`的y坐标参数从上向下渲染，我们将垂直偏移计算为一个字形从字形空间顶部向下推进的距离。回顾FreeType的字形矩阵图片，此垂直偏移用红色箭头标记。



为计算垂直偏移，我们需要获取字形空间的顶部（基本上是原点发出的黑色垂直箭头的长度）。不幸的是，FreeType并不向我们提供这样的机制。我们已知的是有一些字形直接与顶部接触，如字符'H'、'T'或'X'。那么我们通过接触顶部的字形的`bearingY`减去顶部不确定字形的`bearingY`来计算红色矢量的长度。使用这种方法，我们依据字形顶部的点与顶部边差异的距离来向下推进字形。

```
GLfloat ypos = y + (this->Characters['H'].Bearing.y - ch.Bearing.y) * scale;
```

除了更新`ypos`的计算之外，我们还调换了一些顶点的顺序，用以确保所有的顶点在与现在的正交投影矩阵相乘后仍为正方向（详见教程[面剔除](#)）。

向游戏中加入`TextRenderer`并不难：

```
TextRenderer *Text;
void Game::Init()
{
    [...]
    Text = new TextRenderer(this->Width, this->Height);
    Text->Load("fonts/ocraext.TTF", 24);
}
```

使用一个名为OCR A Extended的字体来初始化文本渲染器，该字体可以从[这里](#)下载。您可以使用任意不同的字体，如果不很喜欢这个字体。

现在我们已经有了一个文本渲染器，接下来我们来完成游戏机制。

玩家生命值

当球到底部边界时，我们会给玩家额外的几次机会，而不是立即重置游戏。我们使用玩家生命值的形式来实现，玩家开始时会有初始数量的生命值（比如3），每当球到底部边界，玩家的生命值总数会减1。只有当玩家生命值变为0时才会重置游戏。这样对玩家来说完成关卡会稍容易一点，同时也会感受到难度。

我们向game类中增加玩家的生命值以记录它（在构造函数中将其初始化为3）。

```
class Game
{
    [...]
public:
    GLuint Lives;
}
```

接下来我们修改game类的update函数，不再重置游戏，而是减少玩家生命值，只有当生命值为0时重置游戏。

```
void Game::Update(GLfloat dt)
{
    [...]
    if (Ball->Position.y >= this->Height) // 球是否接触到底部边界？
    {
        --this->Lives;
        // 玩家是否已失去所有生命值？：游戏结束
        if (this->Lives == 0)
        {
            this->ResetLevel();
            this->State = GAME_MENU;
        }
        this->ResetPlayer();
    }
}
```

一旦玩家游戏结束（lives等于0），我们会重置关卡，并将游戏状态改变为GAME_MENU，稍后会详细讲。

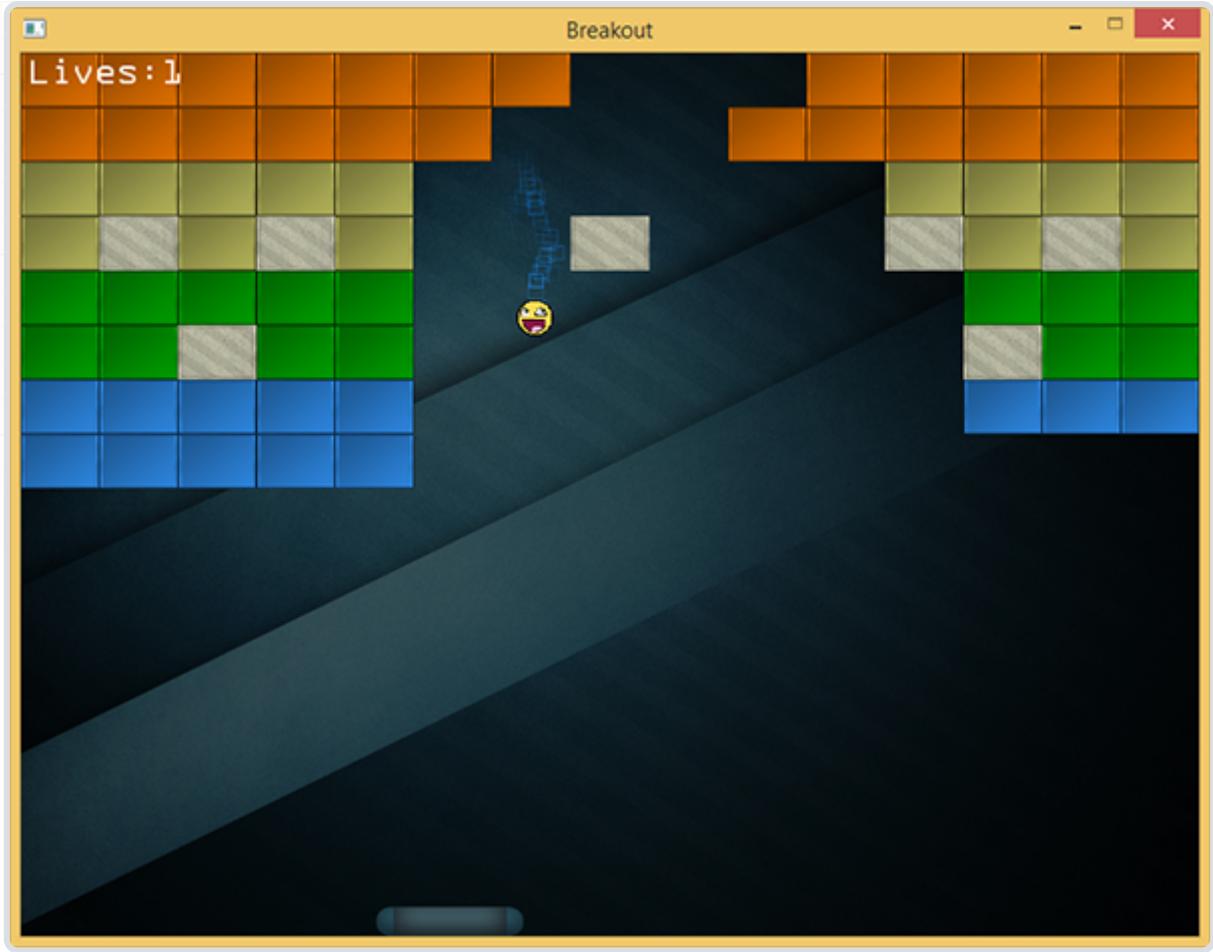
注意不要忘了在重置游戏/关卡时重置玩家生命值：

```
void Game::ResetLevel()
{
    [...]
    this->Lives = 3;
}
```

此时玩家生命值已可以运作，但玩家在游戏时却无法看到自己当前有多少生命值。这时就需要加入文本渲染器。

```
void Game::Render()
{
    if (this->State == GAME_ACTIVE)
    {
        [...]
        std::stringstream ss; ss << this->Lives;
        Text->RenderText("Lives:" + ss.str(), 5.0f, 5.0f, 1.0f);
    }
}
```

这里我们将生命值数量转化为一个字符串，并将其显示在屏幕左上角。看起来将会是像这样：



一旦球接触到底部边界，玩家的生命值会减少，这会在屏幕左上角直接可见。

关卡选择

当玩家所处游戏状态为`GAME_MENU`时，我们希望玩家可以控制选择他想玩的关卡。玩家应该可以使用'w'或's'键在我们加载的所有关卡中滚动选择。当玩家感觉选中的是他想玩的关卡时，他可以按回车键将游戏状态从`GAME_MENU`切换到`GAME_ACTIVE`。

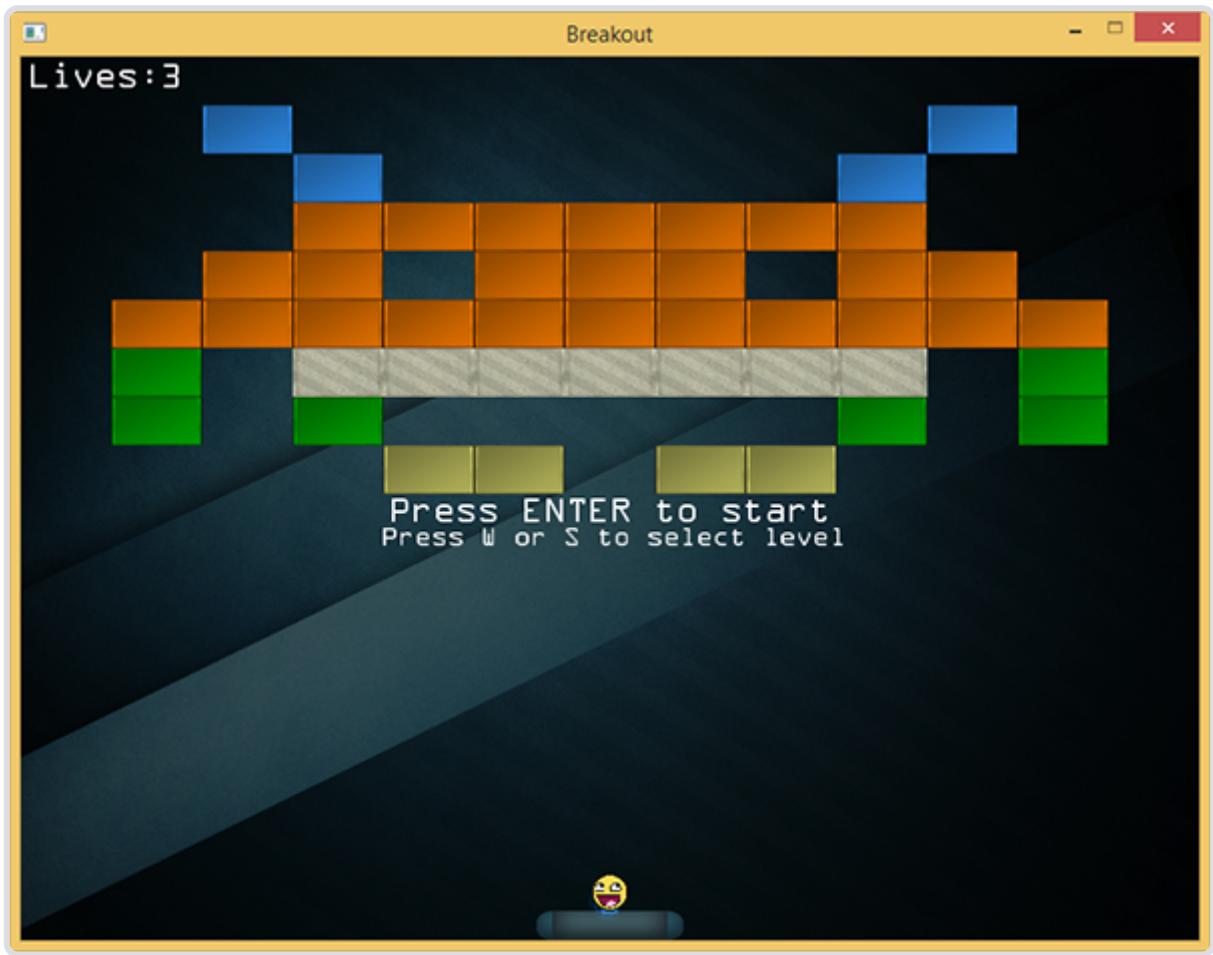
允许玩家选择关卡并不难。我们要做的就是当玩家按下'w'或's'键时分别增加或减小`game`类中的变量`Level`的值：

```
if (this->State == GAME_MENU)
{
    if (this->Keys[GLFW_KEY_ENTER])
        this->State = GAME_ACTIVE;
    if (this->Keys[GLFW_KEY_W])
        this->Level = (this->Level + 1) % 4;
    if (this->Keys[GLFW_KEY_S])
    {
        if (this->Level > 0)
            --this->Level;
        else
            this->Level = 3;
    }
}
```

我们使用取模运算(`%`)以保证变量`Level`在可接受的关卡值范围内（0和3之间）。除了变换关卡之外，在菜单状态时我们还需要将其渲染出来。我们要给玩家一些文本形式的指示，并在背景中展示出选中的关卡。

```
void Game::Render()
{
    if (this->State == GAME_ACTIVE || this->State == GAME_MENU)
    {
        [...] // 游戏状态渲染代码
    }
    if (this->State == GAME_MENU)
    {
        Text->RenderText("Press ENTER to start", 250.0f, Height / 2, 1.0f);
        Text->RenderText("Press W or S to select level", 245.0f, Height / 2 + 20.0f, 0.75f);
    }
}
```

这里无论游戏处在`GAME_ACTIVE`状态还是`GAME_MENU`状态，都会渲染游戏，当游戏处在`GAME_MENU`状态需要渲染两行文本用于告知玩家选择一个关卡并且/或者确认选择。注意，为此必须在启动游戏时将游戏状态默认设置为`GAME_MENU`。



看起来很棒，但当你试图运行代码你很可能会注意到，当按下'w'或's'键时，游戏会在关卡之前快速滚动，很难选中你想玩的关卡。这是因为game会在多帧记录按键直到按键松开。这将导致`ProcessInput`函数处理按下的键不止一次。

我们可以使用GUI系统中常见的一个小技巧来解决这一问题。这一小技巧就是：不仅记录当前按下的键，并且存储已经被按下的键，直到再次松开。然后我们会检查（在处理之前）是否还没有被处理，如果没有被处理的话，处理该按键并将其存储为正在被处理。一旦我们要在未松开时再次处理相同的按键，我们将不会处理该按键。这听起来让人稍微迷惑，但当你在实际应用中见到它，（很可能）就会明白它的意义。

首先我们需要创建另一个布尔数组用来表示处理过的按键。我们在game类定义如下：

```
class Game
{
    [...]
public:
    GLboolean KeysProcessed[1024];
}
```

当相对应的按键被处理时，我们将其设置为`true`，以确保按键只在之前没有被处理过（直到松开）时将其处理。

```
void Game::ProcessInput(GLfloat dt)
{
    if (this->State == GAME_MENU)
    {
        if (this->Keys[GLFW_KEY_ENTER] && !this->KeysProcessed[GLFW_KEY_ENTER])
        {
            this->State = GAME_ACTIVE;
            this->KeysProcessed[GLFW_KEY_ENTER] = GL_TRUE;
        }
        if (this->Keys[GLFW_KEY_W] && !this->KeysProcessed[GLFW_KEY_W])
        {
            this->Level = (this->Level + 1) % 4;
            this->KeysProcessed[GLFW_KEY_W] = GL_TRUE;
        }
        if (this->Keys[GLFW_KEY_S] && !this->KeysProcessed[GLFW_KEY_S])
        {
            if (this->Level > 0)
                --this->Level;
            else
                this->Level = 3;
            this->KeysProcessed[GLFW_KEY_S] = GL_TRUE;
        }
    }
    [...]
}
```

现在，当`KeysProcessed`数组中按键的值未被设置时，我们会处理按键并将其值设为`true`。下一次当到达同一按键的`if`条件时，它已经被处理过了所以我们会假装并没有按下此键，直到它被松开。

之后，我们需要在松开按键时通过GLFW的按键回调函数，重置按键处理后的值，以便于下次再处理：

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    [...]
    if (key >= 0 && key < 1024)
    {
        if (action == GLFW_PRESS)
            Breakout.Keys[key] = GL_TRUE;
        else if (action == GLFW_RELEASE)
        {
            Breakout.Keys[key] = GL_FALSE;
            Breakout.KeysProcessed[key] = GL_FALSE;
        }
    }
}
```

启动游戏会展示一个整洁的关卡选择界面，现在可以每次按键清晰地选择一个关卡，无论按键被按下多久。

获胜

现在玩家可以选择关卡、玩游戏和游戏失败。有些不幸的是玩家在消除了所有的砖块之后会发现无法获得游戏胜利。现在来修复此问题。

在所有实体砖块被消除之后玩家会取得胜利。我们已经在[GameLevel](#)类创建了一个函数用于检查这一条件：

```
GLboolean GameLevel::IsCompleted()
{
    for (GameObject &tile : this->Bricks)
        if (!tile.IsSolid && !tile.Destroyed)
            return GL_FALSE;
    return GL_TRUE;
}
```

我们会检查游戏关卡所有的砖块，如果有一个非实体砖块未被消除则返回`false`。我们所要做的就是在[Update](#)函数中检查此条件，一旦返回`true`，我们就将游戏状态改变为[GAME_WIN](#)：

```
void Game::Update(GLfloat dt)
{
    [...]
    if (this->State == GAME_ACTIVE && this->Levels[this->Level].IsCompleted())
    {
        this->ResetLevel();
        this->ResetPlayer();
        Effects->Chaos = GL_TRUE;
        this->State = GAME_WIN;
    }
}
```

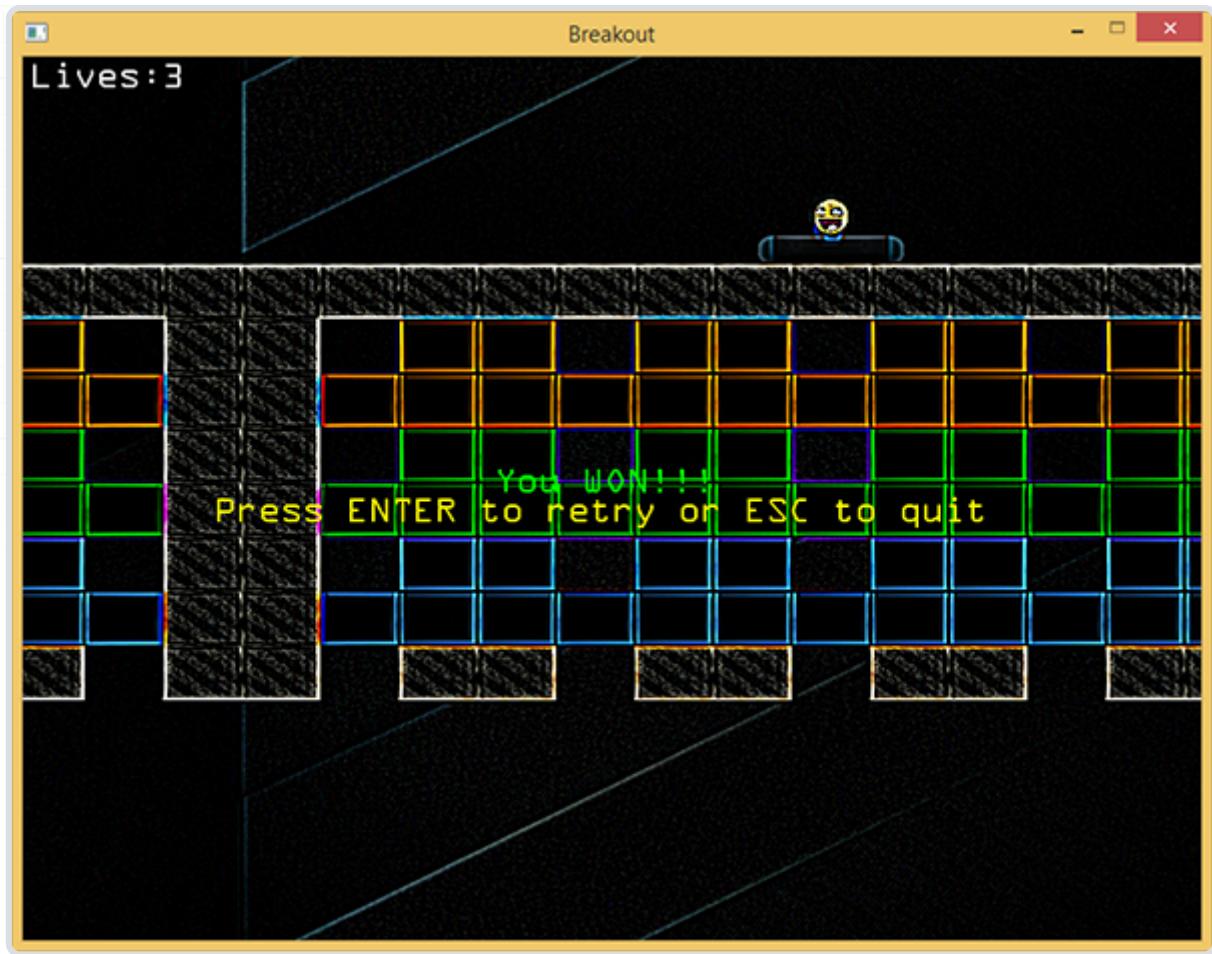
在游戏激活状态，当关卡完成时，我们会重置游戏并且在`GAME_WIN`状态展示一条小的胜利的消息。为了增加趣味性我们会在`GAME_WIN`界面启用混沌效应。在函数`Render`中我们会祝贺玩家并询问其重新开始还是退出游戏。

```
void Game::Render()
{
    [...]
    if (this->State == GAME_WIN)
    {
        Text->RenderText(
            "You WON!!!", 320.0, Height / 2 - 20.0, 1.0, glm::vec3(0.0, 1.0, 0.0)
        );
        Text->RenderText(
            "Press ENTER to retry or ESC to quit", 130.0, Height / 2, 1.0, glm::vec3(1.0, 1.0, 0.0)
        );
    }
}
```

之后我们当然也要处理之前提到的按键：

```
void Game::ProcessInput(GLfloat dt)
{
    [...]
    if (this->State == GAME_WIN)
    {
        if (this->Keys[GLFW_KEY_ENTER])
        {
            this->KeysProcessed[GLFW_KEY_ENTER] = GL_TRUE;
            Effects->Chaos = GL_FALSE;
            this->State = GAME_MENU;
        }
    }
}
```

之后如果你真的可以赢得游戏，你将会看到以下图片：



就是这样！这是游戏Breakout我们要处理的最后一块拼图。尝试按照自己的意愿自定义并把它展示给你的家人和朋友！

以下是最终版的游戏代码：

- Game: [header](#), [code](#).

结语

原文 [Final thoughts](#)

作者 JoeydeVries

翻译 包纸

校对 暂无

Note

本节暂未进行完全的重写，错误可能会很多。如果可能的话，请对照原文进行阅读。如果有报告本节的错误，将会延迟至重写之后进行处理。

与仅仅是用OpenGL创建一个技术演示相比，这一整章的教程给我们了一次体验在此之上的更多内容的机会。我们从零开始制作了一个2D游戏，并学习了如何对特定的底层图形学概念进行抽象、使用基础的碰撞检测技术、创建粒子、展示基于正射投影矩阵的场景。所有的这些都使用了之前教程中讨论过的概念。我们并没有真正地学习和使用OpenGL中新的、令人兴奋的图形技术，更多的是在将所有知识整合至一个更大的整体中。

Breakout这样的一个简单游戏的制作可以被数千种方法完成，而我们的做法也只是其中之一。随着游戏越来越庞大，你开始应用的抽象思想与设计模式就会越多。如果希望进行更深入的学习与阅读，你可以在[game programming patterns](#)找到大部分的抽象思想与设计模式。（译注：《游戏编程模式》一书国内已有中文翻译版，GPP翻译组译，人民邮电出版社）

请记住，编写出一个有着非常干净、考虑周全的代码的游戏是一件很困难的任务（几乎不可能）。你只需要在编写游戏时使用在当时你认为正确的方法。随着你对视频游戏开发的实践越来越多，你学习的新的、更好地解决问题的方法就越多。不必因为编写“完美”代码的困难感到挫败，坚持编程吧！

优化

这些教程的内容和目前已完成的游戏代码的关注点都在于如何尽可能简单地阐述概念，而没有深入地优化细节。因此，很多性能相关的考虑都被忽略了。为了在游戏的帧率开始下降时可以提高性能，我们将列出一些现代的2D OpenGL游戏中常见的改进方案。

- **渲染精灵表单/纹理图谱(Sprite sheet / Texture atlas)**: 代替使用单个渲染精灵渲染单个纹理的渲染方式，我们将所有需要用到的纹理组合到单个大纹理中（如同位图字体），并用纹理坐标来选择合适的精灵与纹理。切换纹理状态是非常昂贵的操作，而使用这种方法让我们几乎可以不用在纹理间进行切换。除此之外，这样做还可以让GPU更有效率地缓存纹理，获得更快的查找速度。（译注：cache的局部性原理）
- **实例化渲染**: 代替一次只渲染一个四边形的渲染方式，我们可以将想要渲染的所有四边形批量化，并使用**实例化渲染**在一次`<>draw call`中成批地渲染四边形。这很容易实现，因为每个精灵都由相同的顶点组成，不同之处只有一个模型矩阵(Model Matrix)，我们可以很容易地将其包含在一个实例化数组中。这样可以使OpenGL每帧渲染更多的精灵。实例化渲染也可以用来渲染粒子和字符字形。
- **三角形带(Triangle Strips)**: 代替每次渲染两个三角形的渲染方式，我们可以用OpenGL的**TRIANGLE_STRIP**渲染图元渲染它们，只需4个顶点而非6个。这节约了三分之一需要传递给GPU的数据量。
- **空间划分(Space partition)算法**: 当检查可能发生的碰撞时，我们将小球与当前关卡中的每一个砖块进行比较，这么做有些浪费CPU资源，因为我们可以很容易得知在这一帧中，大多数砖块都不会与小球很接近。使用BSP，八叉树(Octress)或k-d(imension)树等空间划分算法，我们可以将可见的空间划分成许多较小的区域，并判断小球是否在这个区域中，从而为我们省去大量的碰撞检查。对于Breakout这样的简单游戏来说，这可能是过度的，但对于有着更复杂的碰撞检测算法的复杂游戏，这些算法可以显著地提高性能。
- **最小化状态间的转换**: 状态间的变化（如绑定纹理或切换着色器）在OpenGL中非常昂贵，因此你需要避免大量的状态变化。一种最小化状态间变化的方法是创建自己的状态管理器来存储OpenGL状态的当前值（比如绑定了哪个纹理），并且只在需要改变时进行切换，这可以避免不必要的状态变化。另外一种方式是基于状态切换对所有需要渲染的物体进行排序。首先渲染使用着色器A的所有对象，然后渲染使用着色器B的所有对象，以此类推。当然这可以扩展到着色器、纹理绑定、帧缓冲切换等。

这些应该可以给你一些关于，我们可以用什么样的的高级技巧进一步提高2D游戏性能地提示。这也让你感受到了OpenGL的强大功能。通过亲手完成大部分的渲染，我们对整个渲染过程有了完整的掌握，从而可以实现对过程的优化。如果你对Breakout的性能并不满意，你可以把这些当做练习。

开始创作！

你已经看到了如何在OpenGL中创建一个简单的游戏，现在轮到你来创作属于自己的渲染/游戏程序了。到目前为止我们讨论的许多技术都可以应用于大部分2D游戏中，如渲染精灵、基础的碰撞检测、后期处理、文本渲染和粒子系统。现在你可以将这些技术以你认为合理的方式进行组合与修改，并开发你自己的手制游戏吧！

2.9 历史存档

由于作者在对教程不断地更新，一些比较旧的内容就从教程中删除或者修改了，但是这部分内容仍然对在更新以前就开始学习的读者可能还是会有点用处的。所以，我会将被大段删除或修改的内容留到这里以供大家参考。

注意的是，一些比较小的改动将不会出现在这里。

2.9.1 01-01 OpenGL

基元类型(Primitive Type)

使用OpenGL时，建议使用OpenGL定义的基元类型。比如使用 `float` 时我们加上前缀 `GL`（因此写作 `GLfloat`）。

`int`、`uint`、`char`、`bool` 等等也类似。OpenGL定义的这些GL基元类型的内存布局是与平台无关的，而`int`等基元类型在不同操作系统上可能有不同的内存布局。使用GL基元类型可以保证你的程序在不同的平台上工作一致。

2.9.2 01-02 创建窗口

编译和链接GLEW

GLEW是OpenGL Extension Wrangler Library的缩写，它能解决我们上面提到的那个繁琐的问题。因为GLEW也是一个库，我们同样需要构建并将其链接进工程。GLEW可以从[这里](#)下载，你同样可以选择下载二进制版本，如果你的目标平台列在上面的话，或者下载源码编译，步骤和编译GLFW时差不多。记住，如果不确定的话，选择32位的二进制版本。

我们使用GLEW的静态版本`glew32s.lib`（注意这里的“s”），将库文件添加到你的库目录，将`include`内容添加到你的`include`目录。接下来，在VS的链接器选项里加上`glew32s.lib`。注意GLFW3（默认）也是编译成了一个静态库。

静态(Static)链接是指编译时就将库代码里的内容整合进你的二进制文件。优点就是你不需要管理额外的文件了，只需要发布你单独的一个二进制文件就行了。缺点就是你的可执行文件会变得更大，另外当库有升级版本时，你必须重新进行编译整个程序。

动态(Dynamic)链接是指一个库通过 `.dll` 或 `.so` 的方式存在，它的代码与你的二进制文件的代码是分离的。优点是使你的二进制文件大小变小并且更容易升级，缺点是你最终发布程序时必须带上这些DLL。

如果你希望静态链接GLEW，必须在包含GLEW头文件之前定义预处理器宏 `GLEW_STATIC`：

```
#define GLEW_STATIC
#include <GL/glew.h>
```

如果你希望动态链接，那么你可以省略这个宏。但是记住使用动态链接的话你需要拷贝一份.DLL文件到你的应用程序目录。

对于用GCC编译的Linux用户建议使用这个命令行选项 `-lGLEW -lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi`。没有正确链接相应的库会产生 *undefined reference*(未定义的引用) 这个错误。

我们现在成功编译了GLFW和GLEW库，我们已经准备好将进入[下一节](#)去真正使用GLFW和GLEW来设置OpenGL上下文并创建窗口。记得确保你的头文件和库文件的目录设置正确，以及链接器里引用的库文件名正确。如果仍然遇到错误，可以先看一下评论有没有人遇到类似的问题，请参考额外资源中的例子或者在下面的评论区提问。

2.9.3 01-03 你好，窗口

GLEW

在之前的教程中已经提到过，GLEW是用来管理OpenGL的函数指针的，所以在调用任何OpenGL的函数之前我们需要初始化GLEW。

```
glewExperimental = GL_TRUE;
if (glewInit() != GLEW_OK)
{
    std::cout << "Failed to initialize GLEW" << std::endl;
    return -1;
}
```

请注意，我们在初始化GLEW之前设置`glewExperimental`变量的值为`GL_TRUE`，这样做能让GLEW在管理OpenGL的函数指针时更多地使用现代化的技术，如果把它设置为`GL_FALSE`的话可能会在使用OpenGL的核心模式时出现一些问题。

视口(Viewport)

在我们开始渲染之前还有一件重要的事情要做，我们必须告诉OpenGL渲染窗口的尺寸大小，这样OpenGL才只能知道怎样相对于窗口大小显示数据和坐标。我们可以通过调用`glViewport`函数来设置窗口的维度(Dimension)：

```
int width, height;
glfwGetFramebufferSize(window, &width, &height);

glViewport(0, 0, width, height);
```

`glViewport`函数前两个参数控制窗口左下角的位置。第三个和第四个参数控制渲染窗口的宽度和高度（像素），这里我们是直接从GLFW中获取的。我们从GLFW中获取视口的维度而不设置为800*600是为了让它在高DPI的屏幕上（比如说Apple的视网膜显示屏）也能[正常工作](#)。

我们实际上也可以将视口的维度设置为比GLFW的维度小，这样子之后所有的OpenGL渲染将会在一个更小的窗口中显示，这样子的话我们也可以将一些其它元素显示在OpenGL视口之外。

OpenGL幕后使用`glViewport`中定义的位置和宽高进行2D坐标的转换，将OpenGL中的位置坐标转换为你的屏幕坐标。例如，OpenGL中的坐标(-0.5, 0.5)有可能（最终）被映射为屏幕中的坐标(200,450)。注意，处理过的OpenGL坐标范围只为-1到1，因此我们事实上将(-1到1)范围内的坐标映射到(0, 800)和(0, 600)。

输入

我们同样也希望能够在GLFW中实现一些键盘控制，这可以通过使用GLFW的回调函数(Callback Function)来完成。**回调函数**事实上是一个函数指针，当我们设置好后，GLFW会在合适的时候调用它。按键回调(KeyCallback)是众多回调函数中的一种。当我们设置了按键回调之后，GLFW会在用户有键盘交互时调用它。该回调函数的原型如下所示：

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode);
```

按键回调函数接受一个`GLFWwindow`指针作为它的第一个参数；第二个整形参数用来表示按下的按键；`action`参数表示这个按键是被按下还是释放；最后一个整形参数表示是否有Ctrl、Shift、Alt、Super等按钮的操作。GLFW会在合适的时候调用它，并为各个参数传入适当的值。

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    // 当用户按下ESC键，我们设置window窗口的WindowShouldClose属性为true
    // 关闭应用程序
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

在我们（新创建的）`key_callback`函数中，我们检测了键盘是否按下了Escape键。如果键的确按下了（不释放），我们使用`glfwSetWindowShouldClose`函数设定`WindowShouldClose`属性为`true`从而关闭GLFW。main函数的`while`循环下一次的检测将为失败，程序就关闭了。

最后一件事就是通过GLFW注册我们的函数至合适的回调，代码是这样的：

```
glfwSetKeyCallback(window, key_callback);
```

除了按键回调函数之外，我们还能我们自己的函数注册其它的回调。例如，我们可以注册一个回调函数来处理窗口尺寸变化、处理一些错误信息等。我们可以在创建窗口之后，开始游戏循环之前注册各种回调函数。

2.9.4 01-06 纹理

SOIL

SOIL是简易OpenGL图像库(Simple OpenGL Image Library)的缩写，它支持大多数流行的图像格式，使用起来也很简单，你可以从他们的[主页](#)下载。像其它库一样，你必须自己生成.lib。你可以使用/projects文件夹内的任意一个解决方案(Solution)文件（不用担心他们的Visual Studio版本太老，你可以把它们转变为新的版本，这一般是没问题的。译注：用VS2010的时候，你要用VC8而不是VC9的解决方案，想必更高版本的情况亦是如此）来生成你自己的.lib文件。你还要添加src文件夹里面的文件到你的includes文件夹；对了，不要忘记添加SOIL.lib到你的链接器选项，并在你代码文件的开头加上`#include <SOIL.h>`。

下面的教程中，我们会使用一张木箱的图片。要使用SOIL加载图片，我们需要使用它的`SOIL_load_image`函数：

```
int width, height;
unsigned char* image = SOIL_load_image("container.jpg", &width, &height, 0, SOIL_LOAD_RGB);
```

函数首先需要输入图片文件的路径。然后需要两个`int`指针作为第二个和第三个参数，SOIL会分别返回图片的宽度和高度到其中。后面我们在生成纹理的时候会用图像的宽度和高度。第四个参数指定图片的通道(Channel)数量，但是这里我们只需留为`0`。最后一个参数告诉SOIL如何来加载图片：我们只关注图片的`RGB`值。结果会储存为一个很大的`char`/byte数组。

生成纹理

和之前生成的OpenGL对象一样，纹理也是使用ID引用的。让我们来创建一个：

```
GLuint texture;
 glGenTextures(1, &texture);
```

`glGenTextures`函数首先需要输入生成纹理的数量，然后把它们储存在第二个参数的`GLuint`数组中（我们的例子中只是一个单独的`GLuint`），就像其他对象一样，我们需要绑定它，让之后任何的纹理指令都可以配置当前绑定的纹理：

```
 glBindTexture(GL_TEXTURE_2D, texture);
```

现在纹理已经绑定了，我们可以使用前面载入的图片数据生成一个纹理了。纹理可以通过`glTexImage2D`来生成：

```
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
 glGenerateMipmap(GL_TEXTURE_2D);
```

函数很长，参数也不少，所以我们一个一个地讲解：

- 第一个参数指定了纹理目标(Target)。设置为`GL_TEXTURE_2D`意味着会生成与当前绑定的纹理对象在同一个目标上的纹理（任何绑定到`GL_TEXTURE_1D`和`GL_TEXTURE_3D`的纹理不会受到影响）。
- 第二个参数为纹理指定多级渐远纹理的级别，如果你希望单独手动设置每个多级渐远纹理的级别的话。这里我们填0，也就是基本级别。
- 第三个参数告诉OpenGL我们希望把纹理储存为何种格式。我们的图像只有`RGB`值，因此我们也把纹理储存为`RGB`值。
- 第四个和第五个参数设置最终的纹理的宽度和高度。我们之前加载图像的时候储存了它们，所以我们使用对应的变量。
- 下个参数应该总是被设为`0`（历史遗留问题）。
- 第七第八个参数定义了源图的格式和数据类型。我们使用RGB值加载这个图像，并把它们储存为`char`(byte)数组，我们将会传入对应值。
- 最后一个参数是真正的图像数据。

当调用`glTexImage2D`时，当前绑定的纹理对象就会被附加上纹理图像。然而，目前只有基本级别(Base-level)的纹理图像被加载了，如果要使用多级渐远纹理，我们必须手动设置所有不同的图像（不断递增第二个参数）。或者，直接在生成纹理之后调用`glGenerateMipmap`。这会为当前绑定的纹理自动生成所有需要的多级渐远纹理。

生成了纹理和相应的多级渐远纹理后，释放图像的内存并解绑纹理对象是一个很好的习惯。

```
SOIL_free_image_data(image);
 glBindTexture(GL_TEXTURE_2D, 0);
```

生成一个纹理的过程应该看起来像这样：

```
GLuint texture;
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
 // 为当前绑定的纹理对象设置环绕、过滤方式
 ...
 // 加载并生成纹理
 int width, height;
 unsigned char* image = SOIL_load_image("container.jpg", &width, &height, 0, SOIL_LOAD_RGB);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
 glGenerateMipmap(GL_TEXTURE_2D);
 SOIL_free_image_data(image);
 glBindTexture(GL_TEXTURE_2D, 0);
```

纹理单元

你可能会奇怪为什么 `sampler2D` 变量是个uniform，我们却不用 `glUniform` 给它赋值。使用 `glUniform1i`，我们可以给纹理采样器分配一个位置值，这样的话我们能够在一个片段着色器中设置多个纹理。一个纹理的位置值通常称为一个 **纹理单元**(Texture Unit)。一个纹理的默认纹理单元是0，它是默认的激活纹理单元，所以教程前面部分我们没有分配一个位置值。

纹理单元的主要目的是让我们在着色器中可以使用多于一个的纹理。通过把纹理单元赋值给采样器，我们可以一次绑定多个纹理，只要我们首先激活对应的纹理单元。就像 `glBindTexture` 一样，我们可以使用 `glActiveTexture` 激活纹理单元，传入我们需要使用的纹理单元：

```
glActiveTexture(GL_TEXTURE0); // 在绑定纹理之前先激活纹理单元
 glBindTexture(GL_TEXTURE_2D, texture);
```

激活纹理单元之后，接下来的 `glBindTexture` 函数调用会绑定这个纹理到当前激活的纹理单元，纹理单元 `GL_TEXTURE0` 默认总是被激活，所以我们在前面的例子里当我们使用 `glBindTexture` 的时候，无需激活任何纹理单元。

OpenGL至少保证有16个纹理单元供你使用，也就是说你可以激活从`GL_TEXTURE0`到`GL_TEXTURE15`。它们都是按顺序定义的，所以我们也可以通过`GL_TEXTURE0 + 8`的方式获得`GL_TEXTURE8`，这在当我们需要循环一些纹理单元的时候会很有用。

我们仍然需要编辑片段着色器来接收另一个采样器。这应该相对来说非常直接了：

```
#version 330 core
...
uniform sampler2D ourTexture1;
uniform sampler2D ourTexture2;

void main()
{
    color = mix(texture(ourTexture1, TexCoord), texture(ourTexture2, TexCoord), 0.2);
}
```

最终输出颜色现在是两个纹理的结合。GLSL内建的 `mix` 函数需要接受两个值作为参数，并对它们根据第三个参数进行线性插值。。如果第三个值是 `0.0`，它会返回第一个输入；如果是 `1.0`，会返回第二个输入值。`0.2` 会返回 `80%` 的第一个输入颜色和 `20%` 的第二个输入颜色，即返回两个纹理的混合色。

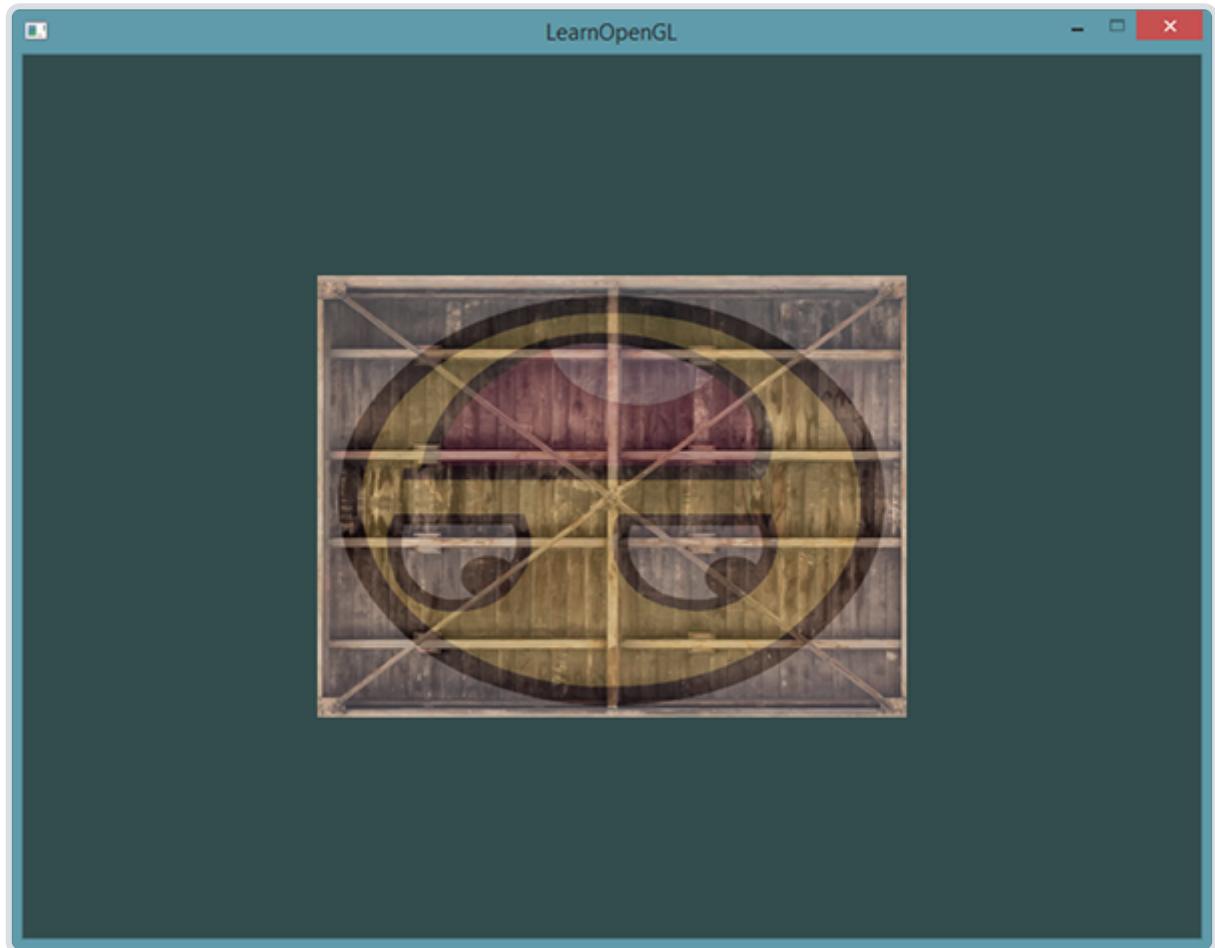
我们现在需要载入并创建另一个纹理；你应该对这些步骤很熟悉了。记得创建另一个纹理对象，载入图片，使用`glTexImage2D`生成最终纹理。对于第二个纹理我们使用一张[你学习OpenGL时的面部表情图片](#)。

为了使用第二个纹理（以及第一个），我们必须改变一点渲染流程，先绑定两个纹理到对应的纹理单元，然后定义哪个uniform采样器对应哪个纹理单元：

```
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texture1);
 glUniform1i(glGetUniformLocation(ourShader.Program, "ourTexture1"), 0);
 glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, texture2);
 glUniform1i(glGetUniformLocation(ourShader.Program, "ourTexture2"), 1);

 glBindVertexArray(VAO);
 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
 glBindVertexArray(0);
```

注意，我们使用`glUniform1i`设置uniform采样器的位置值，或者说纹理单元。通过`glUniform1i`的设置，我们保证每个uniform采样器对应着正确的纹理单元。你应该能得到下面的结果：



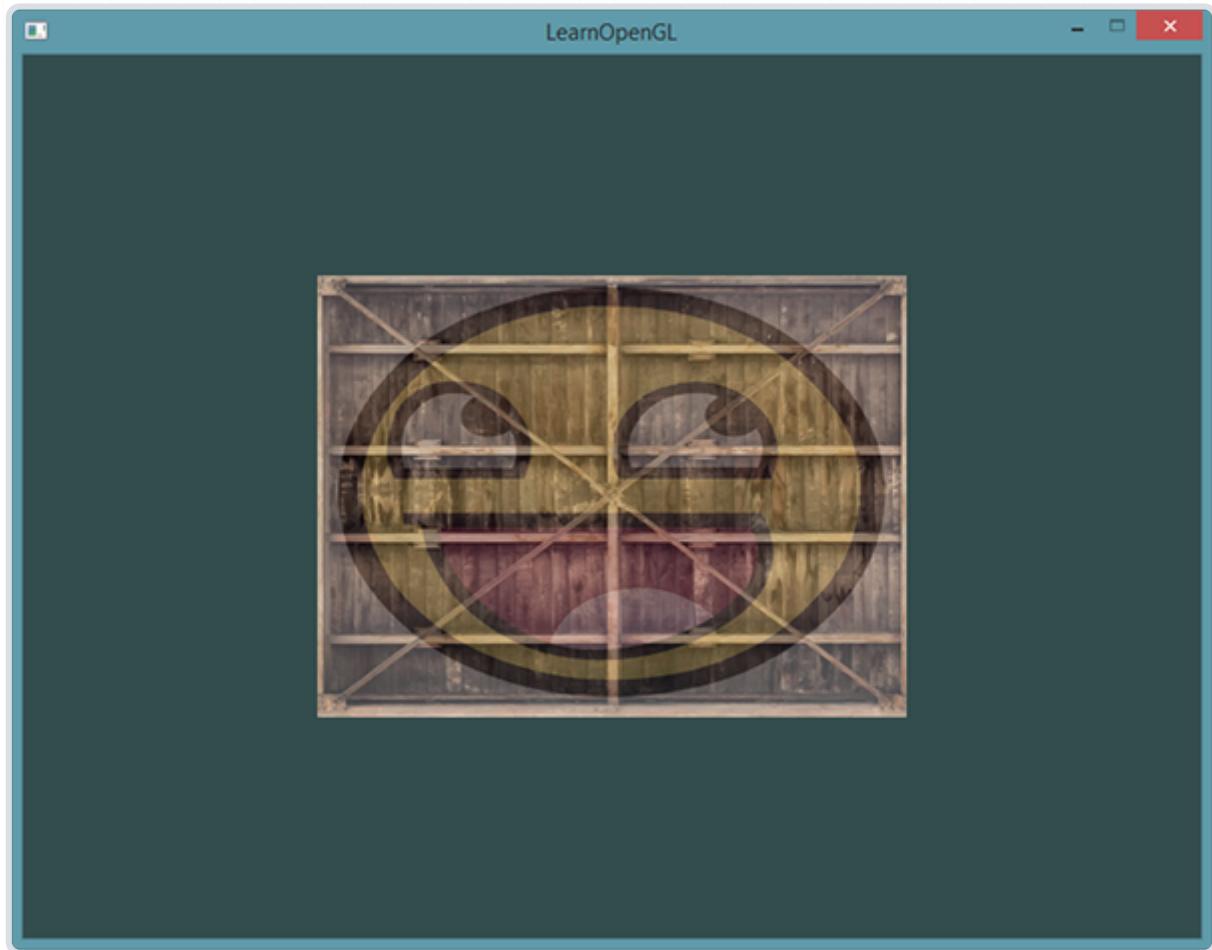
你可能注意到纹理上下颠倒了！这是因为OpenGL要求y轴`0.0`坐标是在图片的底部的，但是图片的y轴`0.0`坐标通常在顶部。一些图片加载器比如[DevIL](#)在加载的时候有选项重置y原点，但是SOIL没有。SOIL却有一个叫做`SOIL_load_OGL_texture`函数可以使用一个叫做`SOIL_FLAG_INVERT_Y`的标记加载并生成纹理，这可以解决我们的问题。不过这个函数用了一些在现代OpenGL中失效的特性，所以我们现在仍需坚持使用`SOIL_load_image`，自己做纹理的生成。

所以修复我们小问题，有两个选择：

1. 我们可以改变顶点数据的纹理坐标，翻转 `y` 值（用`1减去y坐标`）。
2. 我们可以编辑顶点着色器来自动翻转 `y` 坐标，替换 `TexCoord` 的值为 `TexCoord = vec2(texCoord.x, 1.0f - texCoord.y);`。

上面提供的解决方案仅仅通过一些黑科技让图片翻转。它们在大多数情况下都能正常工作，然而实际上这种方案的效果取决于你的实现和纹理，所以最好的解决方案是调整你的图片加载器，或者以一种`y`原点符合OpenGL需求的方式编辑你的纹理图像。

如果你编辑了顶点数据，在顶点着色器中翻转了纵坐标，你会得到下面的结果：



如果你看到了一个开心的箱子，你就做对了。你可以对比一下[源代码](#)，以及[顶点着色器](#)和[片段着色器](#)。

2.9.5 01-09 摄像机

自由移动

让摄像机绕着场景转的确很有趣，但是让我们自己移动摄像机会更有趣！首先我们必须设置一个摄像机系统，所以在我们的程序前面定义一些摄像机变量很有用：

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
```

`LookAt` 函数现在成了：

```
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

我们首先将摄像机位置设置为之前定义的`cameraPos`。方向是当前的位置加上我们刚刚定义的方向向量。这样能保证无论我们怎么移动，摄像机都会注视着目标方向。让我们摆弄一下这些向量，在按下某些按钮时更新`cameraPos`向量。

我们已经为GLFW的键盘输入定义了一个`key_callback`函数，我们来新添加几个需要检查的按键命令：

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    ...
    GLfloat cameraSpeed = 0.05f;
    if(key == GLFW_KEY_W)
        cameraPos += cameraSpeed * cameraFront;
    if(key == GLFW_KEY_S)
        cameraPos -= cameraSpeed * cameraFront;
    if(key == GLFW_KEY_A)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if(key == GLFW_KEY_D)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

当我们按下WASD键的任意一个，摄像机的位置都会相应更新。如果我们希望向前或向后移动，我们就把位置向量加上或减去方向向量。如果我们希望向左右移动，我们使用叉乘来创建一个右向量(Right Vector)，并沿着它相应移动就可以了。这样就创建了使用摄像机时熟悉的扫射(Strafe)效果。

注意，我们对右向量进行了标准化。如果我们没对这个向量进行标准化，最后的叉乘结果会根据`cameraFront`变量返回大小不同的向量。如果我们不对向量进行标准化，我们就得根据摄像机的朝向不同加速或减速移动了，但假如进行了标准化移动就是匀速的。

如果你用这段代码更新`key_callback`函数，你就可以在场景中自由的前后左右移动了。

在摆弄这个基础的摄像机系统之后你可能会注意到这个摄像机系统不能同时朝两个方向移动（对角线移动），而且当你按下一个按键时，它会先顿一下才开始移动。这是因为大多数事件输入系统一次只能处理一个键盘输入，它们的函数只有当我们激活了一个按键时才被调用。虽然这对大多数GUI系统都没什么问题，它对摄像机来说并不合理。我们可以用一些小技巧解决这个问题。

这个技巧是在回调函数中只跟踪哪个按键被按下/释放。在游戏循环中我们读取这些值，检查哪个按键是激活的，然后做出相应反应。我们只储存哪个按键被按下/释放的状态信息，并在游戏循环中对状态做出反应。首先，我们来创建一个boolean数组代表被按下/释放的按键：

```
bool keys[1024];
```

然后我们需要在`key_callback`函数中设置被按下/释放的按键为`true`或`false`：

```
if(action == GLFW_PRESS)
    keys[key] = true;
else if(action == GLFW_RELEASE)
    keys[key] = false;
```

并且我们创建一个新的叫做`do_movement`的函数，在这个函数中，我们将根据正在被按下的按键更新摄像机的值。：

```
void do_movement()
{
    // 摄像机控制
    GLfloat cameraSpeed = 0.01f;
    if(keys[GLFW_KEY_W])
        cameraPos += cameraSpeed * cameraFront;
    if(keys[GLFW_KEY_S])
        cameraPos -= cameraSpeed * cameraFront;
    if(keys[GLFW_KEY_A])
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if(keys[GLFW_KEY_D])
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

之前的代码现在被移动到`do_movement`函数中。由于所有GLFW的按键枚举值本质上都是整数，我们可以把它们当数组索引使用。

最后，我们需要在游戏循环中添加新函数的调用：

```
while(!glfwWindowShouldClose(window))
{
    // 检测并调用事件
    glfwPollEvents();
    do_movement();

    // 渲染
    ...
}
```

至此，你应该可以同时向多个方向移动了，并且当你按下按钮时也会立刻移动了。如遇困难，可以查看下[源代码](#)。

3. 代码仓库

你可以在每一篇教程中找到在线的代码范例，但如果你想自己运行教程的Demo或者将正常工作的范例代码与你的代码进行比较，你可以在[这里](#)找到在线的GitHub代码仓库。

目前，`CMakeLists.txt` 文件能够正常生成Visual Studio的工程文件和make文件，它能够在Windows和Linux上运行。但是它在Apple的macOS和其他的IDE上还没有进行非常完全的测试，所以如果出现问题你可以留言或者帮忙通过Pull Request来更新一下`CMakeLists.txt` 文件让它支持不同的系统。

我非常感谢Zwookie在制作Linux的CMake脚本时提供的巨大帮助。感谢Zwookie对CMakeLists的更新，现在它能够在Windows和Linux上生成工程文件了。

你也可以查看Polytonic的[Glitter](#)，它是一个非常简单的样板工程，它提供了已经预配置好的相关依赖项。