

OpenGL (Open Graphics Library)

原文	OpenGL
作者	JoeyDeVries
翻译	gjy_1992
校对	Geequlim

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应用程序编程接口(Application Programming Interface, API)，它包含了一系列可以操作图形、图像的方法。然而，OpenGL 本身并不是一个 API，仅仅是一个规范，由 [Khronos 组织](#)制定并维护。

OpenGL 严格规定了每个函数该如何执行，以及它们该如何返回。至于内部具体每个函数是如何实现的，将由 OpenGL 库的开发者自行决定(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。每个你购买的显卡都会支持特定版本的 OpenGL，通常是一个系列的显卡专门开发的。当你使用苹果系统的时候，OpenGL 库是由苹果自身维护的。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。(快甩锅)

Important

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 规范书都被寄存在 Khronos 那里，并且都是公开的。有兴趣的读者可以找到 OpenGL3.3(我们将要使用的版本)的[规范书](#)。如果你想深入到 OpenGL 的细节(只关心函数的功能描述而不是函数的实现)，这是个很好的选择。该规范还提供一个强大的可以寻找到每个函数具体功能的参考。

核心模式(Core-profile)与立即渲染模式(Immediate mode)

早期的 OpenGL 使用立即渲染模式(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,开发者很少能控制 OpenGL 如何进行计算。而开发者希望更多的灵活性。随着时间推移,规范越来越灵活,开发者也能更多的控制绘图细节。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,规范书开始废弃立即渲染模式,推出核心模式,这个模式完全移除了旧的特性。

当使用核心模式时,OpenGL 迫使我们使用现代的做法。当我们试图使用一个废弃的函数时,OpenGL 会抛出一个错误并终止绘图。现代做法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 实际操作中抽象掉了很多细节,因而它易于学习的同时,也很难去把握 OpenGL 具体是如何操作的。现代做法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,但是值得去努力。

现今更高版本的 OpenGL 已经发布(目前最新是 4.5),你可能会问:为什么我们还要学习 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,添加了额外的功能,并不更改进核心架构。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。

Attention

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,并有选择的启用新特性。

在有些教程里你会发现像如下方式注明的新特性。

扩展(Extension)

OpenGL 的一大特性就是对扩展的支持,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,就可以方便

的检查显卡是否支持此扩展。通常，当一个扩展非常流行或有用的时候，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)
{
    // 使用一些新的特性
}
else
{
    // 不支持此扩展：用旧的方式去做
}
```

使用 OpenGL 3.3 时，我们很少需要使用扩展来完成大多数功能，但是掌握这种方式是必须的。

状态机(State Machine)

OpenGL 自身是一个巨大的状态机：一个描述 OpenGL 该如何操作的所有变量的大集合。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：设置一些选项，操作一些缓冲。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。

用 OpenGL 工作时，我们会遇到一些状态设置函数(State-changing Function)，以及一些在这些状态的基础上状态应用的函数(State-using Function)。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(Object)

OpenGL 库是用 C 语言写的，同时也支持多种语言的派生，但是核心是一个 C 库。一些 C 语言的结构不易被翻译到其他高层语言，因此 OpenGL 设计的时候引入了一些抽象概念。“对象”就是其中一个。

在 OpenGL 中一个对象是指一些选项的集合，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，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 object_name {  
    GLfloat option1;  
    GLuint option2;  
    GLchar[] name;  
};
```

Important

原始类型(**Primitive Type**) 使用 OpenGL 时，建议使用 OpenGL 定义的原始类型。比如使用 `float` 时我们加上前缀 `GL`(因此写作 `GLfloat`)。`int`, `uint`, `char`, `bool` 等等类似。OpenGL 定义的这些 `GL` 原始类型是平台无关的内存排列方式。而 `int` 等原始类型在不同平台上可能有不同的内存排列方式。使用 `GL` 原始类型可以保证你的程序在不同的平台上工作一致。

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

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

```
...
};

// 创建对象

GLuint objectId = 0;

glGenObject(1, &objectId);

// 绑定对象至上下文

glBindObject(GL_WINDOW_TARGET, objectId);

// 设置 GL_WINDOW_TARGET 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);

// 将上下文的 GL_WINDOW_TARGET 对象设回默认

glBindObject(GL_WINDOW_TARGET, 0);
```

这一小片代码将会是以后使用 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(例子里窗口对象的目标位置被定义成 `GL_WINDOW_TARGET`)。接下来我们设置窗口的选项。最后我们通过将目标位置的对象 id 设回 0 的方式解绑这个对象。设置的选项被保存在 `objectId` 代表的对象中，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。

Attention

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我们在程序中不止可以定义一个对象并且设置他们的状态，在我们需要进行一个操作的时候，只需要绑定预设了需要设置的对象即可。比如，有一个作为 3D 模型的数据(一栋房子或一个人物，由多个子模型构成)容器对象，在我们想绘制其中任何一个 3D 模型的时候，只需绑定相应的子模型数据的对象(我们预先创建并设置好了它们的选项)就可以了。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，简单的将相应的对象绑定上去，便不需要再进行重复的设置选项的操作了。

让我们开始吧

你现在已经知道一些 OpenGL 的相关知识了，包括 OpenGL 作为规范和库，OpenGL 大致的操作流程，以及一些使用扩展的小技巧。不要担心你还没有完全消化它们，通过后面的教程我们会仔细地讲解每一步，你会通过足够的例子来把握 OpenGL。如果你已经做好了开始下一步的准备，我们可以开始建立 OpenGL 上下文以及我们的第一个窗口了。[点击这里](#)(第一章第二节)

额外的资源

- [opengl.org](#): OpenGL 官方网站。
- [OpenGL registry](#): OpenGL 各版本的规范和扩展的主站。

创建窗口

原文	Creating a window
作者	JoeyDeVries
翻译	gjy_1992
校对	Geequlim

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 有目的的抽象(Abstract)这些操作。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平台相关代码的时间，提供给我们一个窗口和上下文用来渲染。最流行的几个库有 GLUT，SDL，SFML 和 GLFW。在教程里我们将使用 **GLFW**。

GLFW

GLFW 是一个专门针对 OpenGL 的 C 语言库，它提供了一些渲染物件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。

这一节和下一节的内容是建立 **GLFW** 环境，并保证它恰当地创建窗口和 OpenGL 上下文。本教程会一步步从获取，编译，链接 **GLFW** 库讲起。我们使用 Microsoft Visual Studio 2012 IDE，如果你用的不是它(或者只是 Visual Studio 的旧版本)请不要担心，大多数 IDE 上的操作都是类似的。Visual Studio 2012(或其他版本)可以从微软网站上免费下载(选择 Express 版本或 Community 版本)。

构建 **GLFW**

GLFW 可以从它们网站的[下载页](#)上获取。**GLFW** 已经有针对 Visual Studio 2012/2013 的预编译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，所以需要下载源代码包。

Attention

当你下载二进制版本时，请下载 32 位的版本而不是 64 位的除非你清楚你在做什么。大部分读者报告 64 位版本会出现很多奇怪的问题。

一旦下载完了源码包，解压到某处。我们只关心里面的这些内容：

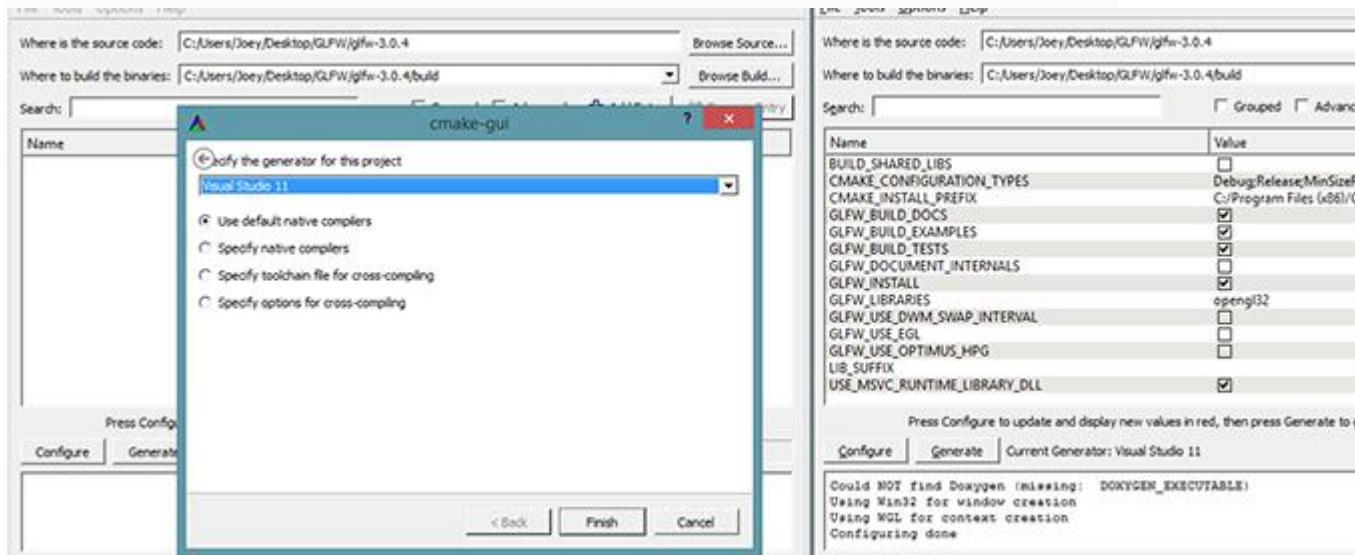
- 编译生成的库
- **include** 文件夹

从源代码编译库可以保证生成的目标代码是针对你的操作系统和 CPU 的，而一个预编译的二进制代码并不保证总是适合。提供源代码的一个问题是每个人用相同的 IDE 来编译，因而提供的工程文件可能和一些人的 IDE 不兼容。所以人们只能从.cpp 和.h 文件来自己建立工程，这是一项笨重的工作。因此诞生了一个叫做 **CMake** 的工具。

CMake

CMake 是一个工程文件生成工具，可以使用预定义好的 **CMake** 脚本，根据用户的选择生成不同 IDE 的工程文件。这允许我们从 **GLFW** 源码里创建一个 Visual Studio 2012 工程文件。首先，我们需要从[这里](#)下载安装 **CMake**。我们选择 Win32 安装程序。

一旦 **CMake** 安装成功，你可以选择从命令行或者 GUI 启动 **CMake**，为了简易我们选择后者。**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_ 文件夹来作为目标目录。



之后，点击 **Configure(设置)** 按钮，我们选择生成的目标平台为 **Visual Studio 11**(因为 Visual Studio 2012 的内部版本号是 11.0)。CMake 会显示可选的编译选项，这里我们使用默认设置，再次点击 **Configure(设置)** 按钮，保存这些设置。保存之后，我们可以点击 **Generate(生成)** 按钮，生成的工程文件就会出现在你的 *build* 文件夹中。

编译

在 **build** 文件夹里可以找到 **GLFW.sln** 文件，用 Visual Studio 2012 打开。因为 CMake 已经配置好了项目所以我们直接点击 **Build Solution(构建解决方案)** 然后编译的结果 **glfw3.lib** 就会出现在 **src/Debug** 文件夹内。(注意我们现在使用的 **glfw** 的版本号为 3.1)

生成库之后，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，之后添加 **GLFW** 的 **include** 目录到 **/include** 里去，相似的将 **glfw3.lib** 添加到 **/lib** 里去。这不是推荐的方式，因为很难去追踪 **library/include** 文件夹，而且重新安装 IDE/Compiler 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/Compiler 中指定这些文件夹。我个人使用一个单独的文件夹包含 **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路径(并且应该在你的多台电脑间共享)，然而要求是每次新建一个工程都需要告诉 IDE/编译器在哪能找到这些文件

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！

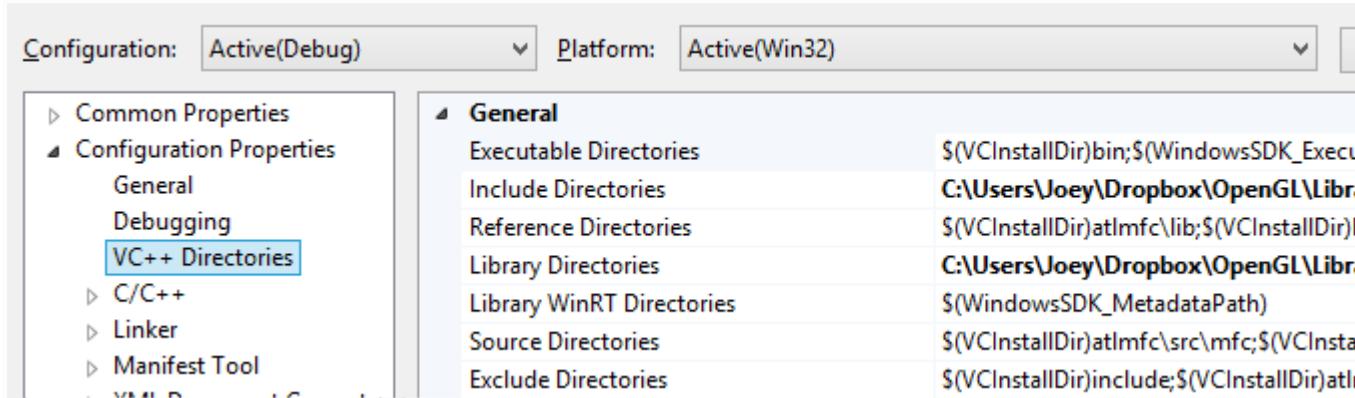
我们的第一个工程

现在，让我们打开 Visual Studio，创建一个新的工程。如果提供了多个选项，选择 Visual C++，然后选择空工程(**Empty Project**)，别忘了给你的工程起一个合适的名字。现在我们有了一个空的工程去创建我们的 OpenGL 程序。

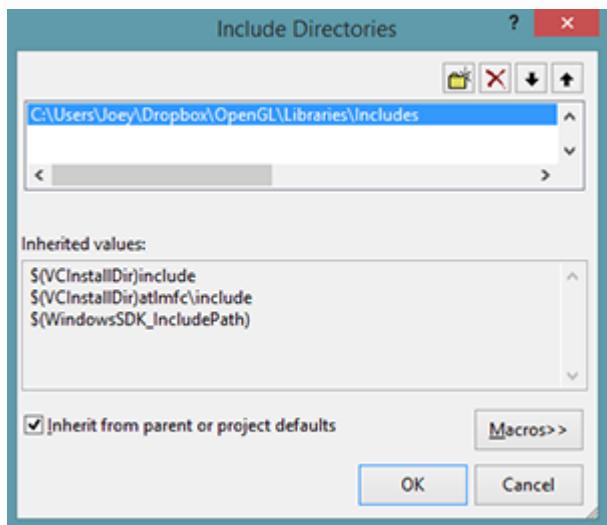
链接(Linking)

为了使我们的程序使用 GLFW，我们需要把 GLFW 库链接(**Link**)进工程。于是我们需要在链接器的设置里写上 **glfw3.lib**。但是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。

为了添加这些目录，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** 选项卡(如下图)。在下面的两栏添加目录：

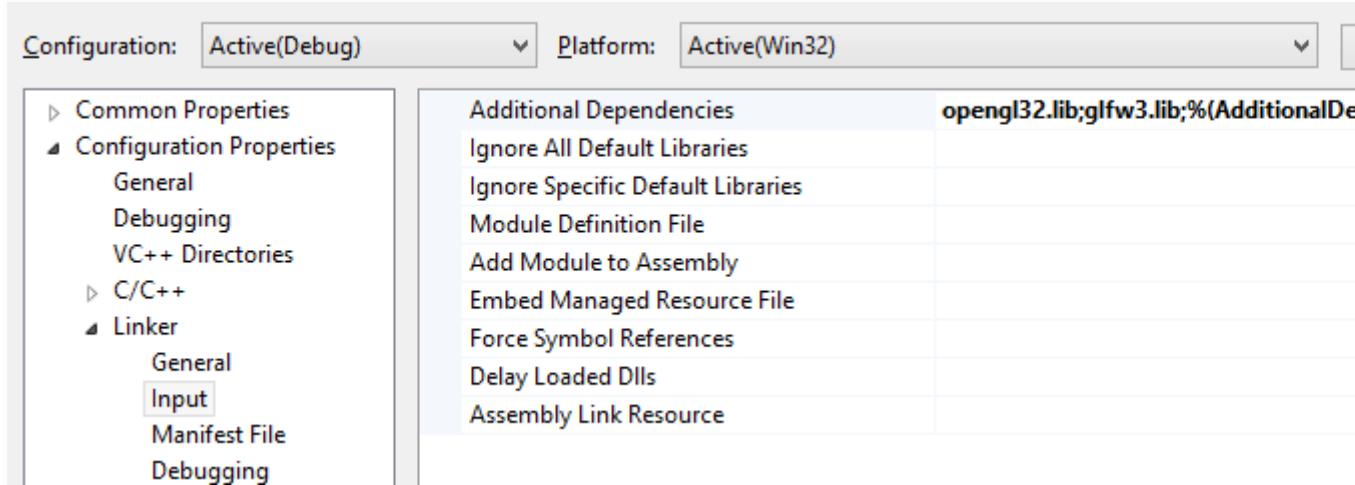


从这里你可以把自己的目录加进去从而工程知道从哪去寻找库文件和头文件。可以手动把目录加在后面，也可以点选项，下面的图是 **Include Directories** 的设置：



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，你就可以使用来引用头文件。库文件也是一样的。

现在 VS 可以找到我们链接 GLFW 需要的所有文件了。最后需要在 **Linker(链接器)** 选项卡里的 **Input** 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。因为我们的库名字是 **glfw3.lib**，我们把它加到 **Additional Dependencies** 域里面(手动或者使用选项)。这样 GLFW 就会被链接进我们的工程。除了 GLFW，你也需要链接 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** 已经随着 Microsoft SDK 装进了 Visual Studio 的默认目录, 所以 Windows 上我们只需将 **opengl32.lib** 添加进 Additional Dependencies。

Linux 上的 OpenGL 库

在 Linux 下你需要链接 **libGL.so**, 所以要添加**-IGL** 到你的链接器设置里。如果找不到这个库你可能需要安装 Mesa, NVidia 或 AMD 的开发包, 这部分因平台而异就不仔细讲解了。

现在, 如果你添加好了 GLFW 和 OpenGL 库, 你可以用如下方式添加 GLFW 头文件:

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

这个头文件包含了 GLFW 的设置。

GLEW

到这里, 我们仍然有一件事要做。因为 OpenGL 只是一个规范, 具体的实现是由驱动开发商针对特定显卡实现的。由于显卡驱动版本众多, 大多数函数都无法在编译时确定下来, 需要在运行时获取。开发者需要运行时获取函数地址并保存下来供以后使用。取得地址的方法因平台而异, Windows 下看起来类似这样:

```
// 定义函数类型
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);
```

你可以看到代码复杂而笨重，因为我们对于每个函数都必须这样。幸运的是，有一个针对此目的的库，GLEW，是目前最流行的做这件事的方式。

编译和链接 GLEW

GLEW 是 OpenGL Extension Wrangler Library 的缩写，它管理我们上面提到的一系列繁琐的任务。因为 GLEW 也是一个库，我们同样需要链接进工程。GLEW 可以从[这里](#)下载，你可以选择下载二进制版本或者下载源码编译。记住，优先选择 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')，用如上的方式添加其库文件和头文件，最后在链接器的选项里加上 `glew32s.lib`。注意 GLFW3 也是编译成了一个静态库。

Important

静态(Static)链接是指编译时就将库代码里的内容合并进二进制文件。优点就是你不需要再放额外的文件，只需要发布你最终的二进制代码文件。缺点就是你的程序会变得更大，另外当库有升级版本时，你必须重新进行编译。**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。优点是使你的程序大小变小并且更容易升级，缺点是你发布时必须带上这些`dll`。

如果你希望静态链接 GLEW，必须在包含 GLEW 头文件之前定义预编译宏

`GLEW_STATIC`:

```
#define GLEW_STATIC  
  
#include <GL/glew.h>
```

如果你希望动态链接，那么就不要定义这个宏。但是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。

Important

对于 Linux 用户建议使用这个命令行 `-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，我们将进入[下一节](#)去使用 GLFW 和 GLEW 来设置 OpenGL 上下文并创建窗口。记住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，请参考额外资源中的例子。

额外的资源

- [Building applications](#): 提供了很多编译链接相关的信息以及一大批错误的解决方法。
- [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 写的一个自动工具的教程，关于如何在 Linux 上建立编译环境，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，这是一个很好的东西。

你好，窗口

原文	Hello Window
作者	JoeyDeVries
翻译	Geequlim
校对	Geequlim

上一节中我们获取并编译了 GLFW 和 GLEW 这两个开源库，现在我们就可以使用它们来创建一个 OpenGL 绘图窗口了。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。注意，之所以定义 `GLEW_STATIC` 宏，是因为我们使用 GLEW 的静态链接库。

```
// GLEW  
  
#define GLEW_STATIC  
  
#include <GL/glew.h>
```

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

Attention

请确认在包含 `GLFW` 的头文件之前包含了 `GLEW` 的头文件。在包含 `glew.h` 头文件时会引入许多 `OpenGL` 必要的头文件(例如 `GL/gl.h`)，所以`#include` 应放在引入其他头文件的代码之前。

接下来我们创建 `main` 函数，并做一些初始化 `GLFW` 的操作：

```
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_RESIZABLE, GL_FALSE);  
  
    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` 函数的第一个参数表示我们要进行什么样的配置，我们可以选择大量以 `GLFW_` 开头的枚举值；第二个参数接受一个整形，用来设置这个配置的值。该函数的所有的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` 文件会得到大量的连接错误，这是因为你还需要进一步设置 `GLFW`。

由于本站的教程都是基于 `OpenGL3.3` 以后的版本展开讨论的，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。这也确保用户在没有适当的 `OpenGL` 版本支持的情况下无法运行。在这里我们告诉 `GLFW` 想要的 `OpenGL` 版本号是 `3.3`，并且不允许用户调整窗口的大小。我们明确地告诉 `GLFW` 我们想要使用核心模式。

(Core-profile)，这将导致我们无法使用那些已经废弃的 API，而这不正是一个很好的提醒吗？当我们不小心用了旧功能时报错，就能避免使用一些被废弃的用法了。如果你使用的是 Mac OSX 系统你还需要加下面这行代码这些配置才能起作用：

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

Important

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。可以通过运行 `glew` 附带的 `glxinfo` 程序或者其他工具(例如 [OpenGL Extension Viewer](#))来查看你的 OpenGL 版本。如果你的 OpenGL 版本低于 3.3 请更新你的驱动程序或者有必要的话更新设备。

接下来我们创建一个窗口对象，这个窗口对象中具有和窗口相关的许多数据，而且会被 GLFW 的其他函数频繁地用到。

```
GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpenGL",
                                      nullptr, nullptr);
if (window == nullptr)
{
    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；第三个参数表示只这个窗口的名称(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，先置为空指针就行。它的返回值 `GLFWwindow` 对象的指针会在其他的 GLFW 操作中使用到。创建完窗口我们就可以通知 GLFW 给我们的窗口在当前的线程中创建我们等待已久的 OpenGL 上下文了。

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(Core-profile)时出现一些问题。

视口(Viewport)

在我们绘制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 才只能知道要显示数据的窗口坐标。我们可以通过调用 `glViewport` 函数来设置这些维度：

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

前两个参数设置窗口左下角的位置。第三个和第四个参数设置渲染窗口的宽度和高度，我们设置成与 GLFW 的窗口的宽高大小，我们也可以将这个值设置成比窗口小的数值，然后所有的 OpenGL 渲染将会显示在一个较小的区域。

Important

OpenGL 使用 `glViewport` 定义的位置和宽高进行位置坐标的转换，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)有可能被转换为屏幕中的坐标(200,450)。注意，OpenGL 只会把-1 到 1 之间的坐标转换为屏幕坐标，因此在此例中(-1, 1)转换为屏幕坐标是(0,600)。

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关闭窗口并立即退出。我们希望程序在我们明确地关闭它之前一直保持运行状态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游戏循环(Game Loop)，这样我们的程序就能在我们让 GLFW 退出前保持运行了。下面几行的代码就实现了一个简单的游戏循环：

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

- `glfwWindowShouldClose` 函数在我们每次循环的开始前检查一次 GLFW 是否准备好要退出，如果是这样的话该函数返回 `true` 然后游戏循环便结束了，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(比如键盘有按钮按下、鼠标移动等)然后调用对应的回调函数(我们可以手动设置这些回调函数)。我们一般在游戏循环的一开始就检查事件。
- 调用 `glfwSwapBuffers` 会交换缓冲区(储存着 GLFW 窗口每一个像素颜色的缓冲区)

Important

双缓冲区(Double buffer)

应用程序使用单缓冲区绘图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，而是通过一步一步地计算结果绘制的，这可能会花费一些时间。为了规避这些问题，我们应用双缓冲区渲染窗口应用程序。前面的缓冲区保存着计算后可显示给用户的图像，被显示到屏幕上；所有的渲染命令被传递到后台的缓冲区进行计算。当所有的渲染命令执行结束后，我们交换前台缓冲和后台缓冲，这样图像就立即呈显出来，之后清空缓冲区。

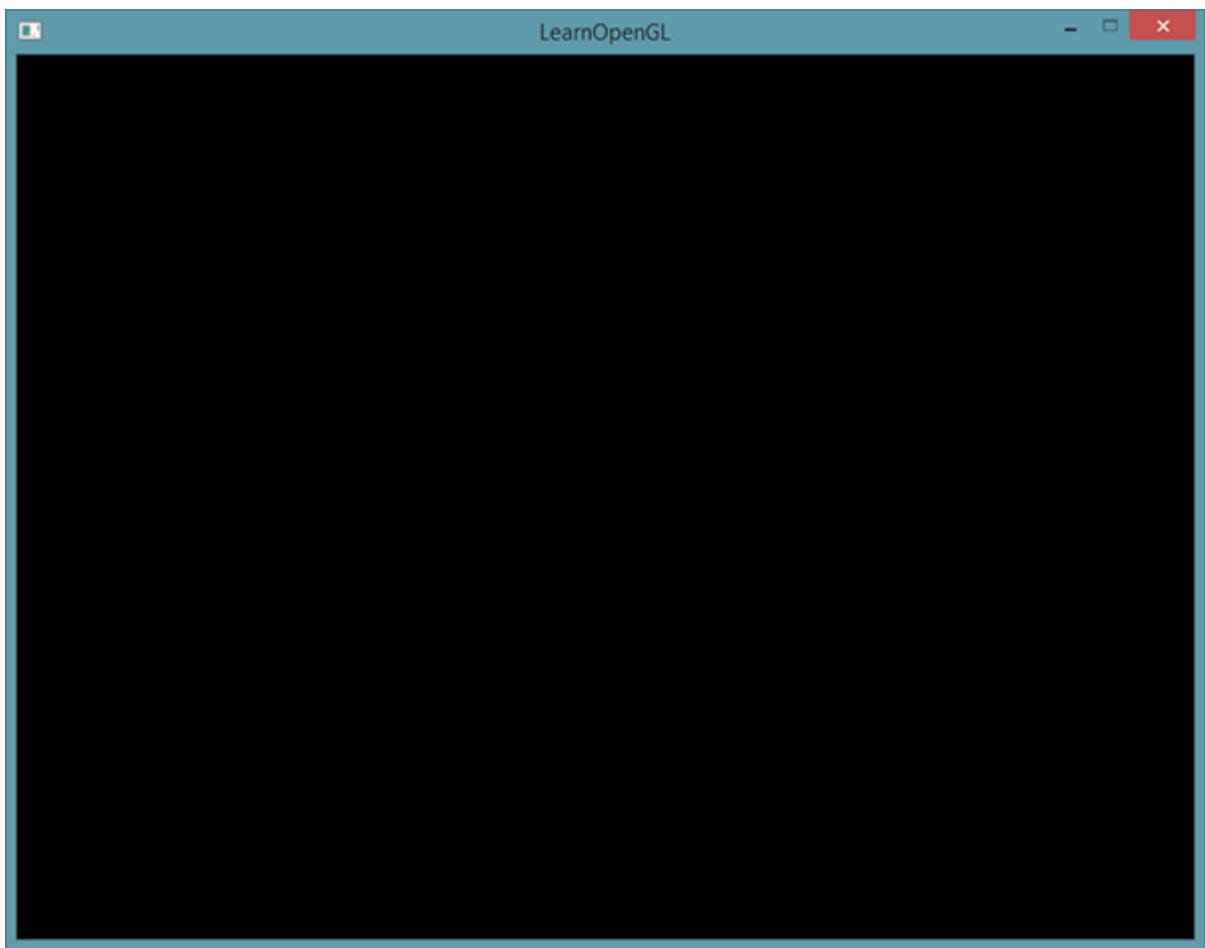
最后一件事

当游戏循环结束后我们需要释放之前的操作分配的资源，我们可以在 `main` 函数的最后调用 `glfwTerminate` 函数来释放 GLFW 分配的内存。

```
glfwTerminate();
```

```
return 0;
```

这样便能清空 GLFW 分配的内存然后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，你将会看到如下的一个黑色窗口：



如果你没有编译通过或者有什么问题的话，首先请检查你程序的链接选项是否正确。然后对比本教程的代码，检查你的代码是不是哪里写错了，你也可以[点击这里](#)获取我的完整代码。

输入

我们同样也希望能够在 GLFW 中实现一些键盘控制，这是通过设置 GLFW 的回调函数(**Callback Function**)来实现的。回调函数事实上是一个函数指针，当我们为 GLFW 设置回调函数后，GLFW 会在恰当的时候调用它。**按键回调**(**KeyCallback**)是众多回调函数中的一种，当我们为 GLFW 设置按键回调之后，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事件的按键；第三个整形参数描述用户是否有第二个键按下或释放；第四个整形参数表示事件类型，如按下或释放；最后一个参数是表示是否有 `Ctrl`、`Shift`、`Alt`、`Super` 等按钮的操作。GLFW 会在恰当的时候调用它，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);
```

```
}
```

在这个 `key_callback` 函数中，它检测键盘是否按下了 `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失败并且程序关闭。

最后一件事就是通过 **GLFW** 使用适合的回调来注册我们的函数，代码是这样的：

```
glfwSetKeyCallback(window, key_callback);
```

除了按键回调函数之外，我们还能为 **GLFW** 注册其他的回调函数。例如，我们可以注册一个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到开始游戏循环之前注册各种回调函数。

渲染(Rendering)

我们要把所有的渲染操作放到游戏循环中，因为我们想让这些渲染操作在每次游戏循环迭代的时候都能被执行。我们将做如下的操作：

```
// 程序循环  
  
while(!glfwWindowShouldClose(window))  
  
{  
    // 检查事件  
    glfwPollEvents();  
  
    // 在这里执行各种渲染操作  
    ...  
  
    // 交换缓冲区  
    glfwSwapBuffers(window);  
}
```

为了测试一切都正常，我们想让屏幕清空为一种我们选择的颜色。在每次执行新的渲染之前我们都希望清除上一次循环的渲染结果，除非我们想要看到上一次的结果。我们可以通过调用 **glClear** 函数来清空屏幕缓冲区的颜色，他接受一个整形常量参数来指定要清空的缓冲区，这个常量可以是 **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区。

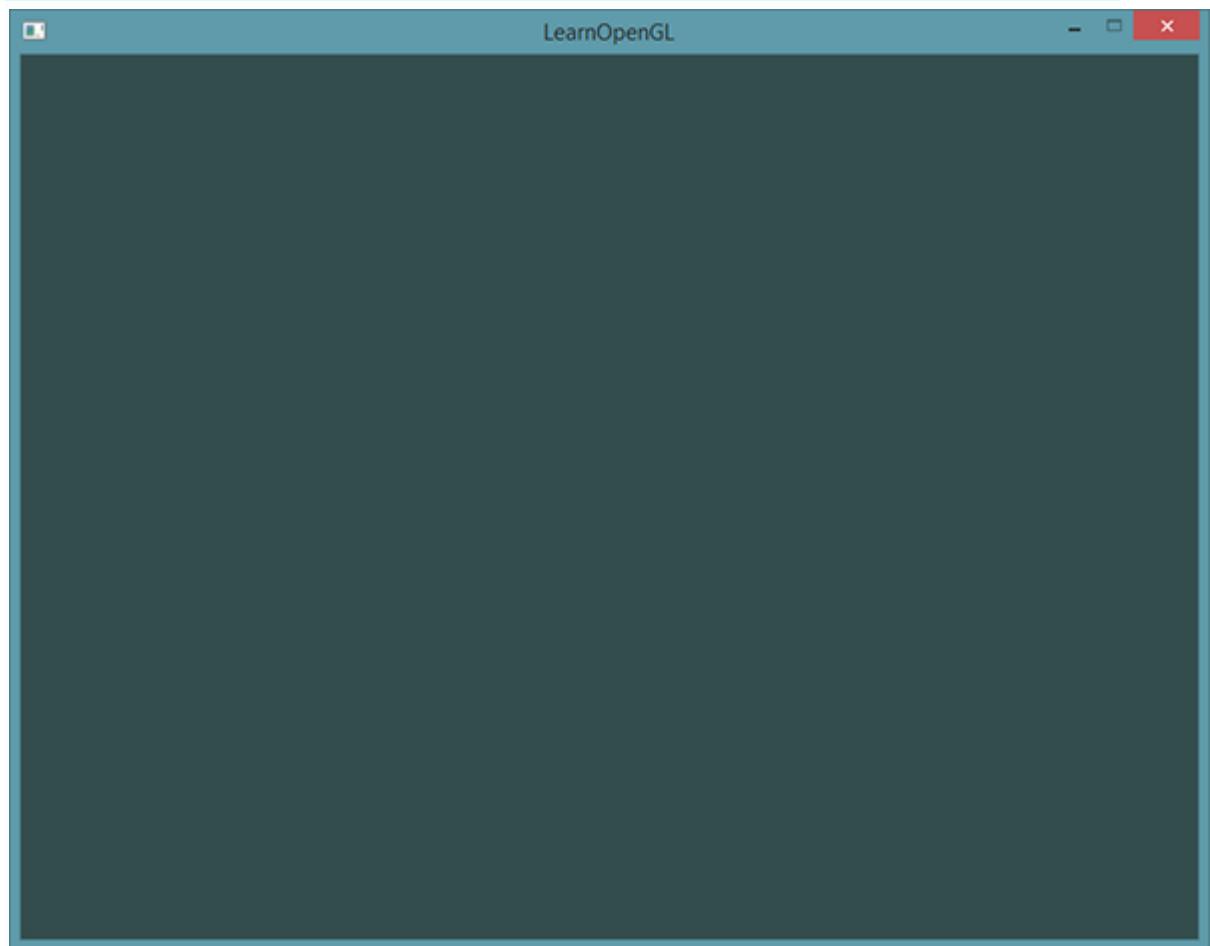
```
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
```

```
glClear(GL_COLOR_BUFFER_BIT);
```

注意，除了 `glClear` 之外，我们还要调用 `glClearColor` 来设置要清空缓冲的颜色。当调用 `glClear` 函数之后，整个指定清空的缓冲区都被填充为 `glClearColor` 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。

Important

你应该能够想起来我们在 [OpenGL](#) 教程的内容，`glClearColor` 函数是一个状态设置函数，而 `glClear` 函数则是一个状态应用的函数。



此程序的完整源代码可以在[这里](#)找到。

好了，到目前为止我们已经做好开始在游戏循环中添加许多渲染操作的准备了，我认为我们的闲扯已经够多了，从下一篇教程开始我们将真正的征程。

你好，三角形

原文	Creating a window
作者	JoeyDeVries
翻译	Django
校对	Geequilibrium

图形渲染管线(Pipeline)

在 OpenGL 中任何事物都在 3D 空间中，但是屏幕和窗口是一个 2D 像素阵列，所以 OpenGL 的大部分工作都是关于如何把 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(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：第一个部分把你的 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，以及如何使用它创建一些像素，这对我们来说是有好处的。

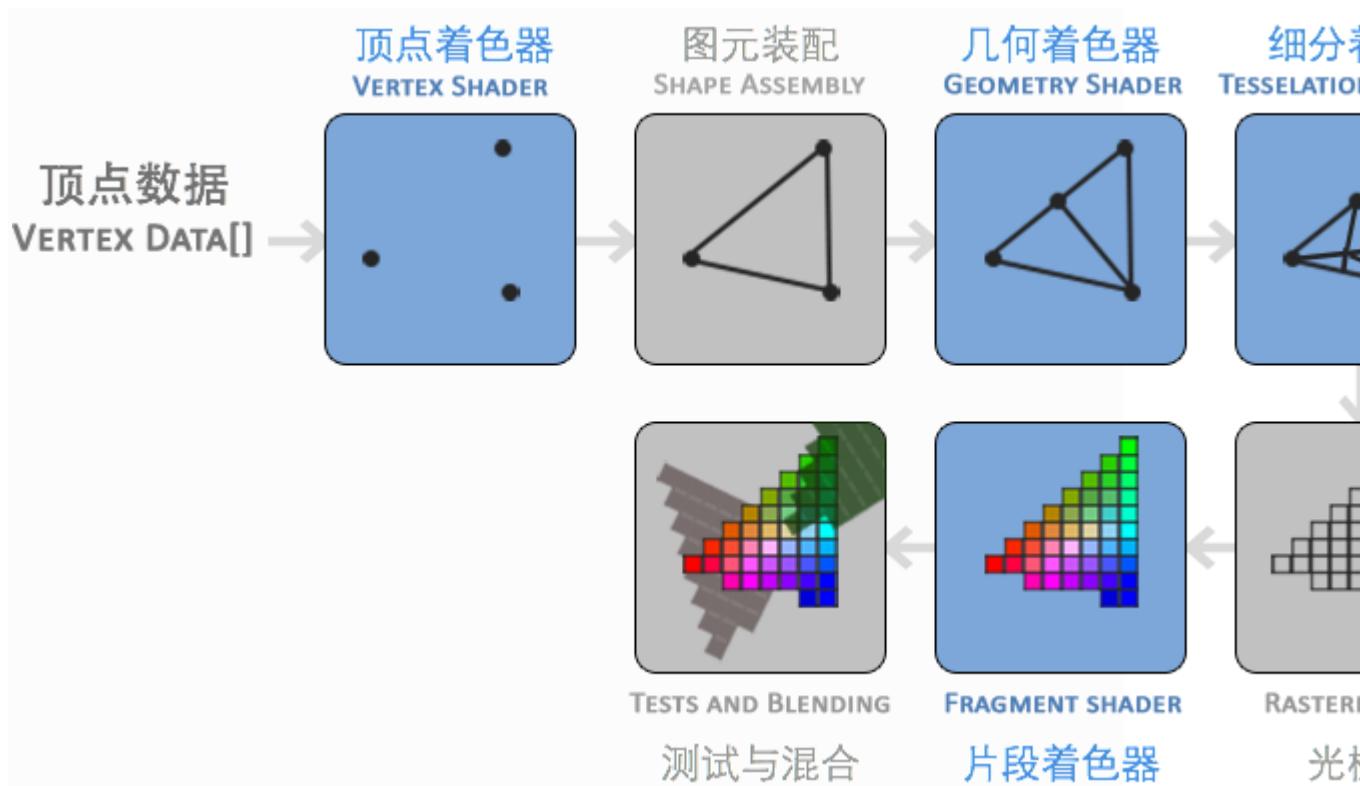
Important

2D 坐标和像素也是不同的，2D 坐标是在 2D 空间中的一个点的非常精确的表达，2D 像素是这个点的近似值，它受到你的屏幕/窗口解析度的限制。

图形渲染管线接收一组 3D 坐标，然后把它们转变为你的屏幕上的有色 2D 像素。图形渲染管线可以被划分为几个阶段，每个阶段需要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(它们有一个特定的函数)，它们能简单地并行执行。由于它们的并行执行特性，当今大多数显卡都有成千上万的小处理核心，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，我们可以用自己写的着色器替换默认的。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，因为它们运行在 GPU 上，所以它们会节约宝贵的 CPU 时间。OpenGL 着色器是用 OpenGL 着色器语言(OpenGL Shading Language, GLSL)写成的，我们在下一节会花更多时间研究它。

在下面，你会看到一个图形渲染管线的每个阶段的抽象表达。要注意蓝色部分代表的是我们可以自定义的着色器。



如你所见，图形渲染管线包含很多部分，每个都是将你的顶点数据转变为最后渲染出来的像素这个大过程中的一个特定阶段。我们会概括性地解释渲染管线的每个部分，从而使你对图形渲染管线的工作方式有个大概了解。

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，它用来表示一个三角形，这个数组叫做顶点数据(Vertex Data)；这里顶点数据是一些顶点的集合。一个顶点是一个 3D 坐标的集合(也就是 x、y、z 数据)。而顶点数据是用顶点属性(Vertex Attributes)表示的，它可以包含任何我们希望用的数据，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](#))和几个颜色值组成的吧。

Important

为了让 OpenGL 知道我们的坐标和颜色值构成的到底是什么，OpenGL 需要你去提示你希望这些数据所表示的是什么类型。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基本图形(Primitives)，任何一个绘制命令的调用都必须把基本图形类型传递给 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。

基本图形装配(Primitive Assembly)阶段把顶点着色器的表示为基本图形的所有顶点作为输入(如果选择的是 `GL_POINTS`, 那么就是一个单独顶点), 把所有点组装为特定的基本图形的形状; 本节例子是一个三角形。

基本图形装配阶段的输出会传递给**几何着色器(Geometry Shader)**。几何着色器把基本图形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(或是其他的)基本图形来生成其他形状。例子中, 它生成了另一个三角形。

细分着色器(Tessellation Shaders)拥有把给定基本图形细分为更多小基本图形的能力。这样我们就能在物体更接近玩家的时候通过创建更多的三角形的方式创建出更加平滑的视觉效果。

细分着色器的输出会进入**光栅化(Rasterization** 也译为像素化)阶段, 这里它会把基本图形映射为屏幕上相应的像素, 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那些像素, 来提升执行效率。

Important

`OpenGL` 中的一个 **fragment** 是 `OpenGL` 渲染一个独立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 `OpenGL` 高级效果产生的地方。通常, 片段着色器包含用来计算像素最终颜色的 3D 场景的一些数据(比如光照、阴影、光的颜色等等)。

在所有相应颜色值确定以后, 最终它会传到另一个阶段, 我们叫做 **alpha 测试** 和**混合(Blending)**阶段。这个阶段检测像素的相应的深度(和 **Stencil**值(后面会讲), 使用这些, 来检查这个像素是否在另一个物体的前面或后面, 如此做到相应取舍。这个阶段也会检查 **alpha 值(alpha** 值是一个物体的透明度值)和物体之间的**混合(Blend)**)。所以, 即使在片段着色器中计算出来了一个像素所输出的颜色, 最后的像素颜色在渲染多个三角形的时候也可能完全不同。

正如你所见的那样, 图形渲染管线非常复杂, 它包含很多要配置的部分。然而, 对于大多数场合, 我们必须做的只是顶点和片段着色器。几何着色器和细分着色器是可选的, 通常使用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, 开始学习现代 `OpenGL` 的时候非常困难, 因为在你能够渲染自己的第一个三角形之前需要一大堆知识。

本节结束就是你可以最终渲染出你的三角形的时候，你也会了解到很多图形编程知识。

顶点输入(Vertex Input)

开始绘制一些东西之前，我们必须给 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在 3D 坐标里(x、y 和 z)。OpenGL 不是简单的把你所有的 3D 坐标变换为你屏幕上的 2D 像素；OpenGL 只是在当它们的 3 个轴(x、y 和 z)在特定的 -1.0 到 1.0 的范围内时才处理 3D 坐标。所有在这个范围内的坐标叫做**标准化设备坐标(Normalized Device Coordinates, NDC)**会最终显示在你的屏幕上(所有出了这个范围的都不会显示)。

由于我们希望渲染一个三角形，我们指定所有的这三个顶点都有一个 3D 位置。我们把它们以 `GLfloat` 数组的方式定义为标准化设备坐标(也就是在 OpenGL 的可见区域)中。

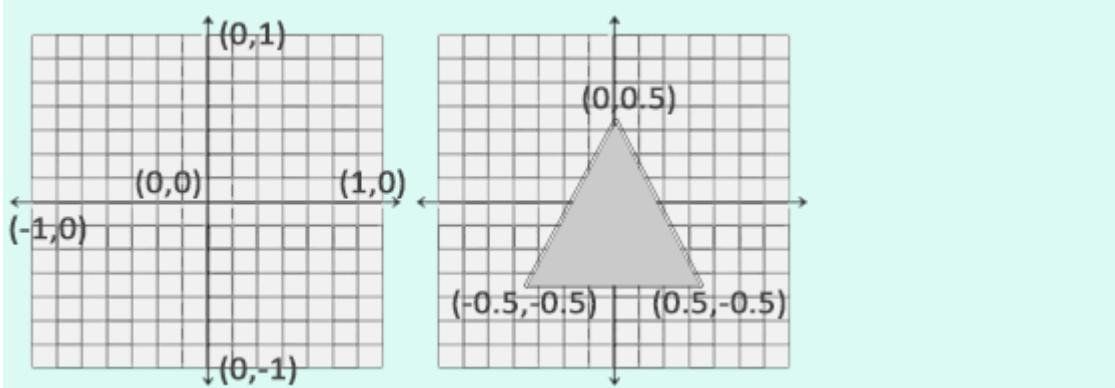
```
GLfloat vertices[] = {  
    -0.5f, -0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    0.0f, 0.5f, 0.0f  
};
```

由于 OpenGL 是在 3D 空间中工作的，我们渲染一个 2D 三角形，它的每个顶点都要有同一个 z 坐标 0.0。在这样的方式中，三角形的每一处的深度(Depth, [译注 2](#))都一样，从而使它看上去就像 2D 的。

Important

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 轴正方向上的点和(0,0)坐标是这个图像的中心，而不是左上角。最后你希望所有(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。最后的屏幕空间坐标被变换为像素输入到片段着色器。

有了这样的顶点数据，我们会把它作为输入数据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 上创建储存空间用于储存我们的顶点数据，还要配置 OpenGL 如何解释这些内存，并且指定如何发送给显卡。顶点着色器接着会处理我们告诉它要处理内存中的顶点的数量。

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)中储存大批顶点。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，所以无论何处我们都要尝试尽量一次性发送尽可能多的数据。当数据到了显卡内存中时，顶点着色器几乎立即就能获得顶点，这非常快。

顶点缓冲对象(VBO)是我们在 OpenGL 教程中第一个出现的 OpenGL 对象。就像 OpenGL 中的其他对象一样，这个缓冲有一个独一无二的 ID，所以我们可以使用 `glGenBuffers` 函数生成一个缓冲 ID：

```
GLuint VBO;
```

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

OpenGL 有很多缓冲对象类型，`GL_ARRAY_BUFFER` 是其中一个顶点缓冲对象的缓冲类型。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` 上：

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

从这一刻起，我们使用的任何缓冲函数(在 `GL_ARRAY_BUFFER` 目标上)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。它的第一个参数是我们希望把数据复制到上面的缓冲类型：顶点缓冲对象当前绑定到 `GL_ARRAY_BUFFER` 目标上。第二个参数指定我们希望传递给缓冲的数据的大小(以字节为单位)；用一个简单的 `sizeof` 计算出顶点数据就行。第三个参数是我们希望发送的真实数据。

第四个参数指定了我们希望显卡如何管理给定的数据。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，所以它使用的类型最好是 `GL_STATIC_DRAW`。如果，比如，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`。这样就能确保图形卡把数据放在高速写入的内存部分。

现在我们把顶点数据储存在显卡的内存中，用 VBO 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，来处理这些数据。现在我们开始着手创建它们吧。

顶点着色器(Vertex Shader)

顶点着色器是几个着色器中的一个，它是可编程的。现代 OpenGL 需要我们至少设置一个顶点和一个片段着色器，如果我们打算做渲染的话。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。下个教程里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** 写顶点着色器，然后编译这个着色器，这样我们就可以在应用中使用它了。下面你会看到一个非常基础的顶点着色器的源代码，它就是使用 **GLSL** 写的：

```
#version 330 core  
  
layout (location = 0) in vec3 position;  
  
void main()  
{  
    gl_Position = vec4(position.x, position.y, position.z, 1.0);  
}
```

就像你所看到的那样，**GLSL** 看起来很像 **C** 语言。每个着色器都起始于一个版本声明。这是因为 **OpenGL 3.3** 和更高的 **GLSL** 版本号要去匹配 **OpenGL** 的版本 (**GLSL420** 版本对应于 **OpenGL 4.2**)。我们同样显式地表示我们会用核心模式 (**Core-profile**)。

下一步，我们在顶点着色器中声明所有的输入顶点属性，使用 **in** 关键字。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(**Attribute**)。**GLSL** 有一个向量数据类型，它包含 1 到 4 个 **float** 元素，包含的数量可以从它的后缀看出来。由于每个顶点都有一个 **3D** 坐标，我们就创建一个 **vec3** 输入变量来表示位置(**Position**)。我们同样也指定输入变量的位置值(**Location**)，这是用 **layout (location = 0)** 来完成的，你后面会看到为什么我们会需要这个位置值。

Important

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位置和方向，二者是有用的数学属性。在 **GLSL** 中一个向量有最多 4 个元素，每个元素值都可以从各自代表一个空间坐标的 **vec.x**、**vec.y**、**vec.z** 和 **vec.w** 来获取到。注意 **vec.w** 元素不是用作表达空间中的位置的(我们处理的是 **3D** 不是 **4D**)而是用在所谓透视划分(**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，这个位置数据是一个 `vec4` 类型的。在 `main` 函数的最后，无论我们给 `gl_Position` 设置成什么，它都会成为着色器的输出。由于我们的输入是一个 3 元素的向量，我们必须把它转换为 4 元素。我们可以通过把 `vec3` 数据作为 `vec4` 初始化构造器的参数，同时把 `w` 元素设置为 `1.0f`（我们会在后面解释为什么）。

这个顶点着色器可能是能想到的最简单的了，因为我们什么都没有处理就把输入数据输出了。在真实的应用里输入数据通常都没有在标准化设备坐标中，所以我们首先就必须把它们放进 OpenGL 的可视区域内。

编译一个着色器

我们已经写了一个顶点着色器源码，但是为了 OpenGL 能够使用它，我们必须在运行时动态编译它的源码。

我们要做的第一件事是创建一个着色器对象，再次引用它的 ID。所以我们储存这个顶点着色器为 `GLuint`，然后用 `glCreateShader` 创建着色器：

```
GLuint vertexShader;  
  
vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

我们把着色器的类型提供 `glCreateShader` 作为它的参数。这里我们传递的参数是 `GL_VERTEX_SHADER` 这样就创建了一个顶点着色器。

下一步我们把这个着色器源码附加到着色器对象，然后编译它：

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

`glShaderSource` 函数把着色器对象作为第一个参数来编译它。第二参数指定了源码中有多少个字符串，这里只有一个。第三个参数是顶点着色器真正的源码，我们可以把第四个参数先设置为 `NULL`。

Important

你可能会希望检测调用 `glCompileShader` 后是否编译成功了，是否要去修正错误。检测编译时错误的方法是：

```
GLint success;
GLchar infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

首先我们定义一个整型来表示是否成功编译，还需要一个储存错误消息的容器(如果有的话)。然后我们用 `glGetShaderiv` 检查是否编译成功了。如果编译失败，我们应该用 `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任何错误，顶点着色器就被编译成功了。

片段着色器(Fragment Shader)

片段着色器是第二个也是最终我们打算创建的用于渲染三角形的着色器。片段着色器的全部，都是用来计算你的像素的最后颜色输出。为了让事情比较简单，我们的片段着色器只输出橘黄色。

Important

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)元素，通常缩写为 RGBA。当定义一个 OpenGL 或 GLSL 的颜色的时候，我们就把每个颜色的强度设置在 0.0 到 1.0 之间。比如，我们设置红色为 1.0f，绿色为 1.0f，这样这个混合色就是黄色了。这三种颜色元素的不同调配可以生成 1600 万不同颜色！

```
#version 330 core
```

```
out vec4 color;
```

```
void main()
```

```
{
```

```
color = 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 元素表示的最终输出颜色的向量，我们可以自己计算出来。我们可以用 `out` 关键字声明输出变量，这里我们命名为 `color`。下面，我们简单的把一个带有 `alpha` 值为 1.0(1.0 代表完全不透明)的橘黄的 `vec4` 赋值给 `color` 作为输出。

编译片段着色器的过程与顶点着色器相似，尽管这次我们使用 `GL_FRAGMENT_SHADER` 作为着色器类型：

```
GLuint fragmentShader;  
  
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);  
  
glShaderSource(fragmentShader, 1, &fragmentShaderSource, null);  
  
glCompileShader(fragmentShader);
```

每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着色器程序中(**Shader Program**)，它是用来渲染的。

着色器程序

着色器程序对象(**Shader Program Object**)是多个着色器最后链接的版本。如果要使用刚才编译的着色器我们必须把它们链接为一个着色器程序对象，然后当渲染物体的时候激活这个着色器程序。激活了的着色器程序的着色器，在调用渲染函数时才可用。

把着色器链接为一个程序就等于把每个着色器的输出链接到下一个着色器的输入。如果你的输出和输入不匹配那么就会得到一个链接错误。

创建一个程序对象很简单：

```
GLuint shaderProgram;  
  
shaderProgram = glCreateProgram();
```

`glCreateProgram` 函数创建一个程序，返回新创建的程序对象的 ID 引用。现在我们需要把前面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);
```

代码不言自明，我们把着色器附加到程序上，然后用 `glLinkProgram` 链接。

Important

就像着色器的编译一样，我们也可以检验链接着色器程序是否失败，获得相应的日志。与 `glGetShaderiv` 和 `glGetShaderInfoLog` 不同，现在我们使用：

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

我们可以调用 `glUseProgram` 函数，用新创建的程序对象作为它的参数，这样就能激活这个程序对象：

```
glUseProgram(shaderProgram);
```

现在在 `glUseProgram` 函数调用之后的每个着色器和渲染函数都会用到这个程序对象(当然还有这些链接的着色器)了。

在我们把着色器对象链接到程序对象以后，不要忘记删除着色器对象；我们不再需要它们了：

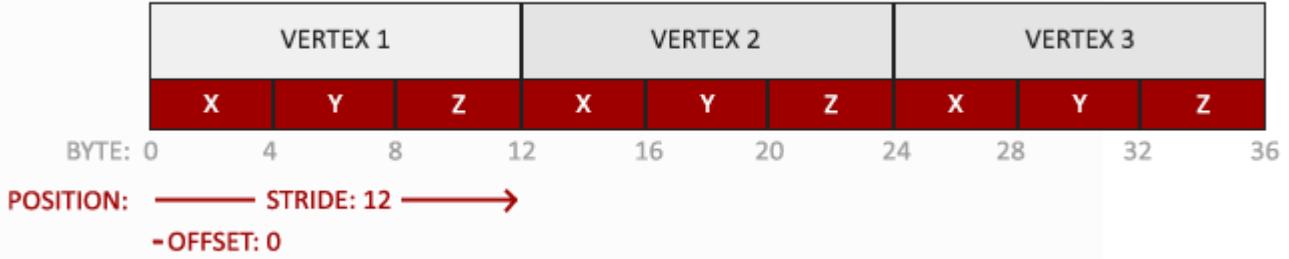
```
glDeleteShader(vertexShader);  
glDeleteShader(fragmentShader);
```

现在，我们把输入顶点数据发送给 GPU，指示 GPU 如何在顶点和片段着色器中处理它。还没结束，OpenGL 还不知道如何解释内存中的顶点数据，以及怎样把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以任何我们想要的形式作为顶点属性(Vertex Attribute)的输入，同样它也具有很强的灵活性，这意味着我们必须手动指定我们的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。这意味着我们必须在渲染前指定OpenGL如何解释顶点数据。

我们的顶点缓冲数据被格式化为下面的形式：



- 位置数据被储存为 32-bit(4 byte)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)。这几个值紧密排列为一个数组。
- 数据中第一个值是缓冲的开始位置。

有了这些信息我们就可以告诉 OpenGL 如何解释顶点数据了(每一个顶点属性)，我们使用 `glVertexAttribPointer` 这个函数：

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),  
(GLvoid*)0);  
glEnableVertexAttribArray(0);
```

`glVertexAttribPointer` 函数有很多参数，所以我们仔细来了解它们：

- 第一个参数指定我们要配置哪一个顶点属性。记住，我们在顶点着色器中使用 `layout(location = 0)` 定义了顶点属性——位置(Position)的位置值(Location)。这样要把顶点属性的位置值(Location)设置为 0，因为我们希望把数据传递到这个顶点属性中，所以我们在那里填 0。
- 第二个参数指定顶点属性的大小。顶点属性是 `vec3` 类型，它由 3 个数值组成。
- 第三个参数指定数据的类型，这里是 `GL_FLOAT`(GLSL 中 `vec*` 是由浮点数组成的)。

- 下个参数定义我们是否希望数据被标准化。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之间间隔有多少。由于下个位置数据在 3 个 `GLfloat` 后面的位置, 我们把步长设置为 `3 * sizeof(GLfloat)`。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)。每当我们有更多的顶点属性, 我们就必须小心地定义每个顶点属性之间的空间, 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)。
- 最后一个参数有奇怪的 `GLvoid*` 的强制类型转换。它表示我们的位置数据在缓冲中起始位置的偏移量。由于位置数据是数组的开始, 所以这里是 0。我们会在后面详细解释这个参数。

Important

每个顶点属性从 VBO 管理的内存中获得它的数据, 它所获取数据的那个 VBO, 就是当调用 `glVertexAttribPointer` 的时候, 最近绑定到 `GL_ARRAY_BUFFER` 的那个 VBO。由于在调用 `glVertexAttribPointer` 之前绑定了 VBO, 顶点属性 0 现在链接到了它的顶点数据。

现在我们定义 OpenGL 如何解释顶点数据, 我们也要开启顶点属性, 使用 `glEnableVertexAttribArray`, 把顶点属性位置值作为它的参数; 顶点属性默认是关闭的。自此, 我们把每件事都做好了: 我们使用一个顶点缓冲对象初始化了一个缓冲中的顶点数据, 设置了一个顶点和片段着色器, 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 绘制一个物体, 看起来会像是这样:

```
// 0. 复制顶点数组到缓冲中提供给 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(GLfloat),
                      (GLvoid*)0);
```

```
glEnableVertexAttribArray(0);  
  
// 2. 当我们打算渲染一个物体时要使用着色器程序  
  
glUseProgram(shaderProgram);  
  
// 3. 绘制物体  
  
someOpenGLFunctionThatDrawsOurTriangle();
```

我们绘制一个物体的时候必须重复这件事。这看起来也不多，但是如果有超过 5 个顶点属性，100 多个不同物体呢(这其实并不罕见)。绑定合适的缓冲对象，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的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？

顶点数组对象(Vertex Array Object, VAO)

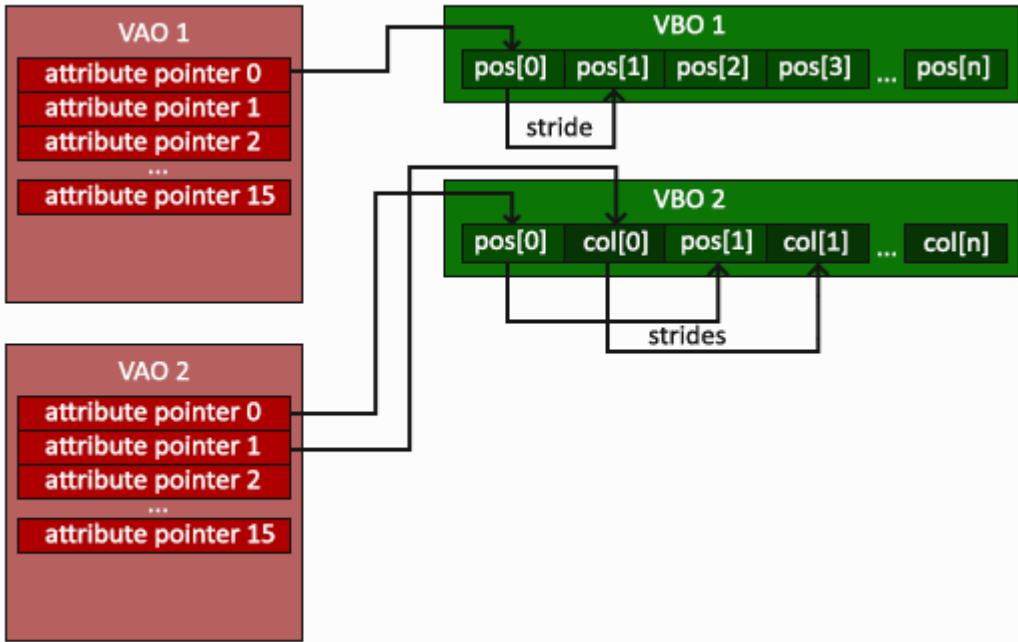
顶点数组对象(**Vertex Array Object, VAO**)可以像顶点缓冲对象一样绑定，任何随后的顶点属性调用都会储存在这个 VAO 中。这有一个好处，当配置顶点属性指针时，你只用做一次，每次绘制一个物体的时候，我们绑定相应 VAO 就行了。切换不同顶点数据和属性配置就像绑定一个不同的 VAO 一样简单。所有状态我们都放到了 VAO 里。

Attention

OpenGL 核心模式版要求我们使用 VAO，这样它就能知道对我们的顶点输入做些什么。如果我们绑定 VAO 失败，OpenGL 会拒绝绘制任何东西。

一个顶点数组对象储存下面的内容：

- 调用 `glEnableVertexAttribArray` 和 `glDisableVertexAttribArray`。
- 使用 `glVertexAttribPointer` 的顶点属性配置。
- 使用 `glVertexAttribPointer` 进行的顶点缓冲对象与顶点属性链接。



生成一个 VAO 和生成 VBO 类似:

```
GLuint VAO;
```

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

使用 VAO 要做的全部就是使用 `glBindVertexArray` 绑定 VAO。自此我们就应该绑定/配置相应的 VBO 和属性指针，然后解绑 VAO 以备后用。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 绑定到希望用到的配置就行了。这段代码应该看起来像这样：

```
// ...: 初始化代码 (一次完成 (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提供给 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(GLfloat),  
(GLvoid * )0);  
glEnableVertexAttribArray(0);
```

//4. 解绑 VAO

```
glBindVertexArray(0);
```

[...]

// ...: 绘制代码 (in Game Loop) :: ..

// 5. 绘制物体

```
glUseProgram(shaderProgram);
```

```
glBindVertexArray(VAO);
```

```
someOpenGLFunctionThatDrawsOurTriangle();
```

```
glBindVertexArray(0);
```

Attention

通常情况下当我们配置好它们以后要解绑 OpenGL 对象，这样我们才不会在某处错误地配置它们。

就是现在！前面做的一切都是等待这一刻，我们已经把我们的顶点属性配置和打算使用的 VBO 储存到一个 VAO 中。一般当你有多个物体打算绘制时，你首先要生成/配置所有的 VAO(它需要 VBO 和属性指针)，然后储存它们准备后面使用。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。

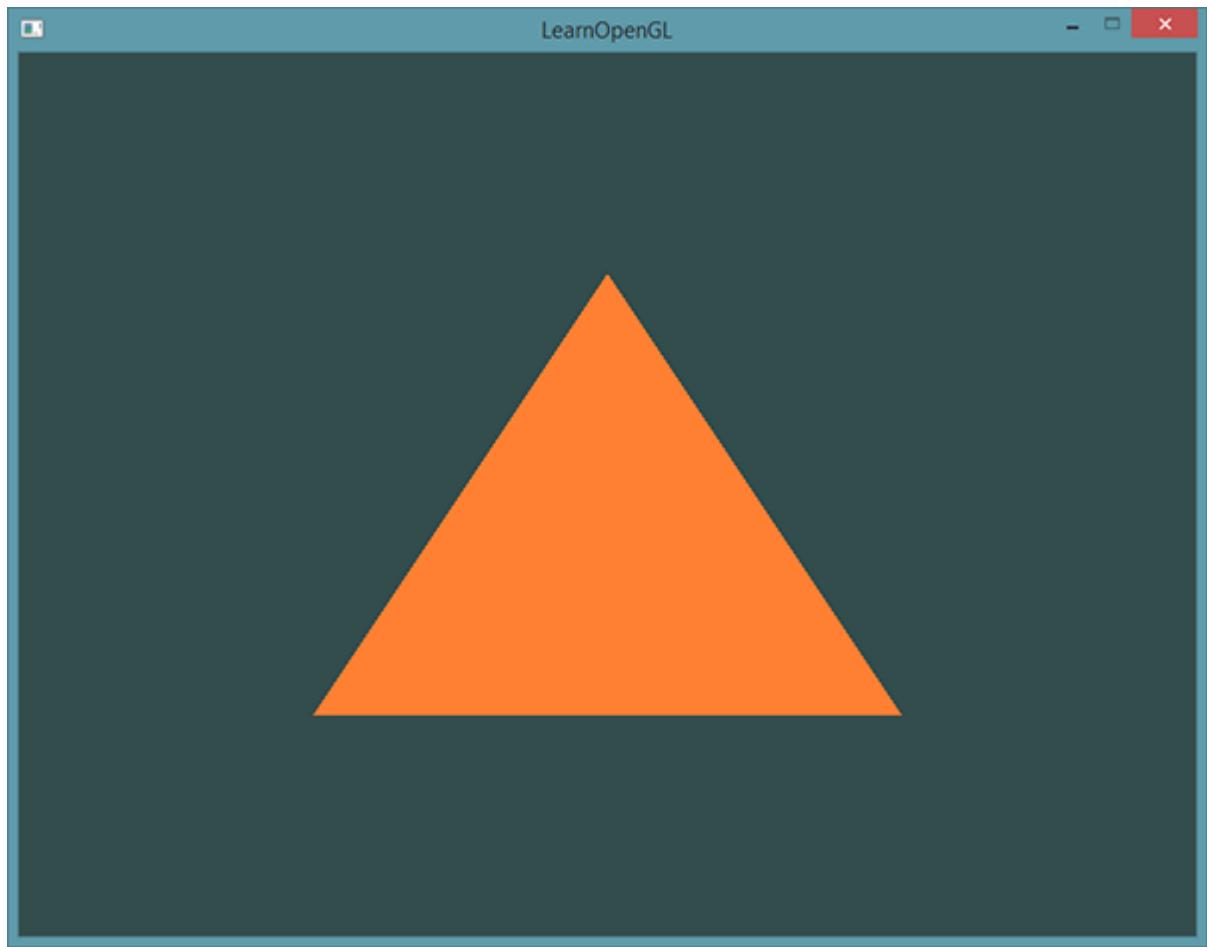
我们一直期待的三角形

OpenGL 的 `glDrawArrays` 函数为我们提供了绘制物体的能力，它使用当前激活的着色器、前面定义的顶点属性配置和 VBO 的顶点数据(通过 VAO 间接绑定)来绘制基本图形。

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

`glDrawArrays` 函数第一个参数是我们打算绘制的 OpenGL 基本图形的类型。由于我们在一开始时说过，我们希望绘制三角形，我们传递 `GL_TRIANGLES` 给它。第二个参数定义了我们打算绘制的那个顶点数组的起始位置的索引；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渲染一个三角形，它只有 `3` 个顶点)。

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不一样，你可能做错了什么，去看源码，看看是否遗漏了什么东西或者在评论部分提问。

索引缓冲对象(Element Buffer Objects, EBO)

这是我们最后一件在渲染顶点这个问题上要讨论的事——索引缓冲对象简称 EBO(或 IBO)。解释索引缓冲对象的工作方式最好是举例子：假设我们不再绘制一个三角形而是矩形。我们就可以绘制两个三角形来组成一个矩形(OpenGL 主要就是绘制三角形)。这会生成下面的顶点的集合：

```
GLfloat 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
```

```
};
```

就像你所看到的那样，有几个顶点叠加了。我们指定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。当我们有超过 1000 个三角的模型这个问题会更糟糕，这会产生一大堆浪费。最好的解决方案就是每个顶点只储存一次，当我们打算绘制这些顶点的时候只调用顶点的索引。这种情况我们只要储存 4 个顶点就能绘制矩形了，我们只要指定我们打算绘制的那个顶点的索引就行了。如果 OpenGL 提供这个功能就好了，对吧？

很幸运，索引缓冲的工作方式正是这样的。一个 EBO 是一个像顶点缓冲对象 (VBO)一样的缓冲，它专门储存索引，OpenGL 调用这些顶点的索引来绘制。索引绘制正是这个问题的解决方案。我们先要定义(独一无二的)顶点，和绘制出矩形的索引：

```
GLfloat 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
```

```
};
```

```
GLuint indices[] = { // 起始于0!
```

```
    0, 1, 3, // 第一个三角形
```

```
    1, 2, 3 // 第二个三角形
```

```
};
```

你可以看到，当时用索引的时候，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：

```
GLuint EBO;
```

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

与 VBO 相似，我们绑定 EBO 然后用 `glBufferData` 把索引复制到缓冲里。同样，和 VBO 相似，我们会把这些函数调用放在绑定和解绑函数调用之间，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);
```

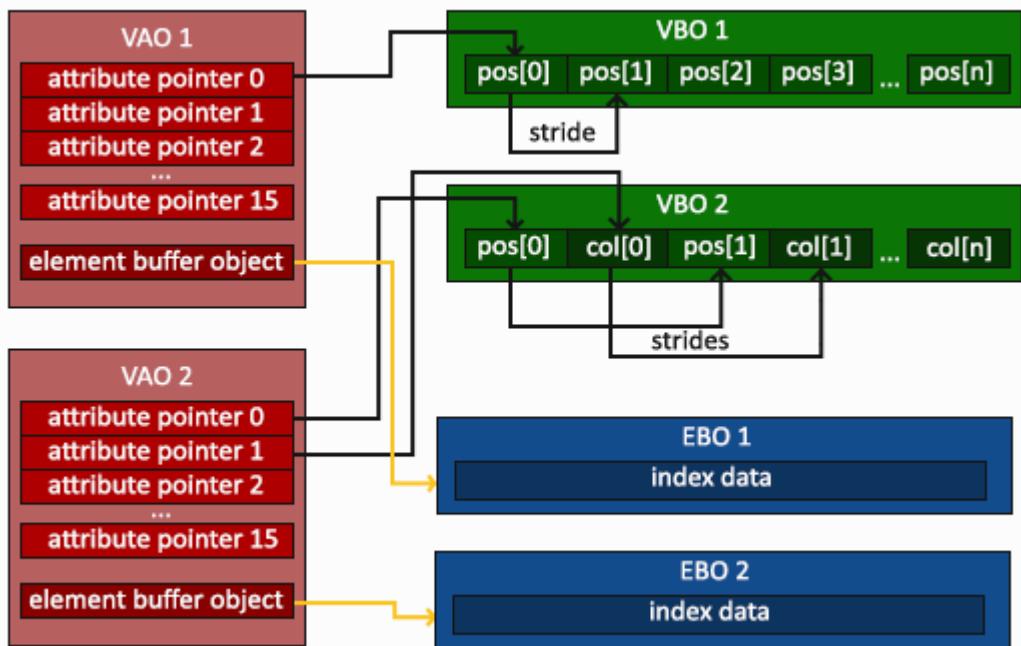
要注意的是，我们现在用 `GL_ELEMENT_ARRAY_BUFFER` 当作缓冲目标。最后一件要做的事是用 `glDrawElements` 来替换 `glDrawArrays` 函数，来指明我们从索引缓冲渲染。当时用 `glDrawElements` 的时候，我们就会用当前绑定的索引缓冲进行绘制：

```
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次数。我们填 6，说明我们总共想绘制 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，但是这只是当你不是在使用索引缓冲对象的时候)，但是我们只打算在这里填写 0。

`glDrawElements` 函数从当前绑定到 `GL_ELEMENT_ARRAY_BUFFER` 目标的 EBO 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。不过顶点数组对象仍可以保存索引缓冲对象的绑定状态。VAO 绑定之后可以索引缓冲对象，EBO 就成为了 VAO 的索引缓冲对象。再次绑定 VAO 的同时也会自动绑定 EBO。



Attention

当目标是 `GL_ELEMENT_ARRAY_BUFFER` 的时候，VAO 储存了 `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 VAO
```

```
glBindVertexArray(VAO);
```

```
// 2. 把我们的顶点数组复制到一个顶点缓冲中，提供给 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，提供给 OpenGL 使用

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
```

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

// 3. 设置顶点属性指针

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),  
(GLvoid * )0);
```

```
 glEnableVertexAttribArray(0);
```

// 4. 解绑 VAO，不解绑 EBO (译注：解绑缓冲相当于没有绑定缓冲，可以在解绑 VAO

之后解绑缓冲)

```
 glBindVertexArray(0);
```

[...]

// ...: 绘制代码(在游戏循环中) :: ...

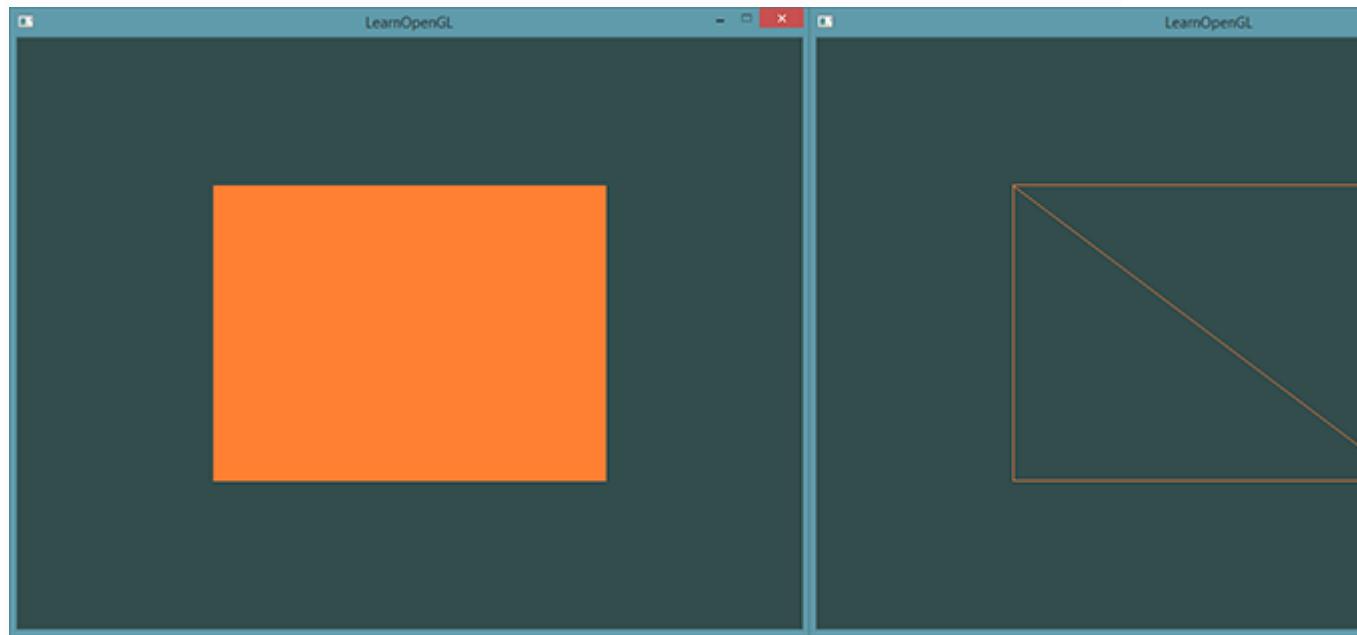
```
 glUseProgram(shaderProgram);
```

```
 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。



Important

线框模式(Wireframe Mode)

如果用线框模式绘制你的三角，你可以配置 OpenGL 绘制用的基本图形，调用 `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`。第一个参数说：我们打算应用到所有的三角形的前面和背面，第二个参数告诉我们用线来绘制。在随后的绘制函数调用后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)` 设置回了默认模式。

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[在这里获得全部源码](#)，也可以在评论区自由提问。

如果你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需要很多知识。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: 一个渲染第一个三角形的教程。

- open.gl/drawing: Alexander Overvoorde 的关于渲染第一个三角形的教程。
- antongerdelan.net/vertexbuffers: 顶点缓冲对象的一些深入探讨。

练习

为了更好的理解讨论的那些概念最好做点练习。建议在继续下面的主题之前做完这些练习，确保你对这些有比较好的理解。

- 尝试使用 `glDrawArrays` 以在你的数据中添加更多顶点的方式，绘制两个彼此相连的三角形：[参考解答](#)
- 现在，使用不同的 VAO(和 VBO)创建同样的 2 个三角形，每个三角形的数据要不同(提示：创建 2 个顶点数据数组，而不是 1 个)：[参考解答](#)
- 创建连个着色器程序(Shader Program)，第二个程序使用不同的片段着色器，它输出黄色；绘制这两个三角形，其中一个输出为黄色：[参考解答](#)

着色器(Shader)

原文	Shaders
作者	JoeyDeVries
翻译	Django
校对	Geequlim

在 [Hello Triangle](#) 教程中提到，着色器是运行在 GPU 上的小程序。这些小程序为图形渲染管线的一个特定部分而运行。从基本意义上来说，着色器不是别的，只是一种把输入转化为输出的程序。着色器也是一种相当独立的程序，它们不能相互通信；只能通过输入和输出的方式来进行沟通。

前面的教程里我们简要地触及了一点着色器的皮毛。了解了如何恰当地使用它们。现在我们会用一种更加通用的方式详细解释着色器，特别是 OpenGL 着色器语言。

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。每个着色器的入口都是 main 函数，在这里我们处理所有输入变量，用输出变量输出结果。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
    ...
    // 输出
    out_variable_name = weird_stuff_we_processed;
}
```

当我们谈论特别是谈到顶点着色器的时候，每个输入变量也叫顶点属性(Vertex Attribute)。能声明多少个顶点属性是由硬件决定的。OpenGL 确保至少有 16 个包含 4 个元素的顶点属性可用，但是有些硬件或许可用更多，你可以查询 [GL_MAX_VERTEX_ATTRIBS](#) 来获取这个数目。

```
GLint nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
```

```
std::cout << "Maximum nr of vertex attributes supported: " <<
```

```
nrAttributes << std::endl;
```

通常情况下它会返回至少 16 个，大部分情况下是够用了。

数据类型

GLSL 有像其他编程语言相似的数据类型。GLSL 有 C 风格的默认基础数据类型：`int`、`float`、`double`、`uint` 和 `bool`。GLSL 也有两种容器类型，教程中我们会使用很多，它们是向量(Vector)和矩阵(Matrix)，其中矩阵我们在之后的教程里再讨论。

向量(Vector)

GLSL 中的向量可以包含有 1、2、3 或者 4 个分量，分量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 代表元素数量):

类型	含义
<code>vecn</code>	包含 n 个默认为 <code>float</code> 元素的向量
<code>bvecn</code>	包含 n 个布尔元素向量
<code>ivecn</code>	包含 n 个 <code>int</code> 元素的向量
<code>uvecn</code>	包含 n 个 <code>unsigned int</code> 元素的向量
<code>dvecn</code>	包含 n 个 <code>double</code> 元素的向量

大多数时候我们使用 `vecn`，因为 `float` 足够满足大多数要求。

一个向量的元素可以通过 `vec.x` 这种方式获取，这里 `x` 是指这个向量的第一个元素。你可以分别使用 `.x`、`.y`、`.z` 和 `.w` 来获取它们的第 1、2、3、4 号元素。GLSL 也允许你使用 `rgba` 来获取颜色的元素，或是 `stpq` 获取纹理坐标元素。

向量的数据类型也允许一些有趣而灵活的元素选择方式，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;
```

你可以使用上面任何 4 个字母组合来创建一个新的和原来向量一样长的向量(但 4 个元素需要是同一种类型); 不允许在一个 `vec2` 向量中去获取.`z` 元素。我们可以把一个向量作为一个参数传给不同的向量构造函数, 以减少参数需求的数量:

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

向量是一种灵活的数据类型, 我们可以把用在所有输入和输出上。学完教程你会看到很多如何创造性地管理向量的例子。

输入与输出(in vs out)

着色器是各自独立的小程序, 但是它们都是一个整体的局部, 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` 关键字来实现这个目的。每个着色器使用这些关键字定义输入和输出, 无论在哪儿, 一个输出变量就能与一个下一个阶段的输入变量相匹配。他们在顶点和片段着色器之间有点不同。

顶点着色器应该接收的输入是一种特有形式, 否则就会效率低下。顶点着色器的输入是特殊的, 它所接受的是从顶点数据直接输入的。为了定义顶点数据被如何组织, 我们使用 `location` 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 `layout (location = 0)`。顶点着色器需要为它的输入提供一个额外的 `layout` 定义, 这样我们才能把它链接到顶点数据。

Important

也可以移除 `layout (location = 0)`, 通过在 `OpenGL` 代码中使用 `glGetAttribLocation` 请求属性地址(`Location`), 但是我更喜欢在着色器中设置它们, 理解容易而且节省时间。

另一个例外是片段着色器需要一个 `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同名输入。当名字和类型都一样的时候，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(Program)对象时完成的)。为了展示这是这么工作的，我们会改变前面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 position; // 位置变量的属性为0

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

void main()
{
    gl_Position = vec4(position, 1.0); // 把一个vec3 作为vec4 的构造
    // 器的参数
    vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f); // 把输出颜色设置为暗
    // 红色
}
```

片段着色器

```
#version 330 core
```

```
in vec4 vertexColor; // 和顶点着色器的vertexColor 变量类型相同、名称相
```

```
同
```

```
out vec4 color; // 片段着色器输出的变量名可以任意命名，类型必须是vec4
```

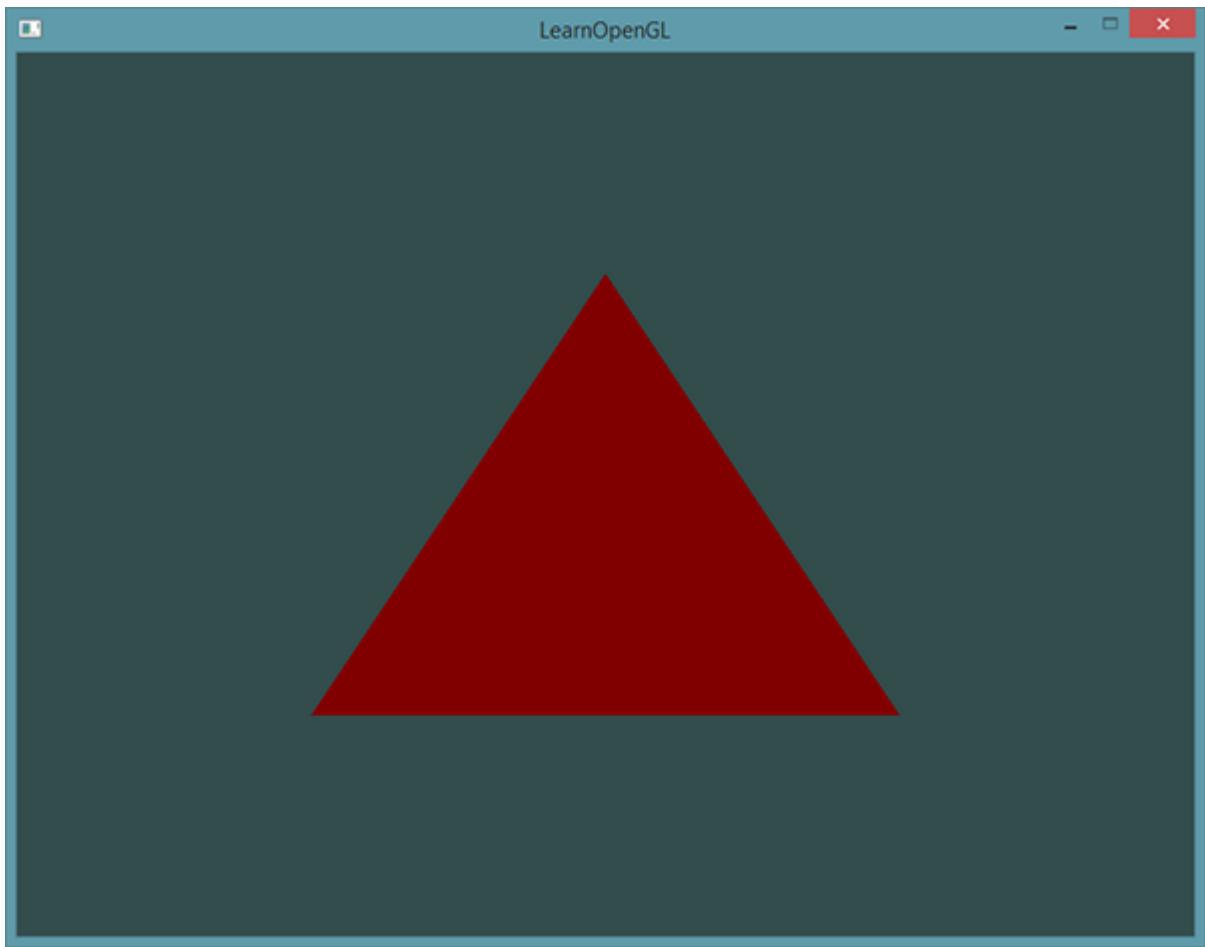
```
void main()
```

```
{
```

```
    color = 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，在片段着色器声明了一个一样的 `vertexColor`。由于它们类型相同并且名字也相同，片段着色器中的 `vertexColor` 就和顶点着色器中的 `vertexColor` 链接了。因为我们在顶点着色器中设置的颜色是深红色的，片段着色器输出的结果也是深红色的。下面的图片展示了输出结果：



我们完成了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` 是另一种从 CPU 应用向 GPU 着色器发送数据的方式，但 `uniform` 和顶点属性有点不同。首先，`uniform` 是全局的(**Global**)。这里全局的意思是 `uniform` 变量必须在所有着色器程序对象中都是独一无二的，它可以在着色器程序的任何着色器任何阶段使用。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。

我们可以简单地通过在片段着色器中设置 `uniform` 关键字接类型和变量名来声明一个 GLSL 的 `uniform`。之后，我们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 color;
```

```
uniform vec4 ourColor; //在程序代码中设置
```

```
void main()
```

```
{
```

```
    color = 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。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` 所以不用在那里定义它。

Attention

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，从而最后编译出的版本中并不会包含它，如果有一个从没用过的 `uniform` 出现在已编译版本中会出现几个错误，记住这点！

`uniform` 现在还是空的；我们没有给它添加任何数据，所以下面就做这件事。我们首先需要找到着色器中 `uniform` 的索引/地址。当我们得到 `uniform` 的索引/地址后，我们就可以更新它的值了。这里我们不去给像素传递一个颜色，而是随着时间让它改变颜色：

```
GLfloat timeValue = glfwGetTime();
```

```
GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
```

```
GLint 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。然后我们使用余弦函数在 0.0 到 -1.0 之间改变颜色，最后储存到 `greenValue` 里。

接着，我们用 `glGetUniformLocation` 请求 `uniform ourColor` 的地址。我们为请求函数提供着色器程序和 `uniform` 的名字(这是我们希望获得的地址的来源)。如果 `glGetUniformLocation` 返回 -1 就代表没有找到这个地址。最后，我们可以通过 `glUniform4f` 函数设置 `uniform` 值。注意，查询 `uniform` 地址不需要在之前使用着色器程序，但是更新一个 `uniform` 之前必须使用程序(调用 `glUseProgram`)，因为它是在当前激活的着色器程序中设置 `uniform` 的。

Important

因为 OpenGL 是 C 库内核，所以它不支持函数重载，在函数参数不同的时候就要定义新的函数；`glUniform` 是一个典型例子。这个函数有一个特定的作为类型的后缀。有几种可用的后缀：

后缀	含义
f	函数需要以一个 <code>float</code> 作为它的值
i	函数需要一个 <code>int</code> 作为它的值
ui	函数需要一个 <code>unsigned int</code> 作为它的值
3f	函数需要 3 个 <code>float</code> 作为它的值
fv	函数需要一个 <code>float</code> 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的函数。在我们的例子里，我们使用 `uniform` 的 `4float` 版，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帧更新这个 `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))
```

```
{
```

```
// 检测事件
```

```
glfwPollEvents();  
  
// 渲染  
  
// 清空颜色缓冲  
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);  
  
glClear(GL_COLOR_BUFFER_BIT);  
  
// 激活着色器  
glUseProgram(shaderProgram);  
  
// 更新uniform 颜色  
GLfloat timeValue = glfwGetTime();  
  
GLfloat greenValue = (sin(timeValue) / 2) + 0.5;  
  
GLint 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);  
  
glBindVertexArray(0);  
}
```

新代码和上一节的很相似。这次，我们在每个循环绘制三角形前先更新 uniform 值。如果你成功更新 uniform 了，你会看到你的三角形逐渐由绿变黑再变绿。

如果你在哪儿卡住了，[这里有源码](#)。

就像你所看到的那样，`uniform` 是个设置属性的很有用的工具，它可以在渲染循环中改变，也可以在你的应用和着色器之间进行数据交互，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。在顶点属性问题上一个更好的解决方案一定要能包含足够多的数据，这是我们接下来要讲的内容。

更多属性

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 VAO 里。这次，我们同样打算把颜色数据加进顶点数据中。我们将把颜色数据表示为 3 个 float 的顶点数组(**Vertex Array**)。我们为三角形的每个角分别指定为红色、绿色和蓝色：

```
GLfloat 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  
};
```

由于我们现在发送到顶点着色器的数据更多了，有必要调整顶点着色器，使它能够把颜色值作为一个顶点属性输入。需要注意的是我们用 `layout` 标识符来吧 `color` 属性的 `location` 设置为 1：

```
#version 330 core  
  
layout (location = 0) in vec3 position; // 位置变量的属性position 为 0  
layout (location = 1) in vec3 color; // 颜色变量的属性position 为 1  
  
out vec3 ourColor; // 向片段着色器输出一个颜色  
  
void main()
```

```
{
```

```
    gl_Position = vec4(position, 1.0);
```

```
    ourColor = color; // 把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，现在使用的 `ourColor` 输出变量要求必须也去改变片段着色器：

```
#version 330 core
```

```
in vec3 ourColor
```

```
out vec4 color;
```

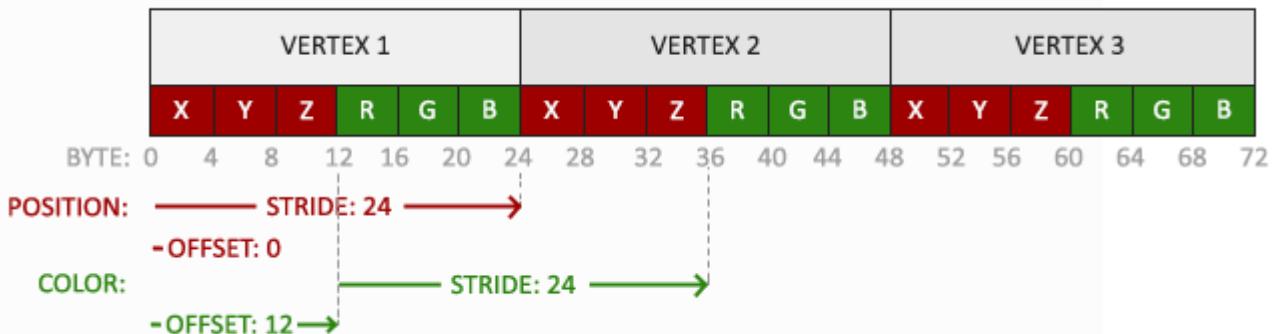
```
void main()
```

```
{
```

```
    color = vec4(ourColor, 1.0f);
```

```
}
```

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：



知道了当前使用的 `layout`，我们就可以使用 `glVertexAttribPointer` 函数更新顶点格式，

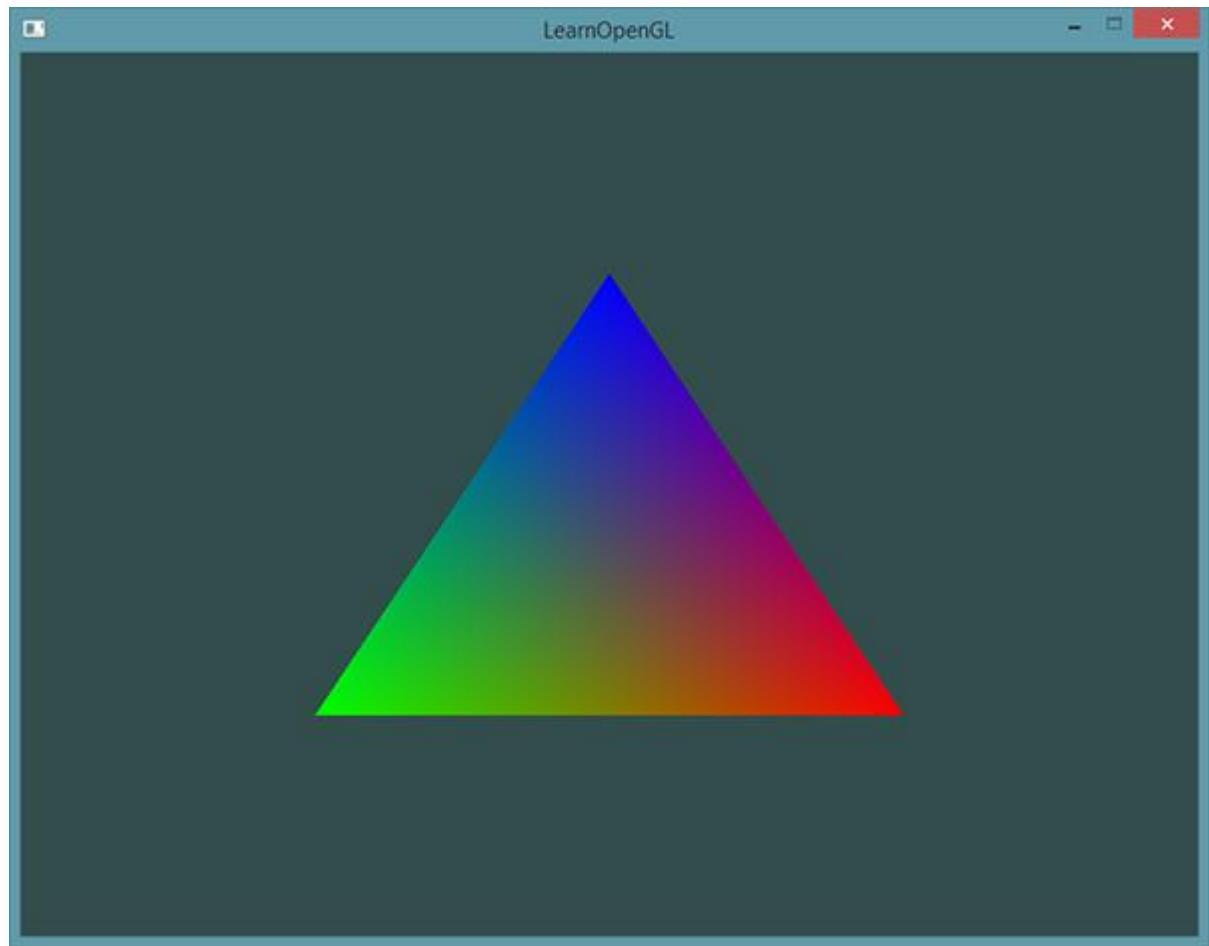
```
// 顶点属性  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat),  
(GLvoid*)0);  
  
glEnableVertexAttribArray(0);  
  
// 颜色属性  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat),  
(GLvoid*)(3 * sizeof(GLfloat)));  
  
glEnableVertexAttribArray(1);
```

`glVertexAttribPointer` 函数的前几个参数比较明了。这次我们配置属性 location 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(**Stride**)。为获得数据队列中下一个属性值(比如位置向量的下个 x 元素)我们必须向右移动 6 个 float，其中 3 个是位置值，另外三个是颜色值。这给了我们 6 个步长的大小，每个步长都是 float 的字节数(=24 字节)。

同样，这次我们必须指定一个偏移量(**Offset**)。对于每个顶点来说，位置(**Position**)顶点属性是先声明的，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(GLfloat)`，用字节来计算就是 12 字节。

运行应用你会看到如下结果：



如果你有困惑，可以[在这里获得源码](#)。

这个图片可能不是你所期望的那种，因为我们只提供 3 个颜色，而不是我们现在看到的大调色板。这是所谓片段着色器进行**片段插值(Fragment Interpolation)**的结果。当渲染一个三角形在像素化(Rasterization 也译为光栅化)阶段通常生成比原来的顶点更多的像素。像素器就会基于每个像素在三角形的所处相对位置决定像素的位置。

基于这些位置，它**插入(Interpolate)**所有片段着色器的输入变量。比如说，我们有一个线段，上面的那个点是绿色的，下面的点是蓝色的。如果一个片段着色器正在处理的那个片段(实际上就是像素)是在线段的 70% 的位置，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。

这正是这个三角形里发生的事。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 50,000 左右的像素，片段着色器为这些像素进行

插值。如果你仔细看这些颜色，你会发现其中的奥秘：红到紫再到蓝。像素插值会应用到所有片段着色器的输入属性上。

我们自己的着色器类

编写、编译、管理着色器是件麻烦事。在着色器的最后主题里，我们会写一个类来让我们的生活轻松一点，这个类从硬盘读着色器，然后编译和链接它们，对它们进行错误检测，这就变得很好用了。这也会给你一些关于如何把我们目前所学的知识封装到一个抽象的对象里的灵感。

我们会在头文件里创建整个类，主要为了学习，也可以方便移植。我们先来添加必要的 `include`，定义类结构：

```
#ifndef SHADER_H  
  
#define SHADER_H  
  
#include <string>  
#include <fstream>  
#include <sstream>  
#include <iostream>  
  
using namespace std;  
  
#include <GL/glew.h>; // 包含glew获取所有的OpenGL必要headers  
  
class Shader  
{  
public:  
    // 程序ID
```

```
GLuint Program;  
  
// 构造器读取并创建Shader  
  
Shader(const GLchar * vertexSourcePath, const GLchar *  
  
fragmentSourcePath);  
  
// 使用Program  
  
void Use();  
  
};  
  
#endif
```

Important

在上面，我们用了几个预处理指令(Preprocessor Directives)。这些预处理指令告知你的编译器，只在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 `shader` 头文件，它是用来防止链接冲突的。
`shader` 类保留了着色器程序的 ID。它的构造器需要顶点和片段着色器源代码的文件路径，我们可以把各自的文本文件储存在硬盘上。`Use` 函数看似平常，但是能够显示这个自造类如何让我们的生活变轻松(虽然只有一点)。

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([译注 1](#))

```
Shader(const GLchar * vertexPath, const GLchar * fragmentPath)  
  
{  
  
    // 1. 从文件路径获得vertex/fragment 源码  
  
    std::string vertexCode;  
  
    std::string fragmentCode;  
  
    try {
```

```
// 打开文件

std::ifstream vShaderFile(vertexPath);

std::ifstream fShaderFile(fragmentPath);

std::stringstream vShaderStream, fShaderStream;

// 读取文件缓冲到流

vShaderStream << vShaderFile.rdbuf();

fShaderStream << fShaderFile.rdbuf();

// 关闭文件句柄

vShaderFile.close();

fShaderFile.close();

// 将流转为GLchar 数组

vertexCode = vShaderStream.str();

fragmentCode = fShaderStream.str();

}

catch(std::exception e)

{

    std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" <<

    std::endl;

}
```

下一步，我们需要编译和链接着色器。注意，我们也要检查编译/链接是否失败，如果失败，打印编译错误，调试的时候这及其重要(这些错误日志你总会需要的)：

```
// 2. 编译着色器
```

```
GLuint vertex, fragment;
```

```
GLint success;
```

```
GLchar infoLog[512];
```

```
// 顶点着色器
```

```
vertex = glCreateShader(GL_VERTEX_SHADER);
```

```
glShaderSource(vertex, 1, &vShaderCode, NULL);
```

```
glCompileShader(vertex);
```

```
// 打印编译时错误
```

```
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
```

```
if(!success)
```

```
{
```

```
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
```

```
    std::cout << "ERROR::SHADER::VERTEX::COMPILE_FAILED\n" <<
```

```
    infoLog << std::endl;
```

```
};
```

```
// 对片段着色器进行类似处理
```

```
[...]
```

```
// 着色器程序
```

```
this->Program = glCreateProgram();
```

```
glAttachShader(this->Program, vertex);

glAttachShader(this->Program, fragment);

glLinkProgram(this->Program);

// 打印连接错误

glGetProgramiv(this->Program, GL_LINK_STATUS, &success);

if(!success)

{

    glGetProgramInfoLog(this->Program, 512, NULL, infoLog);

    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<

    infoLog << std::endl;

}

// 删除着色器

glDeleteShader(vertex);

glDeleteShader(fragment);
```

最后我们也要实现 **Use** 函数：

```
void Use()

{

    glUseProgram(this->Program);

}
```

现在我们写完了一个完整的着色器类。使用着色器类很简单；我们创建一个着色器对象以后，就可以简单的使用了：

```

Shader ourShader("path/to/shaders/shader.vs",
"path/to/shaders/shader.frag");
...
while(...)

{
    ourShader.Use();

    glUniform1f(glGetUniformLocation(ourShader.Program,
"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.frag` 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 `.frag` 作为扩展名很直观。

使用新着色器类的程序, 着色器类, 顶点着色器, 片段着色器。

练习

1. 修改顶点着色器让三角形上下颠倒: [参考解答](#)
2. 通过使用 `uniform` 定义一个水平偏移，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，把像素的颜色设置为与顶点位置相等(看看顶点位置值是如何在三角形中进行插值的)。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](#)

纹理(Textures)

原文	Textures
作者	JoeyDeVries

原文	Textures
翻译	Django
校对	Geequlim, BLumia

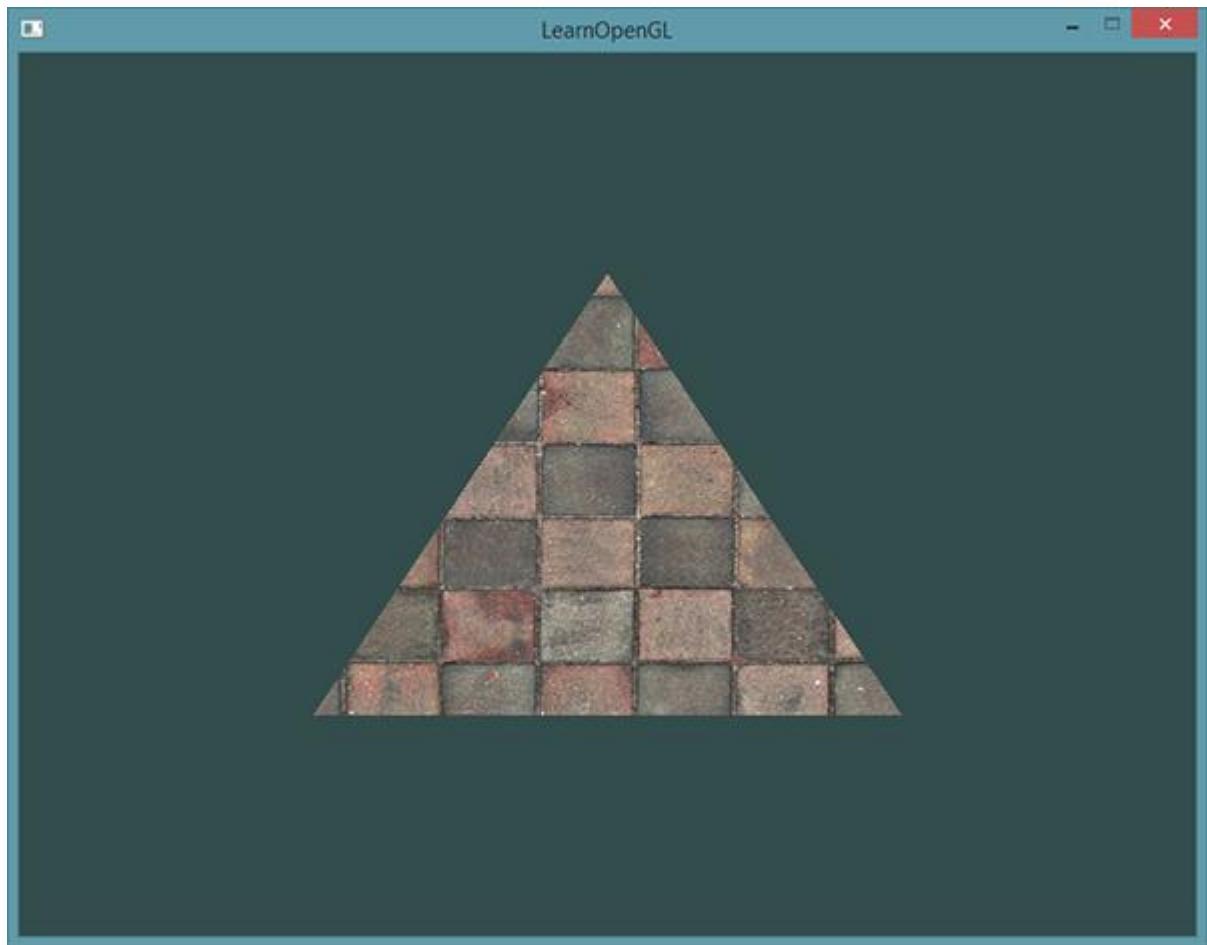
我们已经了解到，我们可以为每个顶点使用颜色来增加图形的细节，从而创建出有趣的图像。但是，如果想让图形看起来更真实我们就必须有足够多的顶点，从而指定足够多的颜色。这将会产生很多额外开销，因为每个模型都会需求更多的顶点和顶点颜色。

艺术家和程序员更喜欢使用**纹理(Texture)**。纹理是一个2D图片(也有1D和3D)，它用来添加物体的细节；这就像有一张绘有砖块的图片贴到你的3D的房子上，你的房子看起来就像一堵砖墙。因为我们可以在一张图片上插入足够多的细节，这样物体就会拥有很多细节而不用增加额外的顶点。

Important

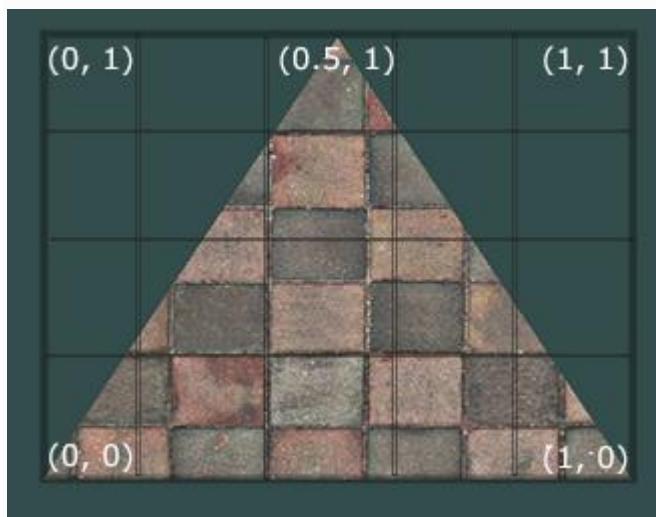
除了图像以外，纹理也可以储存大量的数据，这些数据用来发送到着色器上，但是这不是我们现在的主题。

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。这样每个顶点就会有一个纹理坐标(**Texture Coordinate**)，它指明从纹理图像的哪个地方采样(采集像素颜色)。之后在所有的其他的片段上进行片段插值(**Fragment Interpolation**)。

纹理坐标是 x 和 y 轴上 0 到 1 之间的范围(注意我们使用的是 2D 纹理图片)。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，终结于纹理图片的右上角(1,1)。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。



我们为三角形准备了 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，因此我们把三角左下角的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中间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(1.0,0)。我们只要传递这三个纹理坐标给顶点着色器就行了，接着片段着色器会为每个片段生成纹理坐标的插值。

纹理坐标看起来就像这样：

```
GLfloat texCoords[] = {  
    0.0f, 0.0f, // 左下角  
    1.0f, 0.0f, // 右下角  
    0.5f, 1.0f // 顶部位置  
};
```

纹理采样有几种不同的插值方式。我们需要自己告诉 OpenGL 在纹理中采用哪种采样方式。

纹理环绕方式(Texture Wrapping)

纹理坐标通常的范围是从(0, 0)到(1, 1)，如果我们把纹理坐标设置为范围以外会发生什么？OpenGL 默认的行为是重复这个纹理图像(我们简单地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	纹理的默认行为，重复纹理图像
GL_MIRRORED_REPEAT	和 GL_REPEAT 一样，除了重复的图片是镜像放置的
GL_CLAMP_TO_EDGE	纹理坐标会在 0 到 1 之间，超出的部分会重复纹理坐标的边缘，就是边缘被拉伸
GL_CLAMP_TO_BORDER	超出的部分是用户指定的边缘的颜色

当纹理坐标超出默认范围时，每个值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：



GL_REPEAT



GL_MIRRORED_REPEAT



GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER

前面提到的选项都可以使用 `glTexParameter` 函数单独设置每个坐标轴 `s`、`t`(如果是使用 3D 纹理那么还有一个 `r`)它们和 `x`、`y`(`z`)是相等的：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_MIRRORED_REPEAT); glTexParameteri(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去告知我们希望去设置哪个纹理轴。

我们打算设置的是 `WRAP` 选项，并且指定 `S` 和 `T` 轴。最后一个参数需要我们传递放置方式，在这个例子里面我们在当前激活纹理上应用 `GL_MIRRORED_REPEAT`。

如果我们选择 `GL_CLAMP_TO_BORDER` 选项，我们还要指定一个边缘的颜色。这次使用 `glTexParameter` 函数的 `fv` 后缀形式，加上 `GL_TEXTURE_BORDER_COLOR` 作为选项，这个函数需要我们传递一个边缘颜色的 `float` 数组作为颜色值：

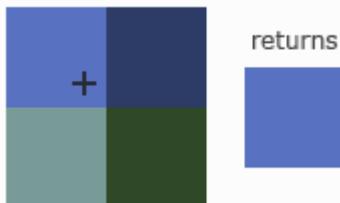
```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };

glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
borderColor);
```

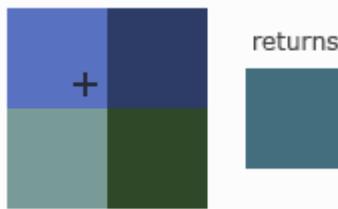
纹理过滤(Texture Filtering)

纹理坐标不依赖于解析度，它可以是任何浮点数值，这样 OpenGL 需要描述出哪个纹理像素对应哪个纹理坐标(**Texture Pixel**, 也叫 **Texel**, [译注 1](#))。当你有一个很大的物体但是纹理解析度很低的时候这就变得很重要了。你可能已经猜到了，OpenGL 也有一个叫做纹理过滤的选项。有多种不同的选项可用，但是现在我们只讨论最重要的两种：`GL_NEAREST` 和 `GL_LINEAR`。

GL_NEAREST(Nearest Neighbor Filtering, 邻近过滤) 是一种 OpenGL 默认的纹理过滤方式。当设置为 `GL_NEAREST` 的时候，OpenGL 选择最接近纹理坐标中心点的那个像素。下图你会看到四个像素，加号代表纹理坐标。左上角的纹理像素是距离纹理坐标最近的那个，这样它就会选择这个作为采样颜色：



GL_LINEAR((Bi)linear Filtering, 线性过滤) 它会从纹理坐标的临近纹理像素进行计算，返回一个多个纹理像素的近似值。一个纹理像素距离纹理坐标越近，那么这个纹理像素对最终的采样颜色的影响越大。下面你会看到临近像素返回的混合颜色：



不同的纹理过滤方式有怎样的视觉效果呢？让我们看看当在一个很大的物体上应用一张低解析度的纹理会发生什么吧(纹理被放大了，纹理像素也能看到)：



GL_NEAREST



GL_LINEAR

如上面两张图片所示，`GL_NEAREST` 返回了格子一样的样式，我们能够清晰看到纹理由像素组成，而 `GL_LINEAR` 产生出更平滑的样式，看不出纹理像素。`GL_LINEAR` 是一种更真实的输出，但有些开发者更喜欢 8-bit 风格，所以他们还是用 `GL_NEAREST` 选项。

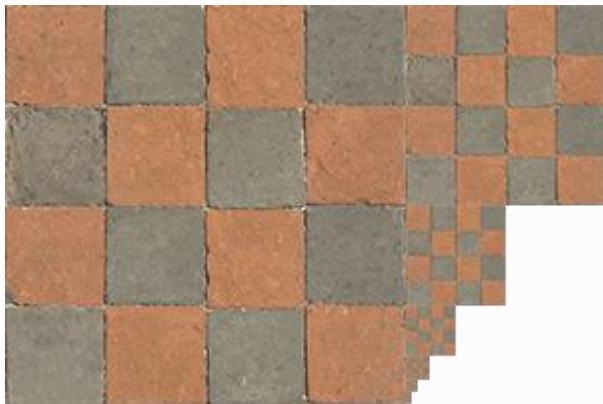
纹理过滤可以为放大和缩小设置不同的选项，这样你可以在纹理被缩小的时候使用最临近过滤，被放大时使用线性过滤。我们必须通过 `glTexParameter` 为放大和缩小指定过滤方式。这段代码看起来和纹理环绕方式(Texture Wrapping)的设置相似：

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

多级渐远纹理(Mipmaps)

想象一下，如果我们在一个有着上千物体的大房间，每个物体上都有纹理。距离观察者远的与距离近的物体的纹理的解析度是相同的。由于远处的物体可能只产生很少的片段，OpenGL 从高解析度纹理中为这些片段获取正确的颜色值就很困难。这是因为它不得不拾为一个纹理跨度很大的片段取纹理颜色。在小物体上这会产生人工感，更不用说在小物体上使用高解析度纹理浪费内存的问题了。

OpenGL 使用一种叫做 **多级渐远纹理(Mipmap)** 的概念解决这个问题，大概来说就是一系列纹理，每个后面的一个纹理是前一个的二分之一。多级渐远纹理背后的思想很简单：距离观察者更远的距离的一段确定的阈值，OpenGL 会把最适合这个距离的物体的不同的多级渐远纹理纹理应用其上。由于距离远，解析度不高也不会被使用者注意到。同时，多级渐远纹理另一加分之处是，执行效率不错。让我们近距离看一看多级渐远纹理纹理：



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，它可以在我们创建完一个纹理后帮我们做所有的多级渐远纹理创建工作。后面的纹理教程中你会看到如何使用它。

OpenGL 渲染的时候，两个不同级别的多级渐远纹理之间会产生不真实感的生硬的边界。就像普通的纹理过滤一样，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。指定不同多级渐远纹理级别之间的过滤方式可以使用下面四种选项代替原来的过滤方式：

过滤方式	描述
<code>GL_NEAREST_MIPMAP_NEAREST</code>	接收最近的多级渐远纹理来匹配像素大小，并使用最临近插值进行纹理采样

过滤方式	描述
GL_LINEAR_MIPMAP_NEAREST	接收最近的多级渐远纹理级别，并使用线性插值采样
GL_NEAREST_MIPMAP_LINEAR	在两个多级渐远纹理之间进行线性插值，通过最邻近插值采样
GL_LINEAR_MIPMAP_LINEAR	在两个相邻的多级渐远纹理进行线性插值，并通过线性插值进行采样

就像纹理过滤一样，前面提到的四种方法也可以使用 `glTexParameter`i 设置过滤方式：

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
               GL_LINEAR_MIPMAP_LINEAR);

glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

常见的错误是，为多级渐远纹理过滤选项设置放大过滤。这样没有任何效果，因为多级渐远纹理主要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，为多级渐远纹理设置放大过滤选项会产生一个 `GL_INVALID_ENUM` 错误。

加载和创建纹理

使用纹理之前要做的第一件事是把它们加载到应用中。纹理图像可能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？一个解决方案是写一个我们自己的某种图像格式加载器比如.PNG，用它来把图像转化为 byte 序列。写自己的图像加载器虽然不难，但是仍然挺烦人的，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。

另一个解决方案是，也许是更好的一种选择，就是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。就像 SOIL 这种库①。

SOIL

SOIL 是 Simple OpenGL Image Library(简易 OpenGL 图像库)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。像大多

数其他库一样，你必须自己生成.lib。你可以使用/projects 文件夹里的解决方案(Solution)文件之一(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；这总是可行的。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)，你也可以使用 CMake 自己生成。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 到你的连接器选项，并在你代码文件的开头加上 #include <SOIL.h>。

下面的纹理部分，我们会使用一张木箱的图片。使用 SOIL 加载图片，我们会使用它的 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 会返回图片的宽度和高度到其中。之后，我们需要图片的宽度和高度来生成纹理。第四个参数指定图片的通道(Channel)数量，但是这里我们只需留 0。最后一个参数告诉 SOIL 如何来加载图片：我们只对图片的 RGB 感兴趣。结果储存为一个大的 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。

```
GLuint texture;  
  
glGenTextures(1, &texture);
```

glGenTextures 函数首先需要输入纹理生成的数量，然后把它们储存在第二个参数的 GLuint 数组中(我们的例子里只有一个 GLuint)，就像其他对象一样，我们需要绑定它，所以下面的纹理命令会配置当前绑定的纹理：

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

现在纹理绑定了，我们可以使用前面载入的图片数据生成纹理了，纹理通过 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：

- 第一个参数指定纹理目标(环境)；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(Target)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)。
- 第二个参数为我们打算创建的纹理指定多级渐远纹理的层级，如果你希望单独手工设置每个多级渐远纹理的层级的话。这里我们填 0 基本级。
- 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，因此我们把纹理储存为 `GL_RGB` 值。
- 第四个和第五个参数设置最终的纹理的宽度和高度。我们加载图像的时候提前储存它们这样我们就能使用相应变量了。下个参数应该一直被设为 0(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，并把它们储存在 `char(byte)`，我们将会传入相应值。
- 最后一个参数是真实的图像数据。

当调用 `glTexImage2D`，当前绑定的纹理对象就会被附加上纹理图像。然而，当前只有基本级别(Base-level)纹理图像加载了，如果要使用多级渐远纹理，我们必须手工设置不同的图像(通过不断把第二个参数增加的方式)或者，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，解绑纹理对象、释放图像的内存很重要。

```
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);
```

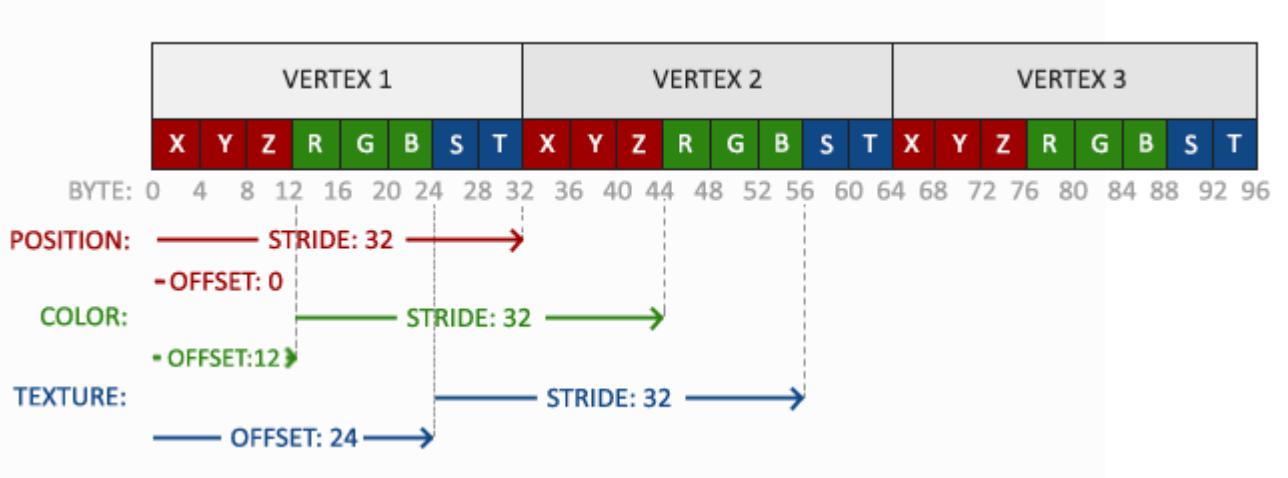
```
//为当前绑定的纹理对象设置环绕、过滤方式  
...  
//加载并生成纹理  
  
int width, height;  
  
unsigned char * image = SOIL_load_image("container.jpg", &width,  
&height, 0, SOIL_LOAD_RGB);  
  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,  
GL_UNSIGNED_BYTE, image);  
  
glGenerateMipmap(GL_TEXTURE_2D);  
  
SOIL_free_image_data(image);  
  
glBindTexture(GL_TEXTURE_2D, 0);
```

应用纹理

后面的部分我们会使用 `glDrawElements` 绘制 [Hello Triangle](#) 教程的最后一部分的矩形。我们需要告知 OpenGL 如何采样纹理，这样我们必须更新顶点纹理坐标数据：

```
GLfloat 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，我们必须通知 OpenGL 新的顶点格式：



```
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat),
```

```
(GLvoid*)(6 * sizeof(GLfloat)));
```

```
 glEnableVertexAttribArray(2);
```

注意，我们必须修正前面两个顶点属性的步长参数为 `8 * sizeof(GLfloat)`。

接着我们需要让顶点着色器把纹理坐标作为一个顶点属性，把坐标传给片段着色器：

```
#version 330 core

layout (location = 0) in vec3 position;

layout (location = 1) in vec3 color;

layout (location = 2) in vec2 texCoord;

out vec3 ourColor;

out vec2 TexCoord;

void main()
{
    gl_Position = vec4(position, 1.0f);
```

```
    ourColor = color;  
  
    TexCoord = texCoord;  
  
}
```

片段着色器应该把输出变量 `TexCoord` 作为输入变量。

片段着色器应该也获取纹理对象，但是我们怎样把纹理对象传给片段着色器？**GLSL** 有一个内建数据类型，供纹理对象使用，叫做采样器(**Sampler**)，它以纹理类型作为后缀，比如 `sampler1D`、`sampler3D`，在我们的例子中它是 `sampler2D`。我们可以简单的声明一个 `uniform sampler2D` 把一个纹理传到片段着色器中，稍后我们把我们的纹理赋值给这个 `uniform`。

```
#version 330 core  
  
in vec3 ourColor;  
  
in vec2 TexCoord;  
  
out vec4 color;  
  
uniform sampler2D ourTexture;  
  
void main()  
{  
    color = texture(ourTexture, TexCoord);  
}
```

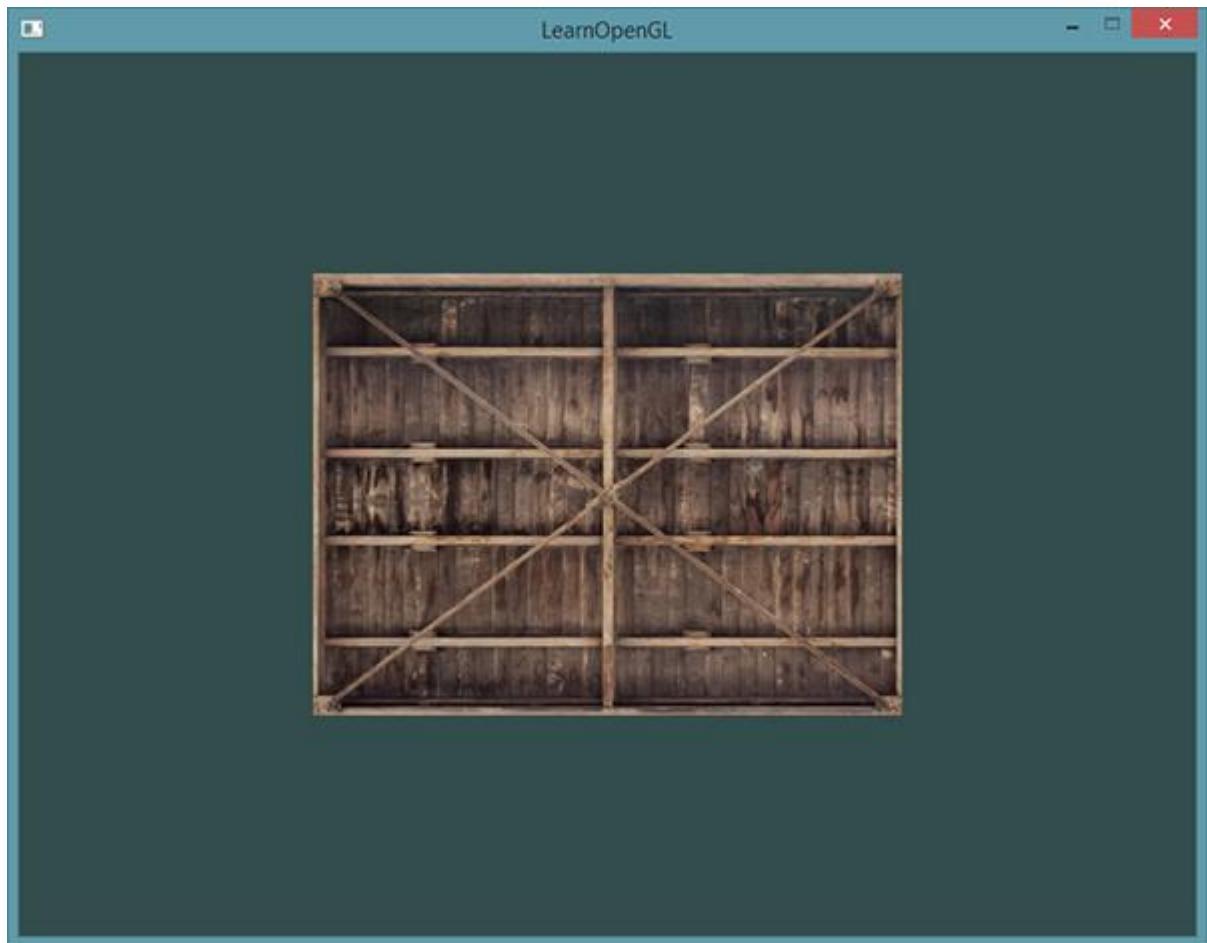
我们使用 **GLSL** 的内建 `texture` 函数来采样纹理的颜色，它第一个参数是纹理采样器，第二个参数是相应的纹理坐标。`texture` 函数使用前面设置的纹理参数对相应颜色值进行采样。这个片段着色器的输出就是纹理的(插值)纹理坐标上的(过滤)颜色。

现在要做的就是在调用 `glDrawElements` 之前绑定纹理，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);  
  
glBindVertexArray(0);
```

如果你跟着这个教程正确的做完了，你会看到下面的图像：

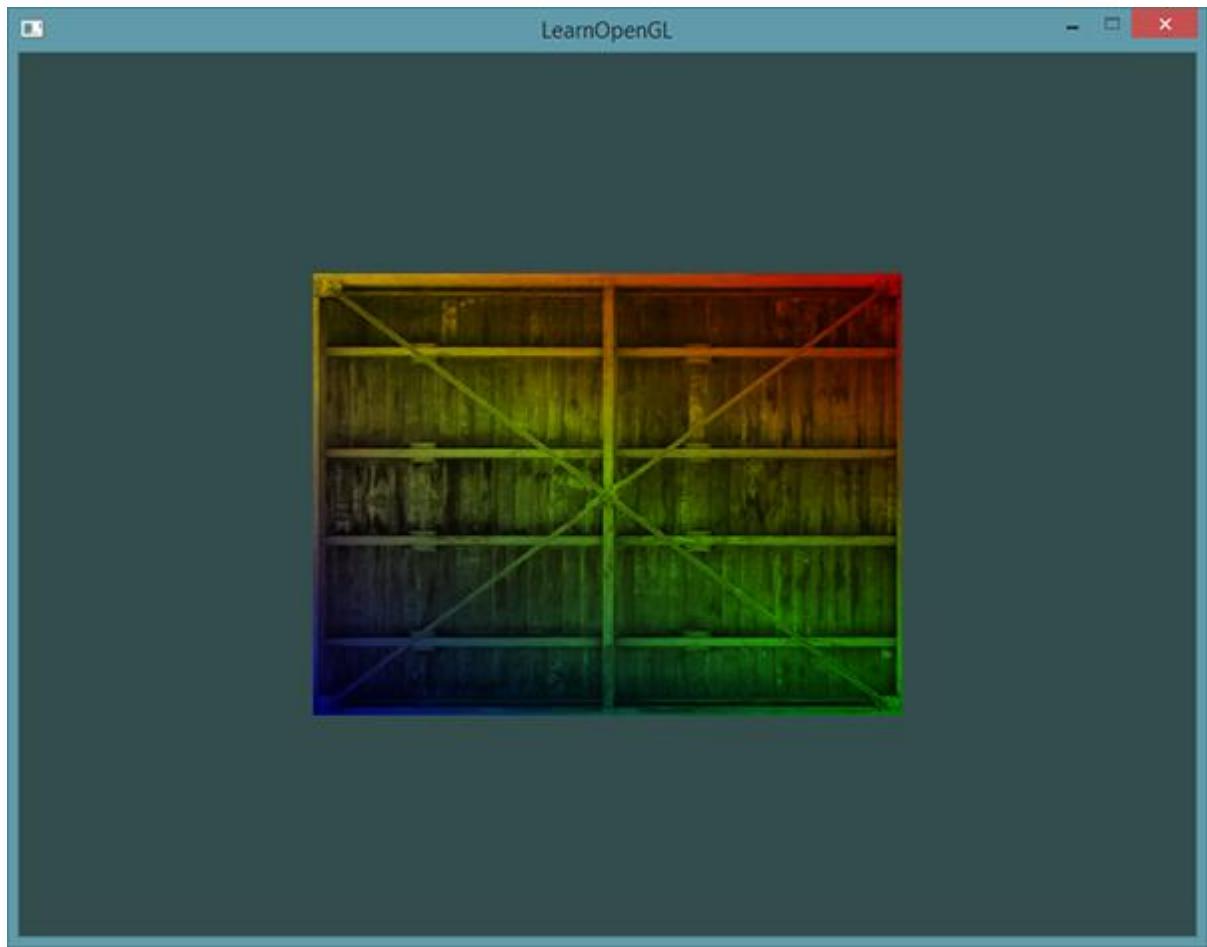


如果你的矩形是全黑或全白的你可能在哪儿做错了什么。检查你的着色器日志，或者尝试对比一下[源码](#)。

我们还可以把纹理颜色和顶点颜色混合，来获得有趣的效果。我们简单的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：

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

最终的效果应该是顶点颜色和纹理颜色的混合色：



这个箱子看起来有点 70 年代迪斯科风格。

纹理单元(Texture Units)

你可能感到奇怪为什么 `sampler2D` 是个 `uniform` 变量，你却不用 `glUniform` 给它赋值，使用 `glUniform1i` 我们就可以给纹理采样器确定一个位置，这样的话我们能够一次在一个片段着色器中设置多纹理。一个纹理的位置通常称为一个纹理单元。一个纹理的默认纹理单元是 0，它是默认激活的纹理单元，所以教程前面部分我们不用给它确定一个位置。

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纹理，只要我们首先激活相应的纹理单元。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 `glBindTexture` 调用函数，会绑定这个纹理到当前激活的纹理单元，纹理单元 `GL_TEXTURE0` 总是默认被激活，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。

Important

OpenGL 至少提供 16 个纹理单元供你使用，也就是说你可以激活 `GL_TEXTURE0` 到 `GL_TEXTURE15`。它们都是顺序定义的，所以我们也可以通过 `GL_TEXTURE0+8` 的方式获得 `GL_TEXTURE8`，这个例子在当我们不得不循环几个纹理的时候变得很有用。

我们仍然要编辑片段着色器来接收另一个采样器。方法现在相对简单了：

```
#version 330 core  
...  
uniform sampler2D ourTexture1;  
uniform sampler2D ourTexture2;  
void main()  
{  
    color = mix(texture(ourTexture1, TexCoord), texture(ourTexture2,  
    TexCoord), 0.2);  
}
```

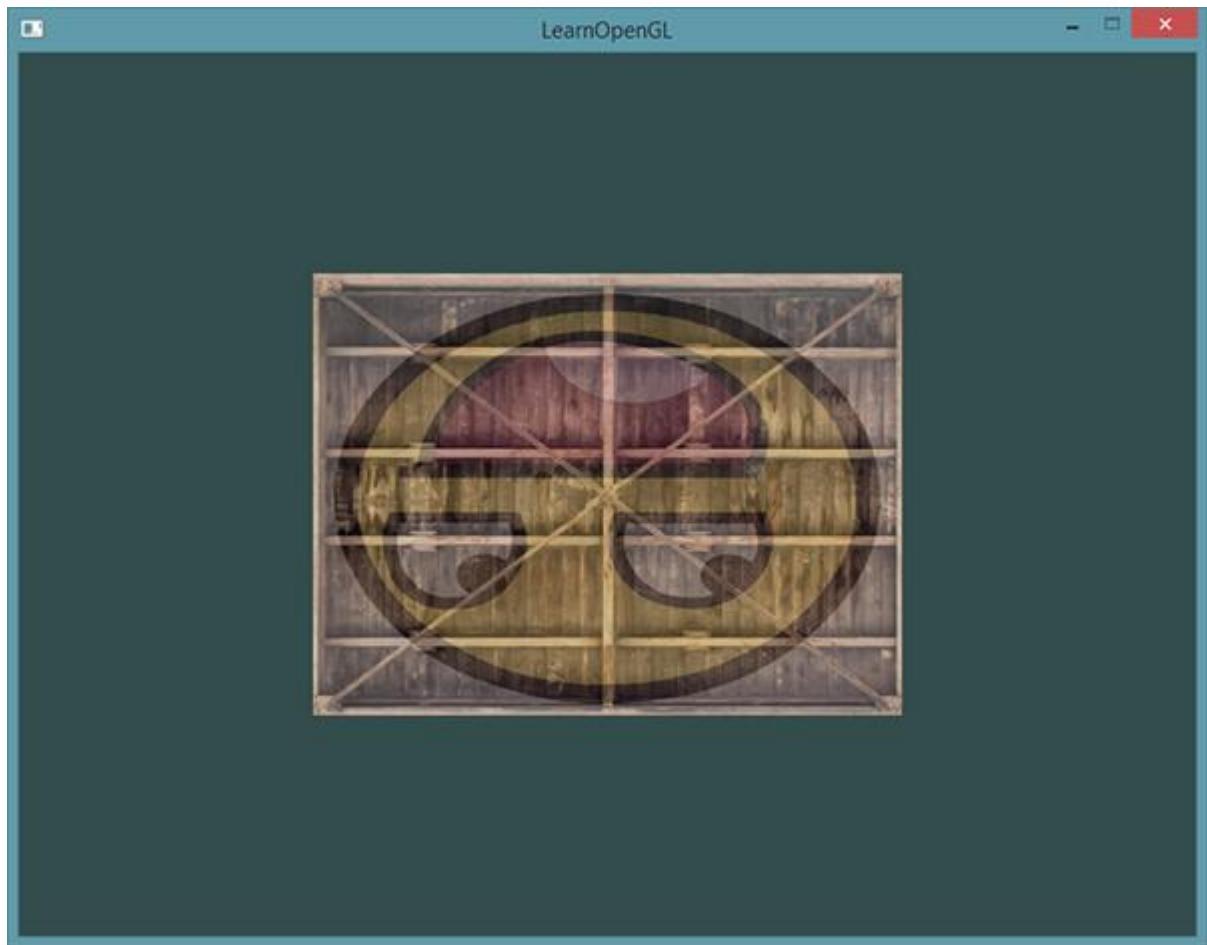
最终输出颜色现在结合了两个纹理查找。GLSL 的内建 `mix` 函数需要两个参数将根据第三个参数为前两者作为输入，并在之间进行线性插值。如果第三个值是 0.0，它返回第一个输入；如果是 1.0，就返回第二个输入值。0.2 返回 80% 的第一个输入颜色和 20% 的第二个输入颜色，返回两个纹理的混合。

我们现在需要载入和创建另一个纹理；我们应该对这些步骤感到熟悉了。确保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 时的表情图片。

为了使用第二个纹理(也包括第一个)，我们必须改变一点渲染流程，先绑定两个纹理到相应的纹理单元，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_IN, 0);  
  
glBindVertexArray(0);
```

注意，我们使用了 `glUniform1i` 设置 uniform 采样器的位置或曰纹理单元。通过 `glUniform1i` 的设置，我们保证了每个 uniform 采样器对应于合适的纹理单元。可以获得下面的结果：



你可能注意到纹理上下颠倒了！这是因为 OpenGL 要求 y 轴 0.0 坐标是在图片的下面的，但是图片通常 y 轴 0.0 坐标在上面。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 有一个叫做

`SOIL_load_OGL_texture` 函数可以使用一个叫做 `SOIL_FLAG_INVERT_Y` 的标记加载和生成纹理，它用来解决我们的问题。不过这个函数在现代 OpenGL 中的这个特性失效了，所以现在我们必须坚持使用 `SOIL_load_image`，自己做纹理生成。

所以修复我们的小问题，有两个选择：

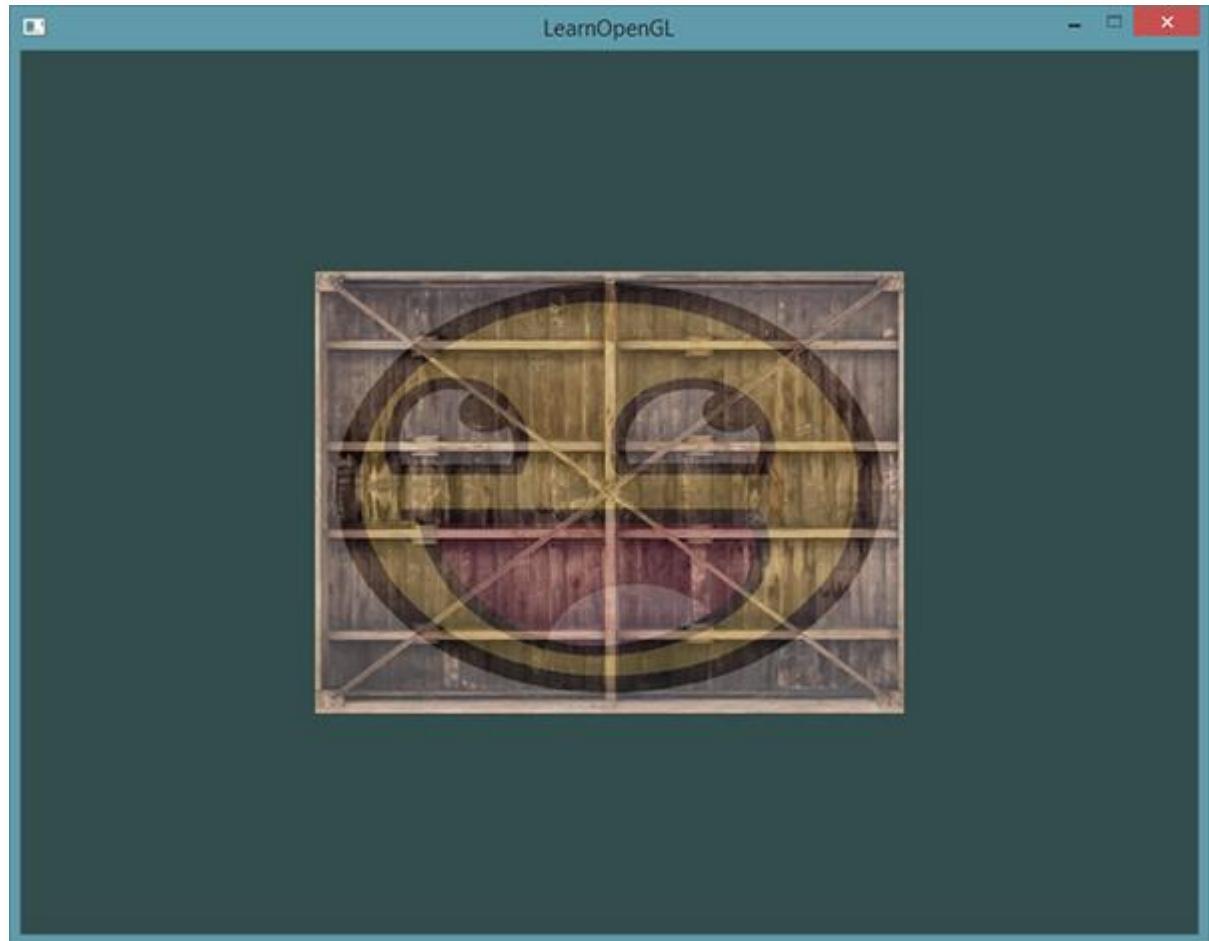
1. 我们切换顶点数据的纹理坐标，翻转 `y` 值(用 1 减去 y 坐标)。
2. 我们可以编辑顶点着色器来翻转 `y` 坐标，自动替换 `TexCoord` 赋值：
`TexCoord = vec2(texCoord.x, 1 - texCoord.y);`

Attention

上面提供的解决方案仅仅通过一些 hacks 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

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：



如果你看到了图片上的笑脸容器，你就做对了。你可以对比[程序源代码](#)，以及[顶点着色器](#)和[片段着色器](#)。

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：

- 使用片段着色器仅对笑脸图案进行翻转，[参考解答](#)
- 尝试用不同的纹理环绕方式，并将纹理坐标的范围设定为从 `0.0f` 到 `2.0f` 而不是原来的 `0.0f` 到 `1.0f`，在木箱子的角落放置 4 个笑脸：[参考解答](#), [结果](#)。记得一定要试试其他的环绕方式。

- 尝试在矩形范围内只显示纹理图的中间一部分，并通过修改纹理坐标来设置显示效果。尝试使用 `GL_NEAREST` 的纹理过滤方式让图像显示得更清晰：[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容器的大小和笑脸是否可见：[参考解答](#), [片段着色器](#)。

变换(Transformations)

原文	Transformations
作者	JoeyDeVries
翻译	Django
校对	Meow J, BLumia

尽管我们现在已经知道了如何创建一个物体、着色、加入纹理从而给它们一些细节的表现，但是它们仍然还是不够有趣，因为它们都还是静态的物体。我们可以尝试着在每一帧改变物体的顶点并且重设缓冲区从而使他们移动，但这太繁琐了，而且会消耗很多的处理时间。然而，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(Matrix)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。

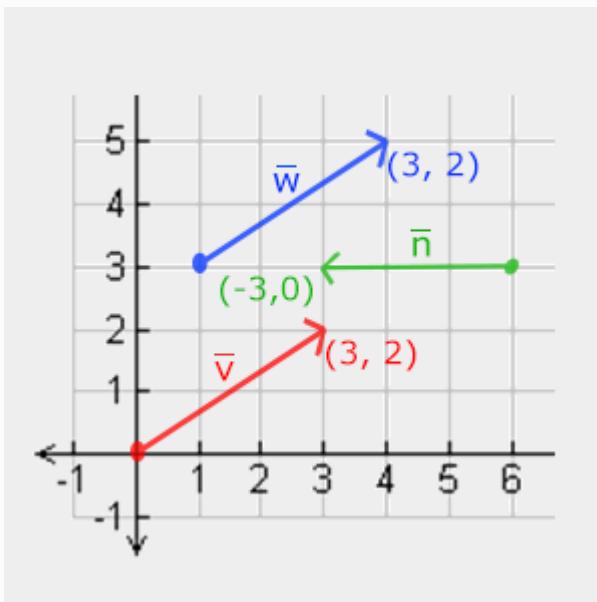
为了深入了解变换，我们首先要在讨论矩阵之前了解一点向量(Vector)。这一节的目标是让你拥有将来需要的最基础的数学背景知识。如果你发现这节十分困难，尽量尝试去理解它们，当你以后需要它们的时候回过头来复习这些概念。

向量(Vector)

向量最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。你可以发现，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。你仍然可以把这些 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 \bar{v} 和 \bar{w} 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 \bar{v} 。当用在公式中时它们通常是这样的：

$$\bar{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

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。我们通常设定这个方向的原点为(0,0,0)，然后指向对应坐标的点，使其变为位置向量(Position Vector)来表示(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)。位置向量(3, 5)的在图像中起点是(0, 0)，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知道了)。

向量与标量运算(Scalar Vector Operations)

标量(**Scalar**)只是一个数字(或者说是仅有一个分量的矢量)。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:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x = \begin{pmatrix} 1 + x \\ 2 + x \\ 3 + x \end{pmatrix}$$

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

向量取反(Vector Negation)

对一个向量取反会将其方向逆转。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):

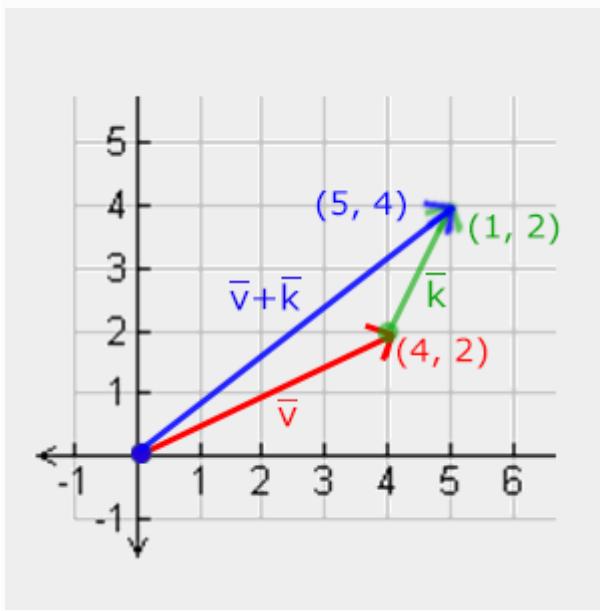
$$-\bar{v} = - \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -v_x \\ -v_y \\ -v_z \end{pmatrix}$$

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:

$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + \bar{k} = \begin{pmatrix} 1 + 4 \\ 2 + 5 \\ 3 + 6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

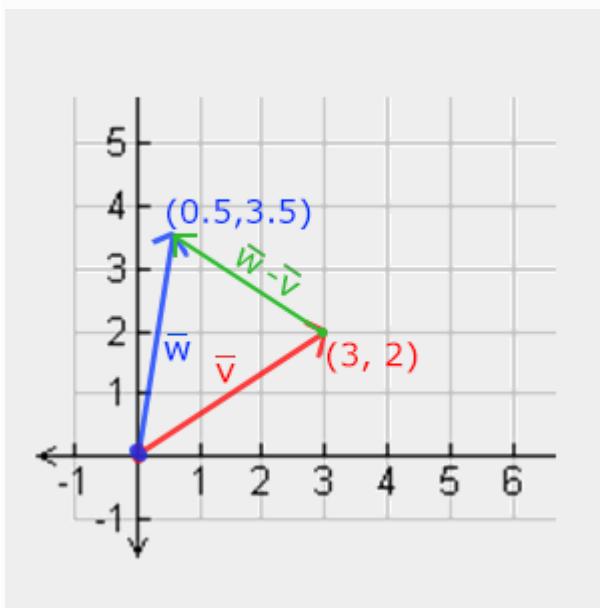
向量 $v = (4, 2)$ 和 $k = (1, 2)$ 直观地表示为:



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数：

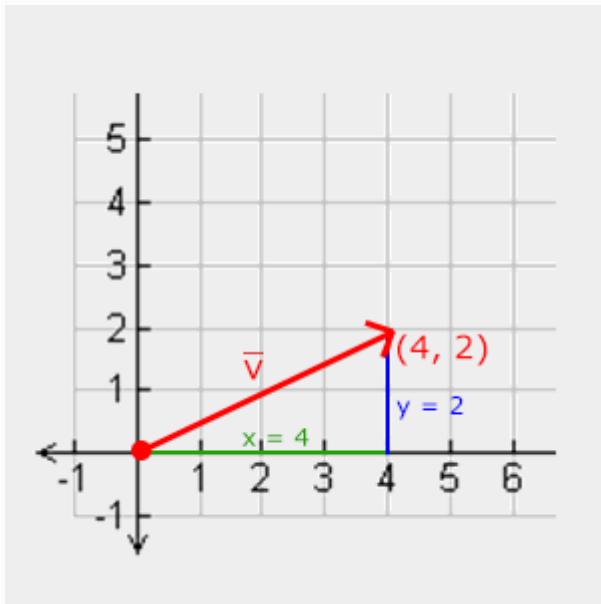
$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + -\bar{k} = \begin{pmatrix} 1 + (-4) \\ 2 + (-5) \\ 3 + (-6) \end{pmatrix} = \begin{pmatrix} -3 \\ -3 \\ -3 \end{pmatrix}$$

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(Length)

我们使用勾股定理(**Pythagoras Theorem**)来获取向量的长度/大小。如果你把向量的 x 与 y 分量画出来，该向量会形成一个以 x 与 y 分量为边的三角形：



因为两条边(x 和 y)是已知的，而且我们希望知道斜边 \bar{v} 的长度，所以我们可以通过勾股定理来计算出它：

$$||\bar{v}|| = \sqrt{x^2 + y^2}$$

$||\bar{v}||$ 表示向量 \bar{v} 的大小，我们也可以很容易加上 z^2 把这个公式拓展到三维空间

例子中向量 $(4, 2)$ 的长度等于：

$$||\bar{v}|| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} = 4.47$$

结果是 4.47。

有一个特殊类型向量叫做**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 \hat{n} :

$$\hat{n} = \frac{\bar{v}}{||\bar{v}||}$$

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 \wedge 样子的记号，并且它会变得很有用，特别是在我们只关心方向不关系长度的时候(如果我们改变向量的长度，它的方向并不会改变)。

向量相乘(Vector-vector Multiplication)

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，但是有两种特定情境，当需要乘法时我们可以从中选择：一个是**点乘(Dot Product)**，记作 $\bar{v} \cdot \bar{k}$ ，另一个是**叉乘(Cross Product)**，记作 $\bar{v} \times \bar{k}$ 。

点乘(Dot Product)

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。听起来有点费解，先看一下公式：

$$\bar{v} \cdot \bar{k} = ||\bar{v}|| \cdot ||\bar{k}|| \cdot \cos \theta$$

它们之间的夹角我们记作 θ 。为什么这很有用？想象如果 \bar{v} 和 \bar{k} 都是单位向量，它们的长度等于 1。公式会有效简化成：

$$\bar{v} \cdot \bar{k} = 1 \cdot 1 \cdot \cos \theta = \cos \theta$$

现在点乘只和两个向量的角度有关。你也许记得当 90 度的余弦是 0，0 度的余弦是 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)。你可能想要了解更多的关于正弦或余弦的知识，我推荐你看[可汗学院](#)的基础三角学视频。

Important

你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大小之积，得到的结果就是夹角的余弦值，即 $\cos\theta$ 。

译注：通过上面点乘定义式可推出：

$$\cos \theta = \frac{\bar{v} \cdot \bar{k}}{||\bar{v}|| \cdot ||\bar{k}||}$$

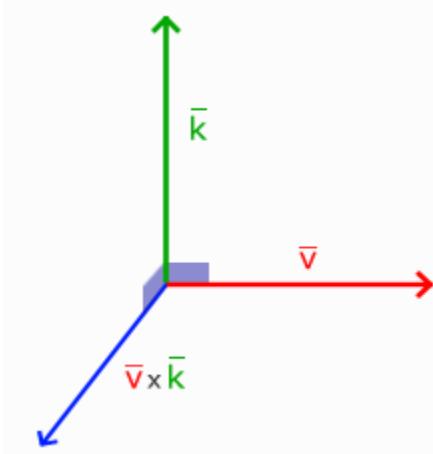
所以，我们如何计算点乘？点乘是按分量逐个相乘，然后再把结果相加。两个单位向量点乘就像这样(你可以用两个长度为 1 的验证)：

$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

计算两个单位余弦的角度，我们使用反余弦 \cos^{-1} ，结果是 143.1 度。现在我们很快就计算出了两个向量的角度。点乘在计算光照的时候会很有用。

叉乘(Cross Product)

叉乘只在 3D 空间有定义，它需要两个不平行向量作为输入，生成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，那么叉乘的结果将会返回 3 个互相正交的向量。接下来的教程中，这很有用。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，会觉得叉乘很反直觉，所以最好记住公式，就没问题(记不住也没问题)。下面你会看到两个正交向量 \mathbf{A} 和 \mathbf{B} 叉乘结果：

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$

就像你所看到的，看起来毫无头绪。可如果你这么做了，你会得到第三个向量，它正交于你的输入向量。

矩阵(Matrix)

现在我们已经讨论了向量的全部内容，是时候看看矩阵了！矩阵简单说是一个矩形的数字、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：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

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)。

关于矩阵基本也就是这些了，它就是矩形数学表达式阵列。矩阵也有非常漂亮的数学属性，就跟向量一样。矩阵有几个运算，叫做：矩阵加法、减法和乘法。

矩阵的加减

矩阵与标量的加减如下所示：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 3 = \begin{bmatrix} 1+3 & 2+3 \\ 3+3 & 4+3 \end{bmatrix} = \begin{bmatrix} 4 & 5 \\ 6 & 7 \end{bmatrix}$$

标量值要加到矩阵的每一个元素上。矩阵与标量的减法也是同样的：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} - 3 = \begin{bmatrix} 1-3 & 2-3 \\ 3-3 & 4-3 \end{bmatrix} = \begin{bmatrix} -2 & -1 \\ 0 & 1 \end{bmatrix}$$

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在同维度的矩阵中是有定义的。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 矩阵是怎样加减的：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

同样的法则也适用于减法：

$$\begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-2 & 2-4 \\ 1-0 & 6-1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 1 & 5 \end{bmatrix}$$

矩阵的数乘(Matrix-scalar Products)

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：

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

现在我们也就能明白为什么一个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)。前面的例子里，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(Matrix-matrix Multiplication)

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)， $A \cdot B \neq B \cdot A$ 。

我们先看一个两个 2×2 矩阵相乘的例子：

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

现在你可能会在想了：我勒个去，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：

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

我们先把左侧矩阵的行和右侧矩阵的列拿出来。这些我们挑出来行和列决定着作为结果的 2×2 矩阵的输出值。如果我们拿出来的是左矩阵的第一行，最终的值就会出现在作为结果的矩阵的第一行，如果我们拿出来的是右矩阵的第一列，最终值会出现在作为结果的矩阵的第一列。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操作我们就无法完成了！

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

如果你在脑子里想象出乘法有困难别担心。用笔写下来，如果遇到困难回头看这页的内容。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矩阵与矩阵乘法的讨论。尝试使用颜色来让这个公式更容易理解。作为一个有用的练习，你可以自己回答这个乘法问题然后对比你的结果和图中的这个(如果你用笔计算，你很快就能掌握它们)。

$$\begin{bmatrix} 4 & 2 & 0 \\ 0 & 8 & 1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 & 1 \\ 2 & 0 & 4 \\ 9 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 4 \cdot 4 + 2 \cdot 2 + 0 \cdot 9 \\ 0 \cdot 4 + 8 \cdot 2 + 1 \cdot 9 \\ 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 2 \end{bmatrix} = \begin{bmatrix} 20 & 8 \\ 25 & 4 \\ 2 & 0 \end{bmatrix}$$

就像你所看到的那样，矩阵与矩阵相乘复杂而容易犯错(这就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，对矩阵的数学属性感到好奇，我强烈推荐你看看[可汗学院](#)的矩阵内容视频。

不管怎样，反正现在我们知道如何进行矩阵相乘了，我们可以开始了解好东西了。

矩阵与向量相乘

到目前，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、颜色和纹理坐标。让我们进到兔子洞更深处：向量基本上就是一个 $N \times 1$ 矩阵， N 是

向量分量的个数(也叫 **N 维(N-dimensional)** 向量)。如果你仔细思考这个问题，会很有意思。向量和矩阵一样都是一个数字序列，但是它只有 1 列。所以，这个新信息能如何帮助我们？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，因为我们的矩阵的列数等于向量的行数，所以它们就能相乘。

但是为什么我们关心矩阵是否能够乘以一个向量？有很多有意思的 2D/3D 变换本质上都是矩阵，而矩阵与我们的向量相乘会变换我们的向量。假如你仍然有些困惑，我们看一些例子，你很快就能明白了。

单位矩阵(Identity Matrix)

在 OpenGL 中，因为有一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因为每一个向量都有 4 个分量的。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。就像你看到的，这个变换矩阵使一个向量完全不变：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

向量看起来完全没动。从乘法法则来看很明显：第一个结果分量是矩阵的第一行的每个对应分量乘以向量的每一个分量。因为每行的分量除了第一个都是 0，可得: $1 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 0 \cdot 4 = 1$ ，这对向量的其他 3 个分量同样适用。

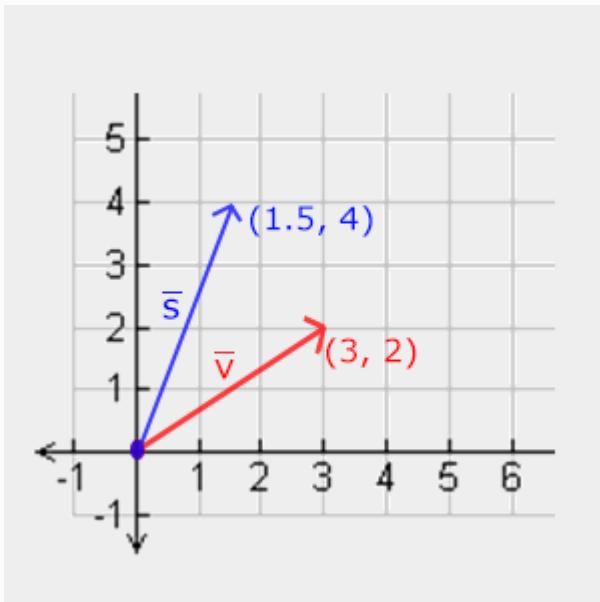
Important

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，这就是一个对证明定理、解线性方程非常有用的矩阵。

缩放(Scaling)

当我们对一个向量进行缩放的时候就是对向量的长度进行缩放，而它的方向保持不变。如果我们进行 2 或 3 维操作，那么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)。

我们可以尝试去缩放向量 $\bar{v} = (3, 2)$ 。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；我们可以沿着 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 $(0.5, 2)$ 所获得的 \bar{s} 是什么样的：



记住，OpenGL通常是在3D空间操作的，对于2D的情况我们可以把z轴缩放1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。如果每个轴的缩放都一样那么就叫**均匀缩放(Uniform Scale)**。

我们下面设置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，每个对角线元素乘以对应的向量分量。如果我们把1变为3会怎样？这种情况，我们就把向量的每个分量乘以3了，这事实上就把向量缩放3。如果我们把缩放变量表示为 (S_1, S_2, S_3) 我们可以为任意向量 (x, y, z) 定义一个缩放矩阵：

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

注意，第四个缩放的向量仍然是1，因为不会缩放3D空间中的w分量。w分量另有其他用途，在后面我们会看到。

平移(Translation)

平移(Translation)是在原来向量的基础上加上另一个的向量从而获得一个在不同位置的新向量的过程，这样就基于平移向量移动(Move)了向量。我们已经讨论了向量加法，所以你应该不会陌生。

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，对于平移来说它们是第四列最上面的3个值。如果我们把缩放向量表示为

(T_x , T_y , T_z)我们就能把平移矩阵定义为：

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

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

Important

齐次坐标(Homogeneous coordinates)

向量的w分量也叫齐次坐标。想要从齐次坐标得到3D坐标，我们可以把x、y和z坐标除以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向量上进行平移(如果没有w分量我们是不能平移向量的)，下一章我们会用w值创建3D图像。

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，这个向量就不能平移(译注：这也就是我们说的不能平移一个方向)。有了平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(Rotation)

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](#)视频。

首先我们来定义一个向量的旋转到底是什么。2D 或 3D 空间中点的旋转用角(**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。

Important

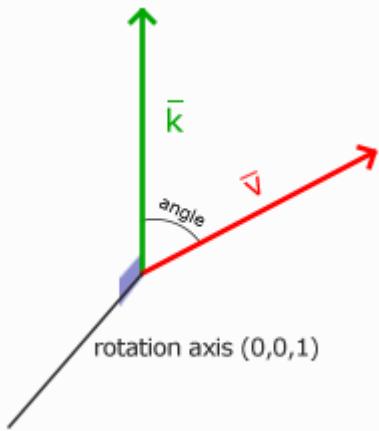
大多数旋转函数需要用弧度制的角，但是角度制的角也可以很容易地转化为弧度制：

- 弧度转角度：角度 = 弧度 * (180.0f / PI)
- 角度转弧度：弧度 = 角度 * (PI / 180.0f)

PI 约等于 3.14159265359。

转半圈会向右旋转 $360/2 = 180$ 度，向右旋转 $1/5$ 圈表示向右旋转 $360/5 = 72$ 度。

这表明 2D 空间的向量 \bar{v} 是由 \bar{k} 向右旋转 72 度得到的：



在 3D 空间中旋转需要一个角和一个**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描述，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。比如 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)。

使用三角学就能把一个向量变换为一个经过旋转特定角度的新向量。这通常是使用一系列正弦和余弦各种巧妙的组合得到的(一般简称 \sin 和 \cos)。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，这个角度表示为 θ :

沿 x 轴旋转:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

沿 y 轴旋转:

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x + \sin \theta \cdot z \\ y \\ -\sin \theta \cdot x + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

沿 z 轴旋转:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x - \sin \theta \cdot y \\ \sin \theta \cdot x + \cos \theta \cdot y \\ z \\ 1 \end{pmatrix}$$

利用旋转矩阵我们可以把我们的位置向量(Position Vectors)沿一个或多个轴进行旋转。也可以把多个矩阵结合起来，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)。我们不会讨论它的细节，但是一个更好的解决方案是沿着任意轴比如(0.662, 0.2, 0.7222)(注意，这是个单位向量)旋转，而不是使用一系列

旋转矩阵的组合。这样一个(超级麻烦)的矩阵是存在的, 下面(R_x , R_y , R_z)代表任意旋转轴:

$$\begin{bmatrix} \cos \theta + R_x^2(1 - \cos \theta) & R_x R_y (1 - \cos \theta) - R_z \sin \theta \\ R_y R_x (1 - \cos \theta) + R_z \sin \theta & \cos \theta + R_y^2(1 - \cos \theta) \\ R_z R_x (1 - \cos \theta) - R_y \sin \theta & R_z R_y (1 - \cos \theta) + R_x \sin \theta \\ 0 & 0 \end{bmatrix}$$

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)**, 它不仅安全, 而且计算更加友好。有关四元数会在后面的教程中讨论。

矩阵的组合

使用矩阵变换的真正力量在于, 根据矩阵之前的乘法, 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多个变换相结合而成的变换矩阵。我们有一个顶点(x , y , z), 我们希望将其缩放 2 倍, 然后用位移(1, 2, 3)来平移它。我们需要一个平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:

$$Trans.Scale = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

注意, 当矩阵相乘时我们先写平移再写缩放变换的。矩阵乘法是不可交换的, 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乘以向量的, 所以你应该从右向左读这个乘法。我们建议您在组合矩阵时, 先进行缩放操作, 然后是旋转, 最后才是平移, 否则它们会(消极地)互相影响。比如, 如果你先平移然后缩放, 平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)!

将我们的矢量左乘最终的变换矩阵会得到以下结果：

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 2x + 1 \\ 2y + 2 \\ 2z + 3 \\ 1 \end{bmatrix}$$

不错！向量先缩放 2 倍，然后平移了(1, 2, 3)个单位。

实践

现在我们已经解释了所有变换背后的理论，是时候将这些知识利用起来了。OpenGL 没有任何自带的矩阵和向量形式，所以我们必须自己定义数学类和方法。在这个教程中我们更愿意抽象所有的数学细节，使用已经做好了的数学库。幸运的是有个使用简单的专门为 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合适的头文件就行了；不用链接和编译。GLM 可以从他们的[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。



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 $(1, 0, 0)$ 平移 $(1, 1, 0)$ 个单位(注意，我们把它定义为一个 `glm::vec4` 类型的值，其中齐次坐标我们设定为 1.0):

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);  
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。接下来我们定义一个 `mat4` 类型的 `trans`，默认是 4×4 单位矩阵。接下来我们创建一个变换矩阵，我们是把单位矩阵和一个平移向量传递给 `glm::translate` 函数来完成这个工作的(然后用给定的矩阵乘以平移矩阵就能获得最后需要的矩阵)。

之后我们把向量乘以平移矩阵并且输出最后的结果。如果我们仍然记得平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，所以这个平移矩阵是正确的。

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，使它变成原来的二分之一。我们先来创建变换矩阵：

```
glm::mat4 trans;  
trans = glm::rotate(trans, 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缩放到 0.5 倍，然后沿 Z 轴旋转 90 度。注意有纹理的那面矩形是在 XY 平面上的，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。

Attention

有些 GLM 版本接收的是弧度而不是角度，这种情况下你可以用 `glm::radians(90.0f)` 将角度转换为弧度。

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 里的 `mat4` 类型。所以我们改写顶点着色器来接收一个 `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 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 TexCoord;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(position, 1.0f);
    ourColor = color;
    TexCoord = vec2(texCoord.x, 1.0 - texCoord.y);
}
```

Attention

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(比如标量-矩阵乘法，矩阵-向量乘法和矩阵-矩阵乘法)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发生了什么的。

在把位置向量传给 `gl_Position` 之前，我们添加一个 `uniform`，并且用变换矩阵乘以它。我们的箱子现在应该是原来的二分之一大小并旋转了 90 度(向左倾斜)。

当然，我们仍需要把变换矩阵传递给着色器：

```
GLuint transformLoc = glGetUniformLocation(ourShader.Program,  
"transform");  
  
glUniformMatrix4fv(transformLoc, 1, GL_FALSE,  
glm::value_ptr(trans));
```

我们首先请求 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 的地址 (Location)。第二个参数告诉 OpenGL 我们将要发送多少个矩阵，目前是 1。第三个参数询问我们是否希望对我们的矩阵进行置换(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以列为主顺序的(**Column-major Ordering**)布局。GLM 已经是用以列为主顺序定义了它的矩阵，所以并不需要置换矩阵，我们填 `GL_FALSE`、最后一个参数是实际的矩阵数据，但是 GLM 并不是把它们的矩阵储存为 OpenGL 所希望的那种，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！我们的箱子向左侧倾斜，是原来的二分之一大小，看来变换成功了。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左下角。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，因为它需要在每一次渲染迭代中被更新。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,(GLfloat)glfwGetTime() * 50.0f,  
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，从而保证我们能够更新旋转矩阵。这也就意味着我们不得不在每次迭代中重新创建变换矩阵。通常在渲染场景的时候，我们也会有多个在每次渲染迭代中都用新的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平移到屏幕的右下角。记住，实际的变换顺序应该从下向上阅读：尽管在代码中我们先平移再旋转，实际的变换却是先应用旋转然后平移的。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的并不简单。只有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！一个平移过的箱子，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。我们可以定义一个无限数量的变换，把它们组合为一个单独的矩阵，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力气，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](#)和[顶点、片段](#)着色器。

下个教程中，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！

练习

- 使用应用在箱子上的最后的变换，尝试将其改变成先旋转，后平移。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](#)
- 尝试着再次调用 `glDrawElements` 画出第二个箱子，但是只能使用变换将其摆放在不同的位置。保证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)。使用 `sin` 函数在这里会很有用；注意使用 `sin` 函数取到负值时会导致物体被翻转：[参考解答](#)

坐标系统(Coordinate System)

原文	Coordinate Systems
作者	JoeyDeVries
翻译	linkoln

原文

[Coordinate Systems](#)

校对

Geequlim, Meow J, [BLumia](#)

在上一个教程中，我们学习了如何有效地利用矩阵变换来对所有顶点进行转换。**OpenGL** 希望在所有顶点着色器运行后，所有我们可见的顶点都变为标准化设备坐标(**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转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**)，再将他们转换为屏幕上的二维坐标或像素。

将坐标转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，也就是类似于流水线那样子，实现的，在流水线里面我们在将对象转换到屏幕空间之前会先将其转换到多个坐标系统。将对象的坐标转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进行一些操作或运算更加方便和容易，这一点很快将会变得很明显。对我们来说比较重要的总共有 5 个不同的坐标系统：

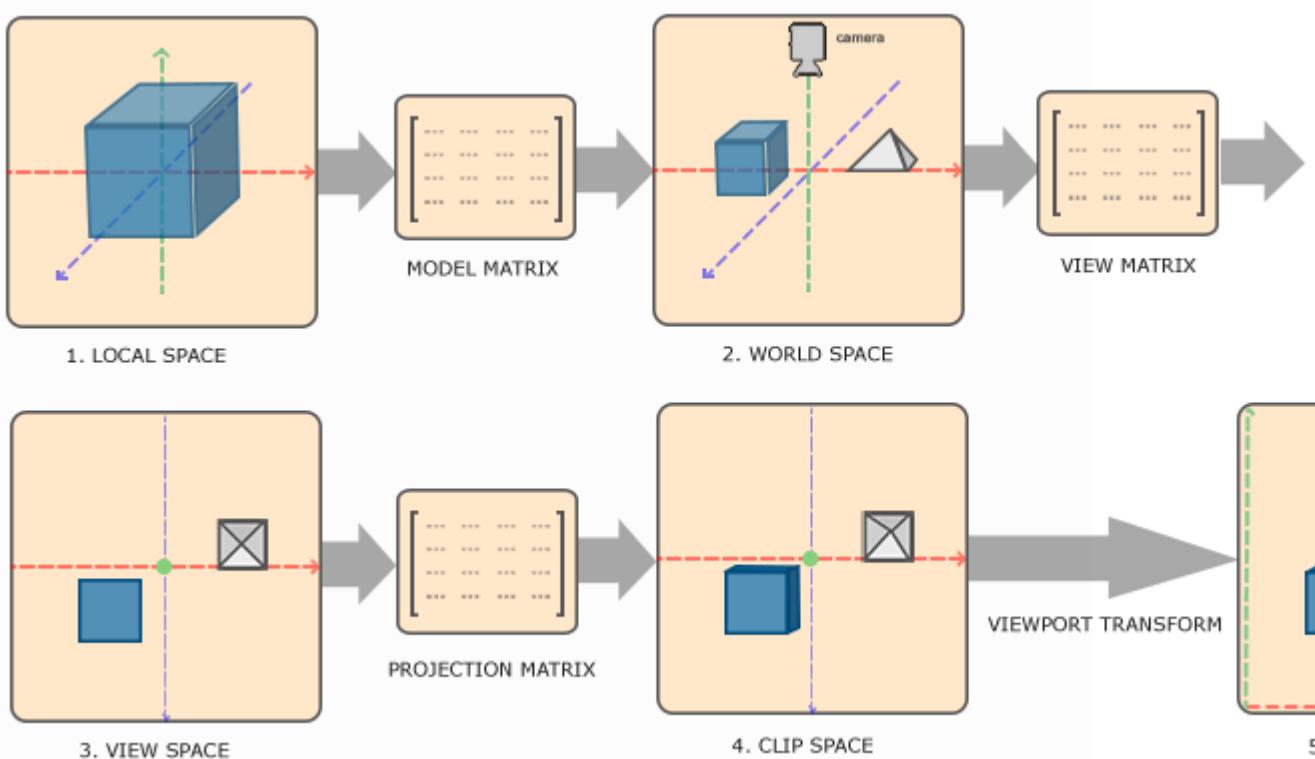
- 局部空间(**Local Space**，或者称为物体空间(**Object Space**))
- 世界空间(**World Space**)
- 观察空间(**View Space**，或者称为视觉空间(**Eye Space**))
- 裁剪空间(**Clip Space**)
- 屏幕空间(**Screen Space**)

这些就是我们将所有顶点转换为片段之前，顶点需要处于的不同的状态。

你现在可能对什么是空间或坐标系到底是什么感到困惑，所以接下来我们将会通过展示完整的图片来解释每一个坐标系实际做了什么。

整体概述

为了将坐标从一个坐标系转换到另一个坐标系，我们需要用到几个转换矩阵，最重要的几个分别是**模型(Model)**、**视图(View)**、**投影(Projection)**三个矩阵。首先，顶点坐标开始于**局部空间(Local Space)**，称为**局部坐标(Local Coordinate)**，然后经过**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)**结束。下面的图示显示了整个流程及各个转换过程做了什么：



1. 局部坐标是对象相对于局部原点的坐标；也是对象开始的坐标。
2. 将局部坐标转换为世界坐标，世界坐标是作为一个更大空间范围的坐标系统。这些坐标是相对于世界的原点的。
3. 接下来我们将世界坐标转换为观察坐标，观察坐标是指以摄像机或观察者的角度观察的坐标。
4. 在将坐标处理到观察空间之后，我们需要将其投影到裁剪坐标。裁剪坐标是处理-1.0 到 1.0 范围内并判断哪些顶点将会出现在屏幕上。
5. 最后，我们需要将裁剪坐标转换为屏幕坐标，我们将这一过程成为**视口变换**
(Viewport Transform)。视口变换将位于-1.0 到 1.0 范围的坐标转换到由
`glViewport` 函数所定义的坐标范围内。最后转换的坐标将会送到光栅器，由光栅器将其转化为片段。

你可能了解了每个单独的坐标空间的作用。我们之所以将顶点转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，当修改对象时，如果在局部空间中则是有意义的；当对对象做相对于其它对象的位置的操作时，在世界坐标系中则是有意义的；等等这些。如果我们愿意，本可以定义一个直接从局部空间到裁剪空间的转换矩阵，但那样会失去灵活性。接下来我们将要更仔细地讨论各个坐标系。

局部空间(Local Space)

局部空间是指对象所在的坐标空间，例如，对象最开始所在的地方。想象你在一个模型建造软件(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 $(0, 0, 0)$ ，即使有可能在最后的应用中位于完全不同的另外一个位置。甚至有可能你创建的所有模型都以 $(0, 0, 0)$ 为初始位置，然而他们会在世界的不同位置。则你的模型的所有顶点都是在**局部空间**：他们相对于你的对象来说都是局部的。

我们一直使用的那个箱子的坐标范围为-0.5 到 0.5，设定 $(0, 0)$ 为它的原点。这些都是局部坐标。

世界空间(World Space)

如果我们想将我们所有的对象导入到程序当中，它们有可能会全挤在世界的原点上 $(0, 0, 0)$ ，然而这并不是我们想要的结果。我们想为每一个对象定义一个位置，从而使对象位于更大的世界当中。世界空间中的坐标就如它们听起来那样：是指顶点相对于(游戏)世界的坐标。物体变换到的最终空间就是世界坐标系，并且你会想让这些物体分散开来摆放(从而显得更真实)。对象的坐标将会从局部坐标转换到世界坐标；该转换是由**模型矩阵(Model Matrix)**实现的。

模型矩阵是一种转换矩阵，它能通过对对象进行平移、缩放、旋转来将它置于它本应该在的位置或方向。你可以想象一下，我们需要转换一栋房子，通过将它缩小(因为它在局部坐标系中显得太大了)，将它往郊区的方向平移，然后沿着 y 轴往坐标旋转。经过这样的变换之后，它将恰好能够与邻居的房子重合。你能够想到上一节讲到的利用模型矩阵将各个箱子放置到这个屏幕上；我们能够将箱子中的局部坐标转换为观察坐标或世界坐标。

观察空间(View Space)

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**)。观察空间就是将对象的世界空间的坐标转换为观察者视野前面的坐标。因此观察空间就是从摄像机的角度观察到的空间。而这通常是由一系列的平移和旋转的组合来平移和旋转场景从而使得特定的对象被转换到摄像机前面。这些组合在一起的转换通常存储在一个**观察矩阵(View Matrix)**里，用来将世界坐标转换到观察空间。在下一个教程我们将广泛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(Clip Space)

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给定的范围内, 且任何在这个范围之外的点都应该被裁剪掉(Clipped)。被裁剪掉的坐标就被忽略了, 所以剩下的坐标就将变为屏幕上可见的片段。这也就是裁剪空间名字的由来。

因为将所有可见的坐标都放置在-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)并将它转换回标准化设备坐标系, 就像 OpenGL 期望它做的那样。

为了将顶点坐标从观察空间转换到裁剪空间, 我们需要定义一个投影矩阵(**Projection Matrix**), 它指定了坐标的范围, 例如, 每个维度都是从-1000 到 1000。投影矩阵接着会将在它指定的范围内的坐标转换到标准化设备坐标系中(-1.0, 1.0)。所有在范围外的坐标在-1.0 到 1.0 之间都不会被绘制出来并且会被裁剪。在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, 随后被转化为在标准化设备坐标中坐标值大于 1.0 的值并且被裁剪掉。

Important

如果只是片段的一部分例如三角形, 超出了裁剪体积(Clipping Volume), 则 OpenGL 会重新构建三角形以使一个或多个三角形能适应在裁剪范围内。

由投影矩阵创建的观察区域(Viewing Box)被称为平截头体(Frustum), 且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。将一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)被称为投影(**Projection**), 因为使用投影矩阵能将 3 维坐标投影(**Project**)到很容易映射的 2D 标准化设备坐标系中。

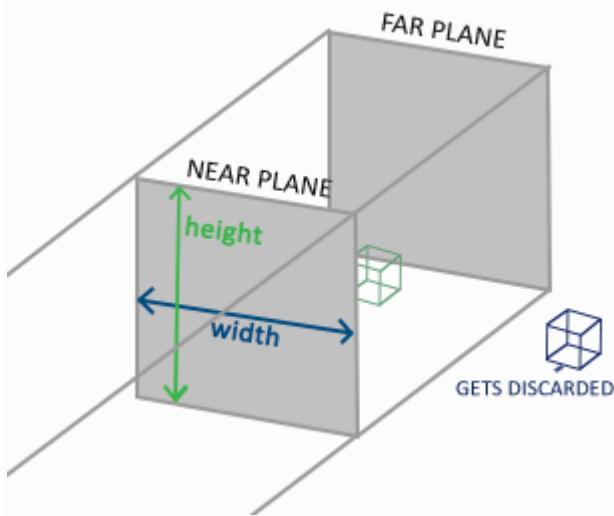
一旦所有顶点被转换到裁剪空间, 最终的操作——透视划分(**Perspective Division**)将会执行, 在这个过程中我们将位置向量的 x, y, z 分量分别除以向量的齐次 w 分量; 透视划分是将 4 维裁剪空间坐标转换为 3 维标准化设备坐标。这一步会在每一个顶点着色器运行的最后被自动执行。

在这一阶段之后, 坐标经过转换的结果将会被映射到屏幕空间(由 `glViewport` 设置)且被转换成片段。

投影矩阵将观察坐标转换为裁剪坐标的过程采用两种不同的方式，每种方式分别定义自己的平截头体。我们可以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(Orthographic Projection)

正射投影矩阵定义了一个类似立方体的平截头体，指定了一个裁剪空间，每一个在这空间外面的顶点都会被裁剪。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。所有在使用正射投影矩阵转换到裁剪空间后如果还处于这个平截头体里面的坐标就不会被裁剪。它的平截头体看起来像一个容器：



上面的平截头体定义了由宽、高、近平面和远平面决定的可视的坐标系。任何出现在近平面前面或远平面后面的坐标都会被裁剪掉。正视平截头体直接将平截头体内部的顶点映射到标准化设备坐标系中，因为每个向量的 w 分量都是不变的；如果 w 分量等于 1.0，则透视为划分不会改变坐标的值。

为了创建一个正射投影矩阵，我们利用 GLM 的构建函数 `glm::ortho`：

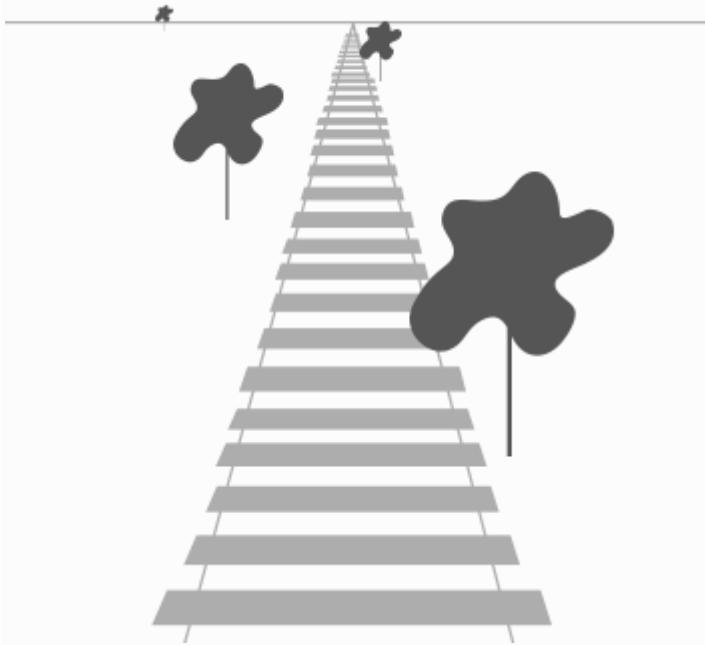
```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

前两个参数指定了平截头体的左右坐标，第三和第四参数指定了平截头体的底部和上部。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。这个指定的投影矩阵将处于这些 x , y , z 范围之间的坐标转换到标准化设备坐标系中。

正射投影矩阵直接将坐标映射到屏幕的二维平面内，但实际上一个直接的投影矩阵将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(Perspective Projection)

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。这个神奇的效果我们称之为**透视**。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，由于透视的原因，平行线似乎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。这个投影矩阵不仅将给定的平截头体范围映射到裁剪空间，同样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。被转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对象都会被裁剪掉)。OpenGL 要求所有可见的坐标都落在 -1.0 到 1.0 范围内从而作为最后的顶点着色器输出，因此一旦坐标在裁剪空间内，透视划分就会被应用到裁剪空间坐标：

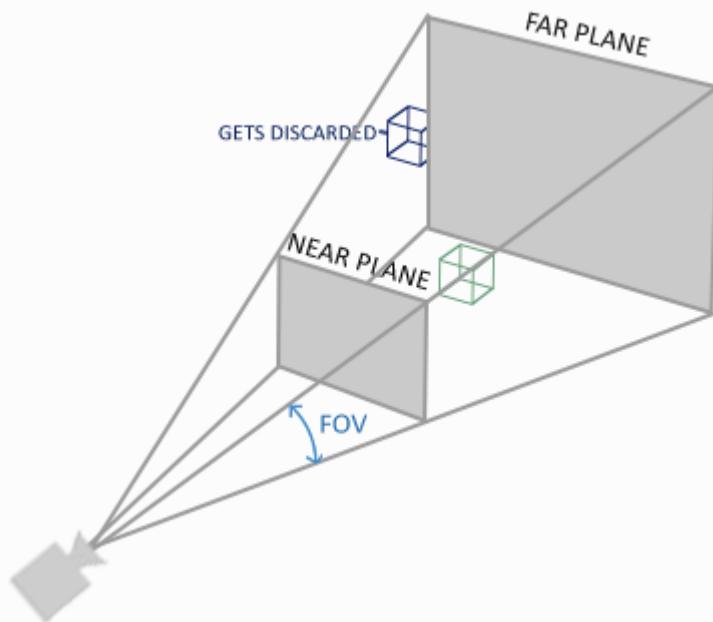
$$out = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

每个顶点坐标的分量都会除以它的 **w** 分量，得到一个距离观察者的较小的顶点坐标。这是也是另一个 **w** 分量很重要的原因，因为它能够帮助我们进行透射投影。最后的结果坐标就是处于标准化设备空间内的。如果你对研究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(45.0f, (float)width/(float)height,  
0.1f, 100.0f);
```

`glm::perspective` 所做的其实就是再次创建了一个定义了可视空间的大的平截头体，任何在这个平截头体的对象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可视化为一个不均匀形状的盒子，在这个盒子内部的每个坐标都会被映射到裁剪空间的点。一张透视平截头体的照片如下所示：



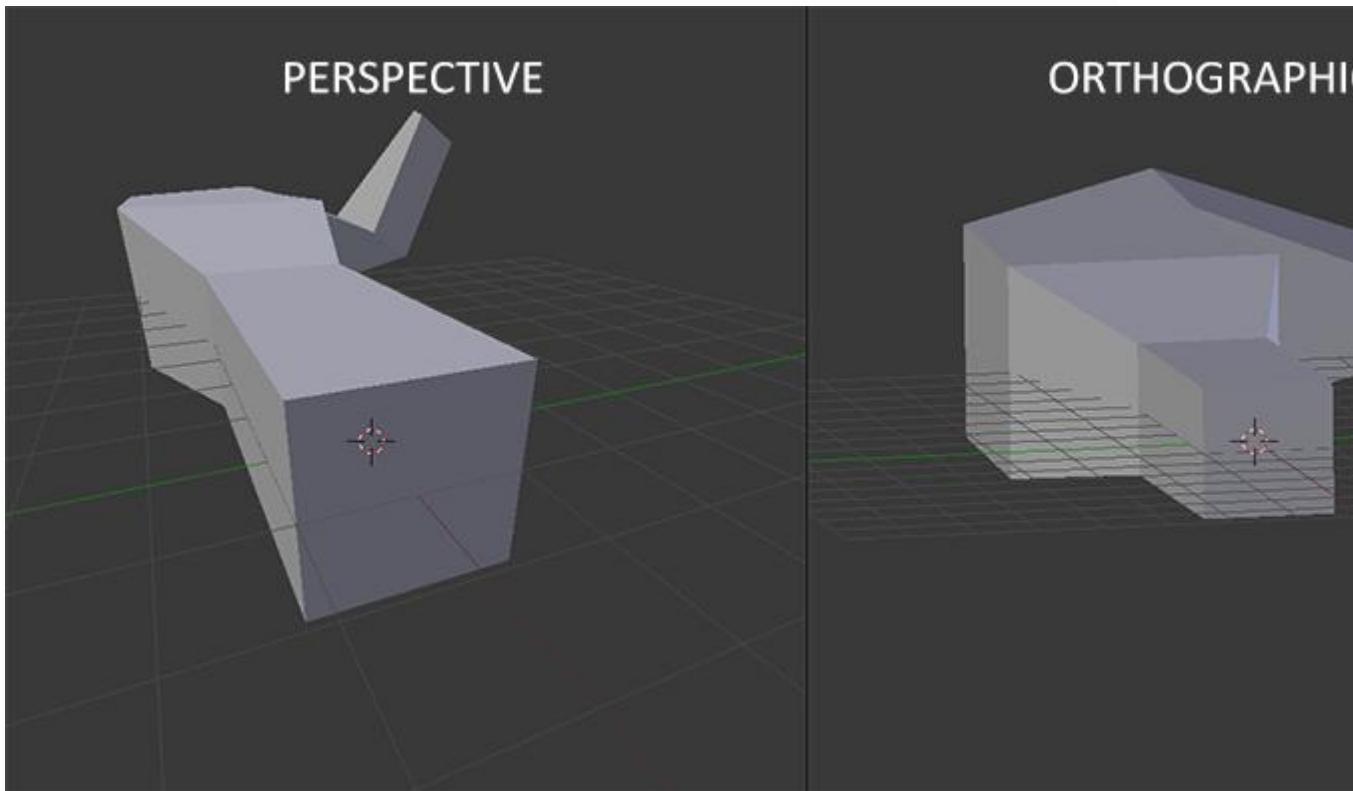
它的第一个参数定义了 **fov** 的值，它表示的是视野(Field of View)，并且设置了观察空间的大小。对于一个真实的观察效果，它的值经常设置为 45.0，但想要看到更多结果你可以设置一个更大的值。第二个参数设置了宽高比，由视口的高度除以宽。第三和第四个参数设置了平截头体的近和远平面。我们经常设置近距离

为 0.1 而远距离设为 100.0。所有在近平面和远平面的顶点且处于平截头体内的顶点都会被渲染。

Important

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

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为划分(它仍然有进行透视为划分, 只是 w 分量没有被操作(它保持为 1)因此没有起作用)。因为正射投影没有使用透视为, 远处的对象不会显得小以产生神奇的视觉输出。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应用, 或者是那些我们不需要使用投影来转换顶点的情况下。某些如 **Blender** 的进行三维建模的软件有时在建模时会使用正射投影, 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转换矩阵：模型矩阵、观察矩阵和投影矩阵。一个顶点的坐标将会根据以下过程被转换到裁剪坐标：

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

注意每个矩阵被运算的顺序是相反的(记住我们需要从右往左乘上每个矩阵)。最后的顶点应该被赋予顶点着色器中的 `gl_Position` 且 OpenGL 将会自动进行透视划分和裁剪。

Important

然后呢？

顶点着色器的输出需要所有的顶点都在裁剪空间内，而这是我们的转换矩阵所做的。OpenGL 然后在裁剪空间中执行透视划分从而将它们转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(在我们的例子中屏幕是 800 * 600)。这个过程称为视口转换。

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好好运用这些坐标空间的并且会有足够的展示例子在接下来的教程中。

进入三维

既然我们知道了如何将三维坐标转换为二维坐标，我们可以开始将我们的对象展示为三维对象而不是目前我们所展示的缺胳膊少腿的二维平面。

在开始进行三维画图时，我们首先创建一个模型矩阵。这个模型矩阵包含了平移、缩放与旋转，我们将会运用它来将对象的顶点转换到全局世界空间。让我们平移一下我们的平面，通过将其绕着 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, -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转换到世界坐标。我们的平面看起来就是在地板上的因此可以代表真实世界的平面。

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对象变成可见的(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，思考下面的问题：

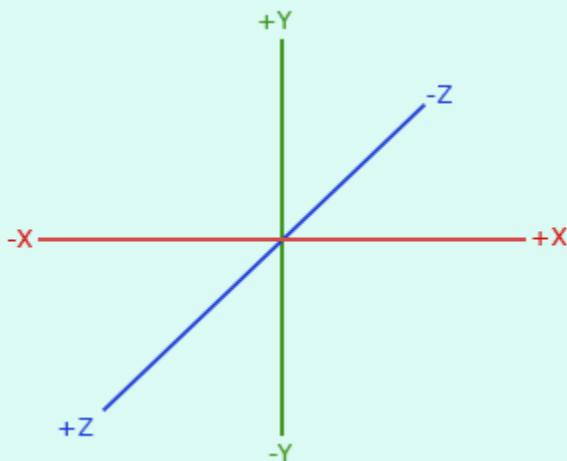
- 将摄像机往后移动跟将整个场景往前移是一样的。

这就是观察空间所做的，我们以相反于移动摄像机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)所以我们要沿着 z 轴的负方向移动。我们会通过将场景沿着 z 轴正方向平移来实现这个。它会给我们一种我们在往后移动的感觉。

Important

右手坐标系(Right-handed System)

按照约定，OpenGL 是一个右手坐标系。最基本的就是说正 x 轴在你的右手边，正 y 轴往上而正 z 轴是往后的。想象你的屏幕处于三个轴的中心且正 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：

- 张开你的右手使正 y 轴沿着你的手往上。
- 使你的大拇指往右。
- 使你的食指往上。
- 向下 90 度弯曲你的中指。

如果你都正确地做了，那么你的大拇指朝着正 x 轴方向，食指朝着正 y 轴方向，中指朝着正 z 轴方向。如果你用左手来做这些动作，你会发现 z 轴的方向是相反的。这就是有名的左手坐标系，它被 DirectX 广泛地使用。注意在标准化设备坐标系中 OpenGL 使用的是左手坐标系(投影矩阵改变了惯用手的习惯)。

在下一个教程中我们将会详细讨论如何在场景中移动。目前的观察矩阵是这样的：

```
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。我们想要在我们的场景中使用透视投影所以我们声明的投影矩阵是像这样的：

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

Attention

再重复一遍，在 `glm` 指定角度的时候要注意。这里我们将参数 `fov` 设置为 45 度，但有些 GLM 的实现是将 `fov` 当成弧度，在这种情况下你需要使用 `glm::radians(45.0)` 来设置。

既然我们创建了转换矩阵，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单位转换矩阵然后将它乘以顶点坐标：

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

```
uniform mat4 projection;

void main()
{
    // 注意从右向左读

    gl_Position = projection * view * model * vec4(position, 1.0f);

    ...
}
```

我们应该将矩阵传入着色器(这通常在每次渲染的时候即转换矩阵将要改变的时候完成):

```
GLint modelLoc = glGetUniformLocation(ourShader.Program, "model");

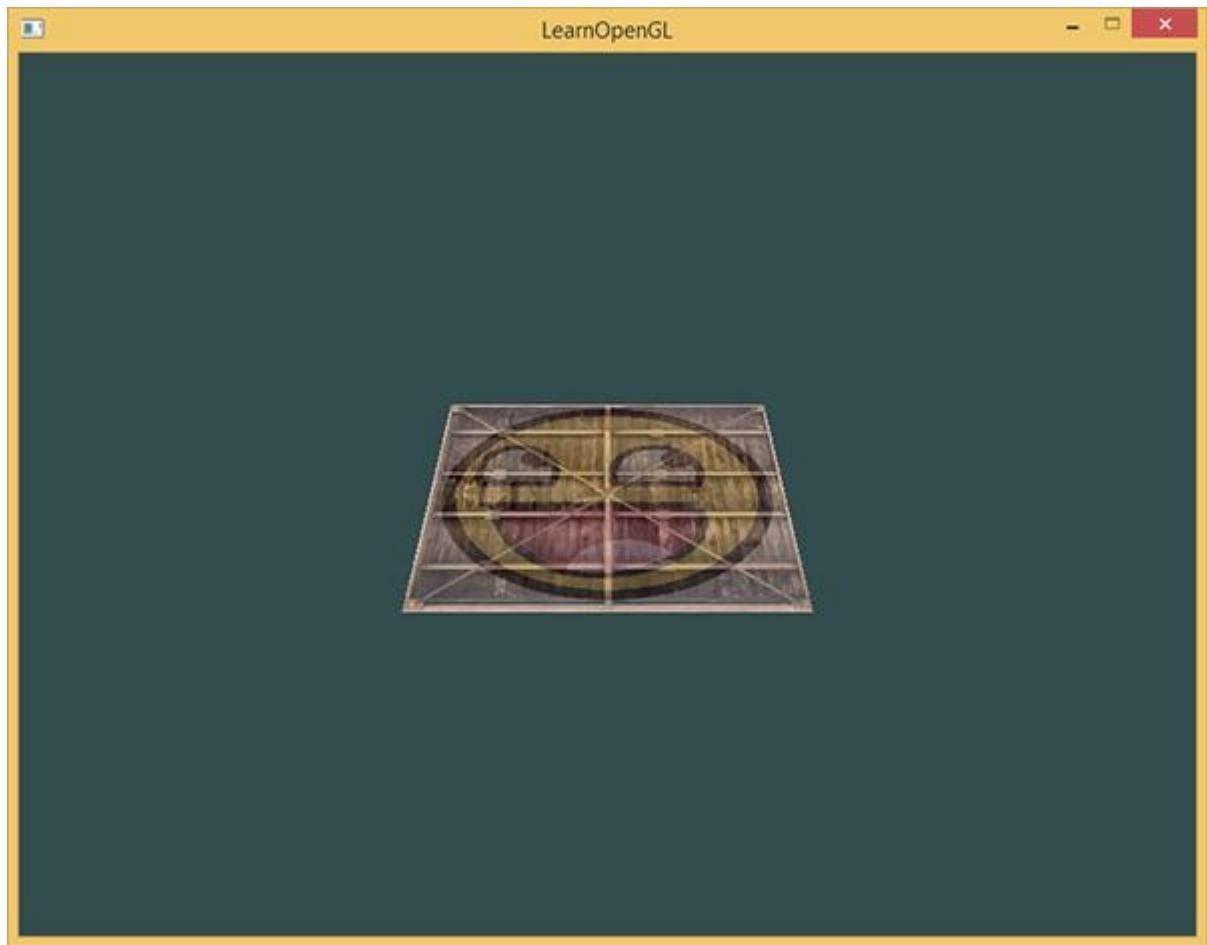
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

... // 观察矩阵和投影矩阵与之类似
```

现在我们的顶点坐标通过模型、观察和投影矩阵来转换，最后的对象应该是：

- 往后向地板倾斜。
- 离我们有点距离。
- 由透视展示(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三维的平面，是静止在一些虚构的地板上的。如果你不是得到相同的结果，请检查下完整的[源代码](#) 以及[顶点](#)和[片段](#)着色器。

更多的 3D

到目前为止，我们在二维平面甚至在三维空间中画图，所以让我们采取大胆的方式来将我们的二维平面扩展为三维立方体。要渲染一个立方体，我们一共需要 36 个顶点(6 个面 × 每个面有 2 个三角形组成 × 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](#)。注意，这一次我们省略了颜色值，因为这次我们只在乎顶点的位置和，我们使用纹理贴图。

为了好玩，我们将让立方体随着时间旋转：

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

然后我们使用 `glDrawArrays` 来画立方体，这一次总共有 36 个顶点。

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

如果一切顺利的话绘制效果将与下面的类似：

这有点像一个立方体，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的其他面的上面。之所以这样是因为 OpenGL 是通过画一个一个三角形来画你的立方体的，所以它将会覆盖之前已经画在那里的像素。因为这个原因，有些三角形会画在其它三角形上面，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 z 缓冲区(Z-buffer)里面，它允许 OpenGL 决定何时覆盖一个像素何时不覆盖。通过使用 z 缓冲区我们可以设置 OpenGL 来进行深度测试。

z 缓冲区

OpenGL 存储它的所有深度信息于 z 缓冲区中，也被称为深度缓冲区(Depth Buffer)。GLFW 会自动为你生成这样一个缓冲区 (就如它有一个颜色缓冲区来存储输出图像的颜色)。深度存储在每个片段里面(作为片段的 z 值)当片段像输出它的颜色时，OpenGL 会将它的深度值和 z 缓冲进行比较然后如果当前的片段在其它片段之后它将会被丢弃，然后重写。这个过程称为深度测试(Depth Testing)并且它是由 OpenGL 自动完成的。

然而，如果我们想要确定 OpenGL 是否真的执行深度测试，首先我们要告诉 OpenGL 我们想要开启深度测试；而这通常是默认关闭的。我们通过 `glEnable` 函数来开启深度测试。`glEnable` 和 `glDisable` 函数允许我们开启或关闭某一个 OpenGL 的功能。该功能会一直是开启或关闭的状态直到另一个调用来关闭或开启它。现在我们想开启深度测试就需要开启 `GL_DEPTH_TEST`：

```
glEnable(GL_DEPTH_TEST);
```

既然我们使用了深度测试我们也想要在每次重复渲染之前清除深度缓冲区(否则前一个片段的深度信息仍然保存在缓冲区中)。就像清除颜色缓冲区一样，我们可以通过在 `glClear` 函数中指定 `DEPTH_BUFFER_BIT` 位来清除深度缓冲区：

```
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转换到世界坐标系中。

首先，让我们为每个立方体定义一个转换向量来指定它在世界空间的位置。我们将要在 `glm::vec3` 数组中定义 10 个立方体位置向量。

```
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，在循环中，我们调用 `glDrawArrays` 10 次，在我们开始渲染之前每次传入一个不同的模型矩阵到顶点着色器中。我们将会创建一个小的循环来通过一个不同

的模型矩阵重复渲染我们的对象 10 次。注意我们也传入了一个旋转参数到每个箱子中：

```
glBindVertexArray(VAO);

for(GLuint i = 0; i < 10; i++)

{
    glm::mat4 model;

    model = glm::translate(model, cubePositions[i]);

    GLfloat angle = 20.0f * i;

    model = glm::rotate(model, angle, glm::vec3(1.0f, 0.3f, 0.5f));

    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    glDrawArrays(GL_TRIANGLES, 0, 36);
}

glBindVertexArray(0);
```

这个代码将会每次都更新模型矩阵然后画出新的立方体，如此总共重复 10 次。然后我们应该就能看到一个拥有 10 个正在奇葩旋转着的立方体的世界。



完美！这就像我们的箱子找到了志同道合的小伙伴一样。如果你在这里卡住了，你可以对照一下[代码](#)以及[顶点着色器](#)和[片段着色器](#)。

练习

- 对 GLM 的投影函数中的 `FoV` 和 `aspect-ratio` 参数进行试验。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平移，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。
- 只使用模型矩阵每次只让 3 个箱子旋转(包括第 1 个)而让剩下的箱子保持静止。[参考解答](#)。

摄像机(Camera)

原文

[Camera](#)

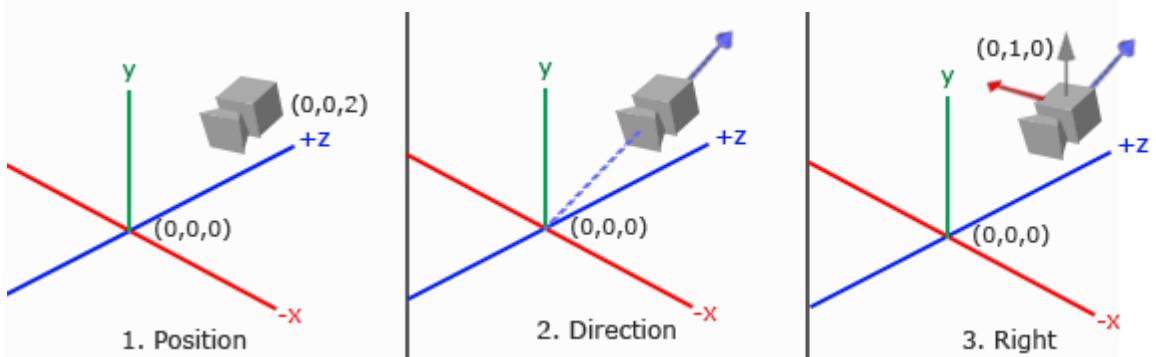
原文	<u>Camera</u>
作者	JoeyDeVries
翻译	Django
校对	Geequlim, BLumia

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。OpenGL本身没有摄像机的概念，但我们可以通过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，这样感觉就像我们在移动，而不是场景在移动。

本节我们将会讨论如何在 OpenGL 中模拟一个摄像机，将会讨论 FPS 风格的可自由在 3D 场景中移动的摄像机。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(Camera/View Space)

当我们讨论摄像机/观察空间的时候，是我们在讨论以摄像机的透视图作为场景原点时场景中所有可见顶点坐标。观察矩阵把所有的世界坐标变换到观察坐标，这些新坐标是相对于摄像机的位置和方向的。定义一个摄像机，我们需要一个摄像机在世界空间中的位置、观察的方向、一个指向它的右测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. 摄像机位置

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代表摄像机位置的向量。我们把摄像机位置设置为前面教程中的那个相同的位置：

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

Important

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，我们就往 z 轴正方向移动。

2. 摄像机方向

下一个需要的向量是摄像机的方向，比如它指向哪个方向。现在我们让摄像机指向场景原点：(0, 0, 0)。用摄像机位置向量减去场景原点向量的结果就是摄像机指向向量。由于我们知道摄像机指向 z 轴负方向，我们希望方向向量指向摄像机的 z 轴正方向。如果我们改变相减的顺序，我们就会获得一个指向摄像机正 z 轴方向的向量(译注：注意看前面的那个图，所说的「方向向量/Direction Vector」是指向 z 的正方向的，而不是摄像机所注视的那个方向)：

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
```

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

Attention

方向向量(Direction Vector)并不是最好的名字，因为它正好指向从它到目标向量的相反方向。

3. 右轴(Right axis)

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：定义一个上向量(Up Vector)。我们把上向量和第二步得到的摄像机方向向量进行叉乘。两个向量叉乘的结果就是同时垂直于两向量的向量，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(Up axis)

现在我们已经有了 x 轴向量和 z 轴向量，获取摄像机的正 y 轴相对简单；我们把右向量和方向向量(Direction Vector)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 Gram-Schmidt(葛兰—施密特)正交。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定义了一个坐标空间，里面有 3 个相互垂直的轴，你可以用这三个轴外加一个平移向量来创建一个矩阵，你可以用这个矩阵乘以任何向量来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：

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R 是右向量， U 是上向量， D 是方向向量 P 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。使用这个 LookAt 矩阵坐标观察矩阵可以很高效地把所有世界坐标变换为观察坐标。LookAt 矩阵就像它的名字表达的那样：它会创建一个观察矩阵 looks at(看着)一个给定目标。

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上向量的世界空间中的向量(我们使用上向量计算右向量)。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。它可以创建一个和前面所说的同样的观察矩阵。

在开始做用户输入之前，我们来做些有意思的事，把我们的摄像机在场景中旋转。我们的注视点保持在(0, 0, 0)。

我们在每一帧都创建 x 和 z 坐标，这要使用一点三角学知识。x 和 z 表示一个在一个圆圈上的一点，我们会使用它作为摄像机的位置。通过重复计算 x 和 y 坐标，遍历所有圆圈上的点，这样摄像机就会绕着场景旋转了。我们预先定义这个圆圈的半径，使用 `glfwGetTime` 函数不断增加它的值，在每次渲染迭代创建一个新的观察矩阵。

```
GLfloat radius = 10.0f;  
  
GLfloat camX = sin(glfwGetTime()) * radius;  
  
GLfloat 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));
```

如果你运行代码你会得到下面的东西：

这一小段代码中，摄像机围绕场景转动。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，这里有[源码](#)、[顶点](#)和[片段](#)着色器。

自由移动

让摄像机绕着场景转很有趣，但是让我们自己移动摄像机更有趣！首先我们必须设置一个摄像机系统，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);
```

我们首先设置之前定义的 `cameraPos` 为摄像机位置。方向(Direction)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，摄像机都会注视目标。我们在按下某个按钮时更新 `cameraPos` 向量。

我们已经为 GLFW 的键盘输入定义了一个 `key_callback` 函数，我们来添加几个新按键命令：

```
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，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。如果我们希望向旁边移动，我们做一个叉乘来创建一个右向量，沿着它移动就可以了。这样就创建了类似使用摄像机横向、前后移动的效果。

Important

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` 变量的大小返回不同的大小。如果我们不对向量进行标准化，我们就得根据摄像机的方位加速或减速移动了，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。

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，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。大多数 GUI 系统都是这样的，它对摄像机来说用并不合理。我们可以用一些小技巧解决这个问题。

这个技巧是只在回调函数中跟踪哪个键被按下/释放。在游戏循环中我们读取这些值，检查那个按键被激活了，然后做出相应反应。我们只储存哪个键被按下/释放的状态信息，在游戏循环中对状态做出反应，我们来创建一个布尔数组代表按下/释放的键：

```
bool keys[1024];
```

然后我们必须在 `key_callback` 函数中设置按下/释放键为 `true` 或 `false`：

```
if(action == GLFW_PRESS)  
    keys[key] = true;  
  
else if(action == GLFW_RELEASE)  
    keys[key] = false;
```

我们创建一个新的叫做 `do_movement` 的函数，用它根据按下的按键来更新摄像机的值：

```
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;  
}
```

之前的代码移动到了 `do_movement` 函数中。由于所有 GLFW 的按键枚举都是整数，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，你可以同时向多个方向移动了，并且当你按下按钮也会立刻运动了。如遇困难查看[源码](#)。

移动速度

目前我们的移动速度是个常量。看起来不错，但是实际情况下根据处理器的能力不同，有的人在同一段时间内会比其他人绘制更多帧。也就是调用了更多次 `do_movement` 函数。每个人的运动速度就都不同了。当你要发布的你应用的时候，你必须确保在所有硬件上移动速度都一样。

图形和游戏应用通常有回跟踪一个 `deltaTime` 变量，它储存渲染上一帧所用的时间。我们把所有速度都去乘以 `deltaTime` 值。当我们的 `deltaTime` 变大时意味着上一帧渲染花了更多时间，所以这一帧使用这个更大的 `deltaTime` 的值乘以速度，会获得更高的速度，这样就与上一帧平衡了。使用这种方法时，无论你的机器快还是慢，摄像机的速度都会保持一致，这样每个用户的体验就都一样了。

我们要用两个全局变量来计算出 `deltaTime` 值：

```
GLfloat deltaTime = 0.0f;    // 当前帧遇上一帧的时间差  
GLfloat 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

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

现在我们有了 `deltaTime` 在计算速度的使用可以使用了：

```
void Do_Movement()  
{  
    GLfloat cameraSpeed = 5.0f * 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。这里是源码。我们可以看到任何移动都会影响返回的 `deltaTime` 值。

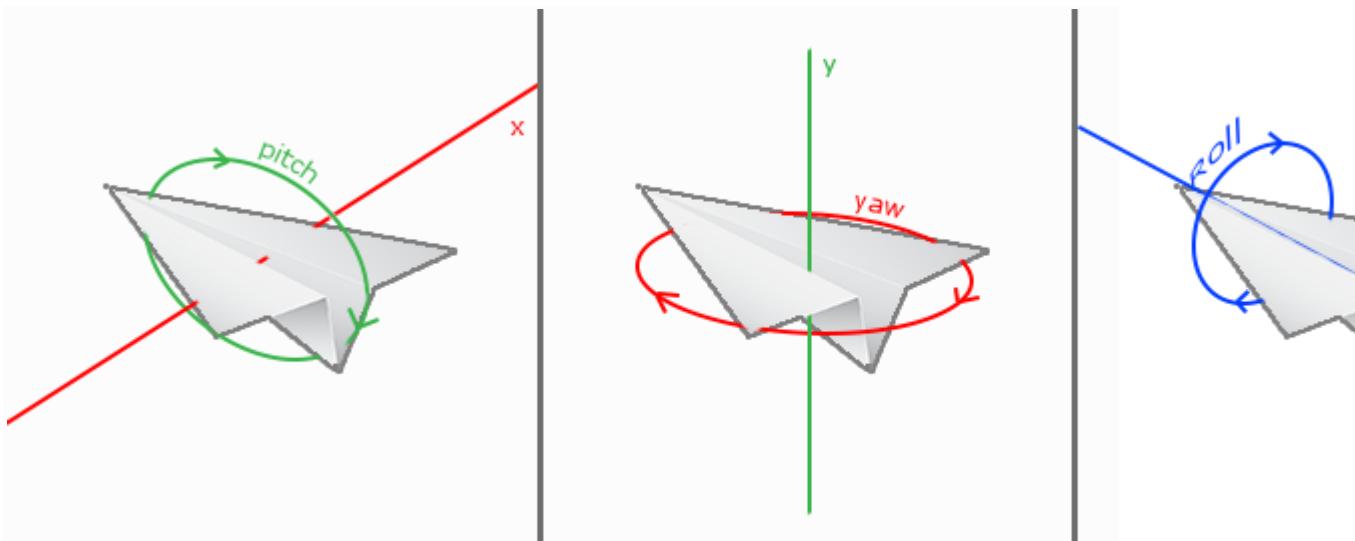
自由观看

只用键盘移动没什么意思。特别是我们还不能转向。是时候使用鼠标了！

为了能够改变方向，我们必须根据鼠标的输入改变 `cameraFront` 向量。然而，根据鼠标旋转改变方向向量有点复杂，需要更多的三角学知识。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。

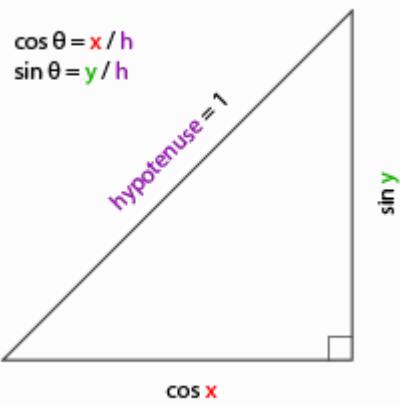
欧拉角

欧拉角是表示 3D 空间中可以表示任何旋转的三个值，由莱昂哈德·欧拉在 18 世纪提出。有三种欧拉角：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：



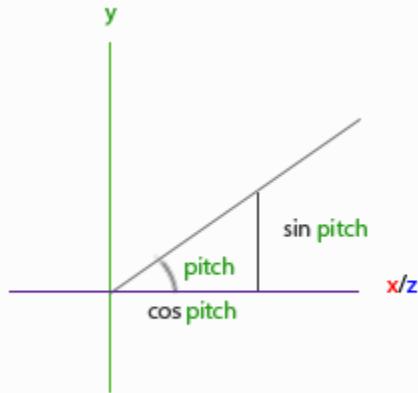
俯仰角是描述我们如何往上和往下看的角，它在第一张图中表示。第二张图显示了偏航角，偏航角表示我们往左和往右看的大小。滚转角代表我们如何翻滚摄像机。每个欧拉角都有一个值来表示，把三个角结合起来我们就能够计算 3D 空间中任何的旋转了。

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。用一个给定的俯仰角和偏航角，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，我们以最基本的情况开始：



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
 $\cos x/h = \cos x/1 = \cos x$ ，它的对边是

$\sin y/h = \sin y/1 = \sin y$ 。这样我们获得了能够得到 x 和 y 方向的长度的公式，它们取决于所给的角度。我们使用它来计算方向向量的元素：



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，正望向 y 轴，我们可以基于第一个三角形计算长度/y 方向的强度(我们往上或往下看多少)。从图中我们可以看到一个给定俯仰角的 y 值等于 $\sin\theta$:

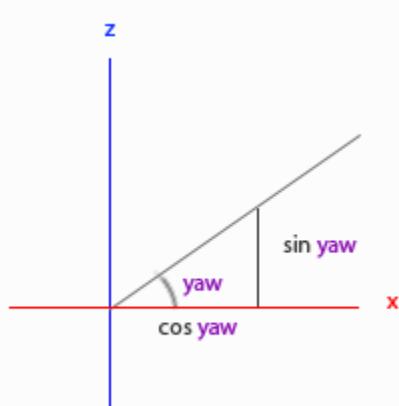
```
direction.y = sin(glm::radians(pitch)); // 注意我们先把角度转为弧度
```

这里我们只更新了 y 值，仔细观察 x 和 z 元素也被影响了。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元素：



就像俯仰角一样我们可以看到 x 元素取决于 $\cos(\text{偏航角})$ 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:

Important

译注:这里的球坐标与笛卡尔坐标的转换把 x 和 z 弄反了, 如果你去看最后的源码, 会发现作者在摄像机源码那里写了 $\text{yaw} = \text{yaw} - 90$, 实际上在这里 x 就应该是 $\sin(\text{glm}::\text{radians}(\text{yaw}))$, z 也是同样处理, 当然也可以认为是这个诡异的坐标系, 但是在这里使用球坐标转笛卡尔坐标有个大问题, 就是在初始渲染时, 无法指定摄像机的初始朝向, 还要花一些功夫自己实现这个; 此外这只能实现像第一人称游戏一样的简易摄像机, 类似 Maya、Unity3D 编辑器窗口的那种摄像机还是最好自己设置摄像机的位置、上、右、前轴, 在旋转时用四元数对这四个变量进行调整, 才能获得更好的效果, 而不是仅仅调整摄像机前轴。

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));//
```

译注: *direction* 代表摄像机的“前”轴, 但此前轴是和本文第一幅图片的第二个摄

像机的*direction* 是相反的

```
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的摄像机的 3 个维度的方向向量了。你可能会奇怪: 我们怎么得到俯仰角和偏航角?

鼠标输入

偏航角和俯仰角是从鼠标移动获得的，鼠标水平移动影响偏航角，鼠标垂直移动影响俯仰角。它的思想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和上一帧的位置相差多少。如果差别越大那么俯仰角或偏航角就改变越大。

首先我们要告诉 **GLFW**，应该隐藏光标，并**捕捉(Capture)**它。捕捉鼠标意味着当应用集中焦点到鼠标上的时候光标就应该留在窗口中(除非应用拾取焦点或退出)。我们可以进行简单的配置：

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

这个函数调用后，无论我们怎么去移动鼠标，它都不会显示了，也不会离开窗口。对于 **FPS** 摄像机系统来说很好：

为计算俯仰角和偏航角我们需要告诉 **GLFW** 监听鼠标移动事件。我们用下面的原型创建一个回调函数来做这件事(和键盘输入差不多)：

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

这里的 `xpos` 和 `ypos` 代表当前鼠标的位置。我们注册了 **GLFW** 的回调函数，鼠标一移动 `mouse_callback` 函数就被调用：

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

在处理 **FPS** 风格的摄像机鼠标输入的时候，我们必须在获取最终的方向向量之前做下面这几步：

1. 计算鼠标和上一帧的偏移量。
2. 把偏移量添加到摄像机和俯仰角和偏航角中。
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。

第一步计算鼠标自上一帧的偏移量。我们必须先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 800 乘 600)：

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

然后在回调函数中我们计算当前帧和上一帧鼠标位置的偏移量：

```
GLfloat xoffset = xpos - lastX;
```

```
GLfloat yoffset = lastY - ypos; // 注意这里是相反的，因为y 坐标的范围
```

是从下往上的

```
lastX = xpos;
```

```
lastY = ypos;
```

```
GLfloat sensitivity = 0.05f;
```

```
xoffset *= sensitivity;
```

```
yoffset *= sensitivity;
```

注意我们把偏移量乘以了 `sensitivity` 值。如果我们移除它，鼠标移动就会太大了；你可以自己调整 `sensitivity` 的值。

下面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。对于俯仰角，要让用户不能看向高于 89 度(90 度时视角会逆转，所以我们把 89 度作为极限)的地方，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更进一步超越过去。限制可以这样做：

```
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是因为我们不希望限制用户的水平旋转。然而，给偏航角设置限制也很容易，只要你愿意。

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前面提到的实际方向向量：

```
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);
```

这回计算出方向向量，根据鼠标点的移动它包含所有的旋转。由于 `cameraFront` 向量已经包含在 `glm::lookAt` 函数中，我们直接去设置。

如果你现在运行代码，你会发现当程序运行第一次捕捉到鼠标的时候摄像机会突然跳一下。原因是当你的鼠标进入窗口鼠标回调函数会使用这时的 `xpos` 和 `ypos`。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布尔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位置更新为 `xpos` 和 `ypos`，这样就能解决这个问题；最后的鼠标移动会使用进入以后鼠标的位置坐标来计算它的偏移量：

```
if(firstMouse) // 这个bool 变量一开始是设定为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;
}
```

```
GLfloat xoffset = xpos - lastX;
```

```
GLfloat yoffset = lastY - ypos;
```

```
lastX = xpos;
```

```
lastY = ypos;
```

```
GLfloat 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)) * sin(glm::radians(pitch));

cameraFront = glm::normalize(front);

}
```

现在我们可以自由的在 3D 场景中移动了！如果你遇到困难，[这是](#)源码。

缩放

我们还要往摄像机系统里加点东西，实现一个缩放接口。前面教程中我们说视野 (Field of View 或 fov) 定义了我们可以看到场景中多大的范围。当视野变小时可视区域就会减小，产生放大了的感觉。我们用鼠标滚轮来放大。和鼠标移动、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(aspect >= 1.0f && aspect <= 45.0f)
        aspect -= yoffset;
    if(aspect <= 1.0f)
        aspect = 1.0f;
    if(aspect >= 45.0f)
        aspect = 45.0f;
}
```

`yoffset` 值代表我们滚动的大小。当 `scroll_callback` 函数调用后，我们改变全局 `aspect` 变量的内容。因为 `45.0f` 是默认的 `fov`，我们将会把缩放级别限制在 `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，但这一次使 `aspect` 变量 作为它的 `fov`：

```
projection = glm::perspective(aspect, (GLfloat)WIDTH/(GLfloat)HEIGHT,
```

```
0.1f, 100.0f);
```

最后不要忘记注册滚动回调函数：

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

现在我们实现了一个简单的摄像机系统，它能够让我们在 3D 环境中自由移动。

自由的去实验，如果遇到困难对比[源代码](#)。

Important

注意，使用欧拉角作为摄像机系统并不完美。你仍然可能遇到[万向节死锁](#)。最好的摄像机系统是使用四元数的，后面会有讨论。

摄像机类

接下来的教程我们会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教程的很大的篇幅，我们会从细节抽象出创建一个自己的摄像机对象。与着色器教程不同我们不会带你一步一步创建摄像机类，如果你想知道怎么工作的的话，只会给你提供一个(有完整注释的)源码。

像着色器对象一样，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。你应该能够理解所有的代码。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。

Attention

我们介绍的欧拉角 **FPS** 风格摄像机系统能够满足大多数情况需要，但是在创建不同的摄像机系统，比如飞行模拟就要当心。每个摄像机系统都有自己的有点和不足，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，由于使用了固定的上向量(0, 1, 0)，我们就不能用滚转角。

使用新的摄像机对象的更新后的版本源码可以[在这里找到](#)。(译注：总而言之这个摄像机实现并不十分完美，你可以看看最终的源码。建议先看[这篇文章](#)，对旋转有更深的理解后，你就能做出更好的摄像机类，不过本文有些内容比如如何防止按键停顿和 glfw 鼠标事件实现摄像机的注意事项比较重要，其它的就要做一定的取舍了)

练习

- 看看你是否能够变换摄像机类从而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，使你能够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的工作：[参考解答](#)

复习

原文	Review
作者	JoeyDeVries
翻译	Meow J
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。

这些就是我们在前几章学习的内容，尝试在教程的基础上进行改动程序，或者实验你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。
- GLEW:** 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。
- 图形管道(Graphics Pipeline):** 一个顶点在呈现为像素之前通过的过程。

- **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来的功能.
- **标准化设备坐标(Normalized Device Coordinates):** 顶点在通过在剪裁坐标系中剪裁与透视为划分后最终呈现在的坐标系. 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.
- **元素缓冲对象(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)并且只能被设定一次.
- **纹理(Texture):** 一种缠绕物体的特殊类型图片, 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):** 被存储的材质一些的缩小版本, 根据距观察者的距离会使用材质的合适大小.
- **SOIL:** 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):** 一个对象的初始空间. 所有的坐标都是相对于对象的原点的.
- **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/透视为划分).
- **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:** 一种特殊类型的观察矩阵, 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), 和滚动角(roll)从而允许我们通过这三个值构造任何 3D 方向.

颜色

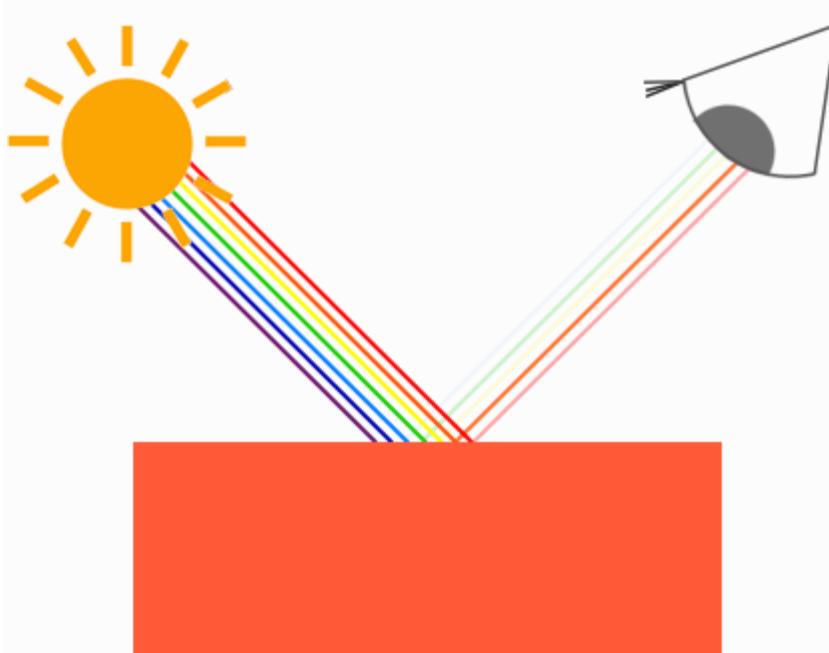
原文	Colors
作者	JoeyDeVries
翻译	Geequlim
校对	Geequlim

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。本节我们将会更广泛地讨论颜色，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。我们要做的工作是使用(有限的)数字来模拟真实世界中(无限)的颜色，因此并不是所有的现实世界中的颜色都可以用数字来表示。然而我们依然可以用数字来代表许多种颜色，并且你甚至可能根本感觉不到他们与真实颜色之间的差异。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。这三个不同的分量组合在一起几乎可以表示存在的任何一种颜色。例如，要获取一个珊瑚红(Coral)颜色我们可以这样定义一个颜色向量：

```
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的真实颜色，而是它所反射(Reflected)的颜色。换句话说，那些不能被物体吸收(Absorb)的颜色(被反射的颜色)就是我们能够感知到的物体的颜色。例如，太阳光被认为是由许多不同的颜色组合成的白色光(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颜色，不被吸收的蓝色光被反射到我们的眼中，使我们看到了一个蓝色的玩具。下图显示的是一个珊瑚红的玩具，它以不同强度的方式反射了几种不同的颜色。



正如你所见，白色的阳光是一种所有可见颜色的集合，上面的物体吸收了其中的大部分颜色，它仅反射了那些代表这个物体颜色的部分，这些被反射颜色的组合就是我们感知到的颜色(此例中为珊瑚红)。

这些颜色反射的规律被直接地运用在图形领域。我们在 OpenGL 中创建一个光源时都会为它定义一个颜色。在前面的段落中所提到光源的颜色都是白色的，那我们就继续来创建一个白色的光源吧。当我们把光源的颜色与物体的颜色相乘，所得到的就是这个物体所反射该光源的颜色(也就是我们感知到的颜色)。让我们再次审视我们的玩具(这一次它还是珊瑚红)并看看如何计算出他的反射颜色。我们通过检索结果颜色的每一个分量来看一下光源色和物体颜色的反射运算：

```
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);
```

我们可以看到玩具在进行反射时吸收了白色光源颜色中的大部分颜色，但它对红、绿、蓝三个分量都有一定的反射，反射量是由物体本身的颜色所决定的。这也代表着现实中的光线原理。由此，我们可以定义物体的颜色为这个物体从一个光源反射各个颜色分量的多少。现在，如果我们使用一束绿色的光又会发生什么呢？

```
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，我们的玩具没有红色和蓝色的光让它来吸收或反射，这个玩具也吸收了光线中一半的绿色，当然它仍然反射了光的一半绿色。它现在看上去是深绿色(Dark-greenish)的。我们可以看到，如果我们用一束绿色的光线照来照射玩具，那么只有绿色能被反射和感知到，没有红色和蓝色能被反射和感知。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)的光线：

```
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);
```

如你所见，我们可以通过物体对不同颜色光的反射来的得到意想不到的颜色，从此创作颜色已经变得非常简单。

目前有了这些颜色相关的理论已经足够了，接下来我们将创建一个场景用来做更多的实验。

创建一个光照场景

在接下来的教程中，我们将通过模拟真实世界中广泛存在的光照和颜色现象来创建有趣的视觉效果。现在我们将在场景中创建一个看得到的物体来代表光源，并且在场景中至少添加一个物体来模拟光照。

首先我们需要一个物体来投光(Cast the light)，我们将无耻地使用前面教程中的立方体箱子。我们还需要一个物体来代表光源，它代表光源在这个3D空间中的确切位置。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？)。

当然，填一个顶点缓冲对象(VBO)，设定一下顶点属性指针和其他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。

所以，我们首先需要一个顶点着色器来绘制箱子。与上一个教程的顶点着色器相比，容器的顶点位置保持不变(虽然这一次我们不需要纹理坐标)，因此顶点着色器中没有新的代码。我们将会使用上一篇教程顶点着色器的精简版：

```
#version 330 core  
  
layout (location = 0) in vec3 position;  
  
uniform mat4 model;  
uniform mat4 view;  
uniform mat4 projection;  
  
void main()  
{  
    gl_Position = projection * view * model * vec4(position, 1.0f);  
}
```

请确认更新你的顶点数据和属性对应的指针与新的顶点着色器一致(当然你可以继续保留纹理数据并保持属性对应的指针有效。在这一节中我们不使用纹理，但如果你想要一个全新的开始那也不是什么坏主意)。

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特殊的 VAO。当然我们也可以让这个灯和其他物体使用同一个 VAO 然后对他的 `model`(模型)矩阵做一些变换，然而接下来的教程中我们会频繁地对顶点数据做一些改变并且需要改变属性对应指针设置，我们并不想因此影响到灯(我们只在乎灯的位置)，因此我们有必要为灯创建一个新的 VAO。

```
GLuint lightVAO;  
  
 glGenVertexArrays(1, &lightVAO);  
  
 glBindVertexArray(lightVAO);
```

```
// 只需要绑定VBO 不用再次设置VBO 的数据，因为容器(物体)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指针(仅设置灯的顶点数据)  
  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),  
                      (GLvoid*)0);  
  
glEnableVertexAttribArray(0);  
  
 glBindVertexArray(0);
```

这段代码对你来说应该非常直观。既然我们已经创建了表示灯和被照物体的立方体，我们只需要再定义一个东西就行了，那就是片段着色器

```
#version 330 core  
  
out vec4 color;  
  
uniform vec3 objectColor;  
uniform vec3 lightColor;  
  
void main()  
{  
    color = vec4(lightColor * objectColor, 1.0f);  
}
```

这个片段着色器接受两个分别表示物体颜色和光源颜色的 uniform 变量。正如本篇教程一开始所讨论的一样，我们将光源的颜色与物体(能反射)的颜色相乘。这个着色器应该很容易理解。接下来让我们把物体的颜色设置为上一节中所提到的珊瑚红并把光源设置为白色：

```
// 在此之前不要忘记首先'使用'对应的着色器程序(来设定uniform)

GLint objectColorLoc = glGetUniformLocation(lightingShader.Program,
"objectColor");

GLint lightColorLoc = glGetUniformLocation(lightingShader.Program,
"lightColor");

glUniform3f(objectColorLoc, 1.0f, 0.5f, 0.31f); // 我们所熟悉的珊瑚红

glUniform3f(lightColorLoc, 1.0f, 1.0f, 1.0f); // 依旧把光源设置为白

色
```

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。我们不希望灯对象的颜色在接下来的教程中因光照计算的结果而受到影响，而希望它能够独立。希望表示灯不受其他光照的影响而一直保持明亮(这样它才更像是一个真实的光源)。

为了实现这个目的，我们需要为灯创建另外的一套着色器程序，从而能保证它能够在其他光照着色器变化的时候保持不变。顶点着色器和我们当前的顶点着色器是一样的，所以你可以直接把灯的顶点着色器复制过来。片段着色器保证了灯的颜色一直是亮的，我们通过给灯定义一个常量的白色来实现：

```
#version 330 core

out vec4 color;

void main()
{
    color = vec4(1.0f); // 设置四维向量的所有元素为 1.0f
}
```

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绘制箱子(或者可能是其它的一些物体)，让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升级这个光照着色器从而能够缓慢的实现更真实的效果。

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，我们将表示光源的灯立方体绘制在与光源同样的位置。我们将使用我们为它新建的片段着色器让它保持它一直处于白色状态，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);
```

然后我们把灯平移到这儿，当然我们需要对它进行缩放，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
```

```
...
```

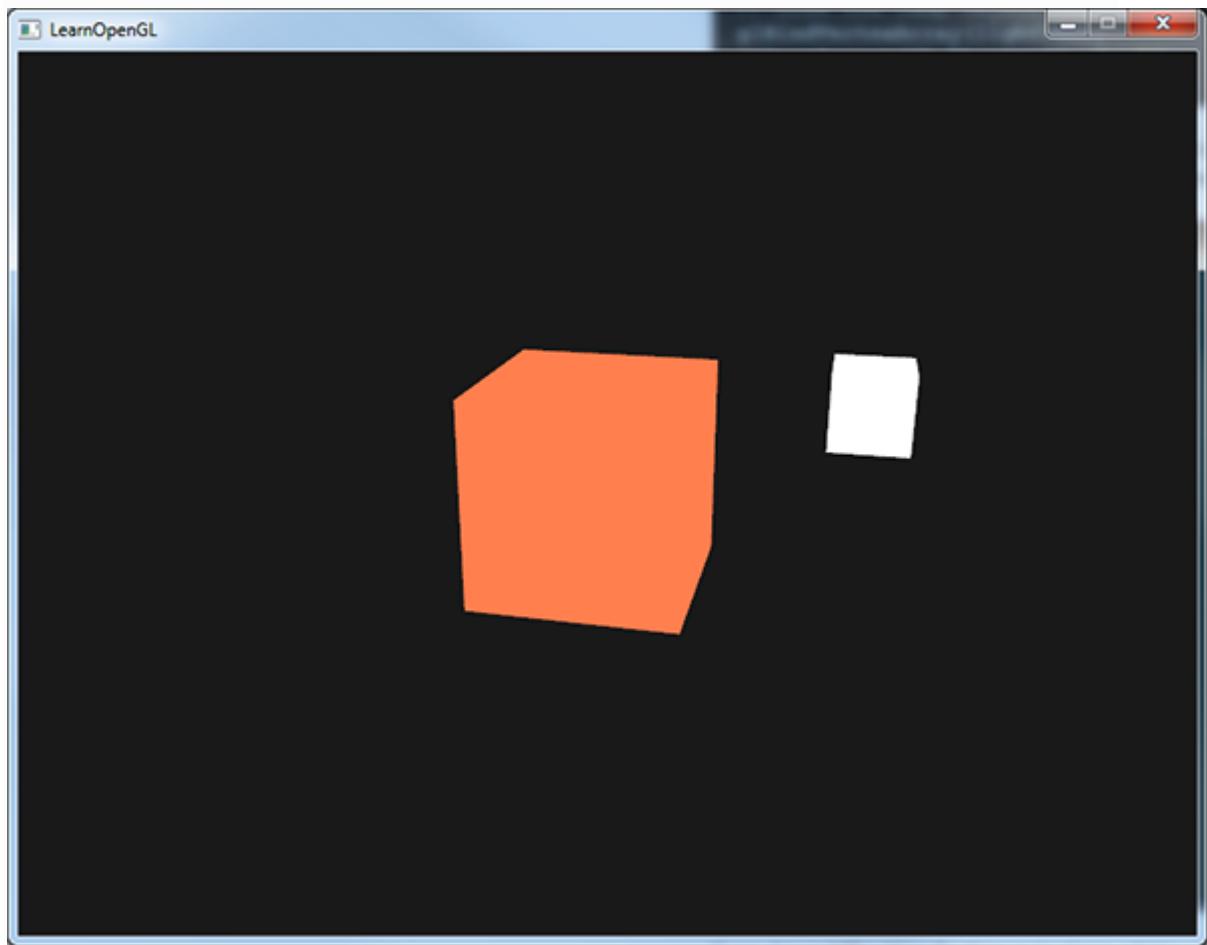
```
// 绘制灯立方体对象
```

```
glBindVertexArray(lightVAO);
```

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

```
glBindVertexArray(0);
```

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给你有趣的视觉效果。

如果你在把上述代码片段放到一起编译遇到困难，可以去认真地看看我的[源代码](#)。你好最自己实现一遍这些操作。

现在我们有了一些关于颜色的知识，并且创建了一个基本的场景能够绘制一些漂亮的光线。你现在可以阅读[下一个教程](#)，真正的魔法即将开始！

光照基础

原文	Basic Lighting
作者	JoeyDeVries
翻译	Django
校对	Geequlim, BLumia

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，这是以目前我们所拥有的处理能力无法模拟的。因此 OpenGL 的光照仅仅使用了简化的模型并基于对现实的估计来进行模拟，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 个元素组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。这些光照元素看起来像下面这样：



- 环境光照(Ambient Lighting): 即使在黑暗的情况下，世界上也仍然有一些光亮(月亮、一个来自远处的光)，所以物体永远不会是完全黑暗的。我们使用环境光照来模拟这种情况，也就是无论如何永远都给物体一些颜色。
- 漫反射光照(Diffuse Lighting): 模拟一个发光物对物体的方向性影响(Direction Impact)。它是冯氏光照模型最显著的组成部分。面向光源的一面比其他面会更亮。
- 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元素。我们将以最简单的一个开始：**环境光照**。

环境光照(Ambient Lighting)

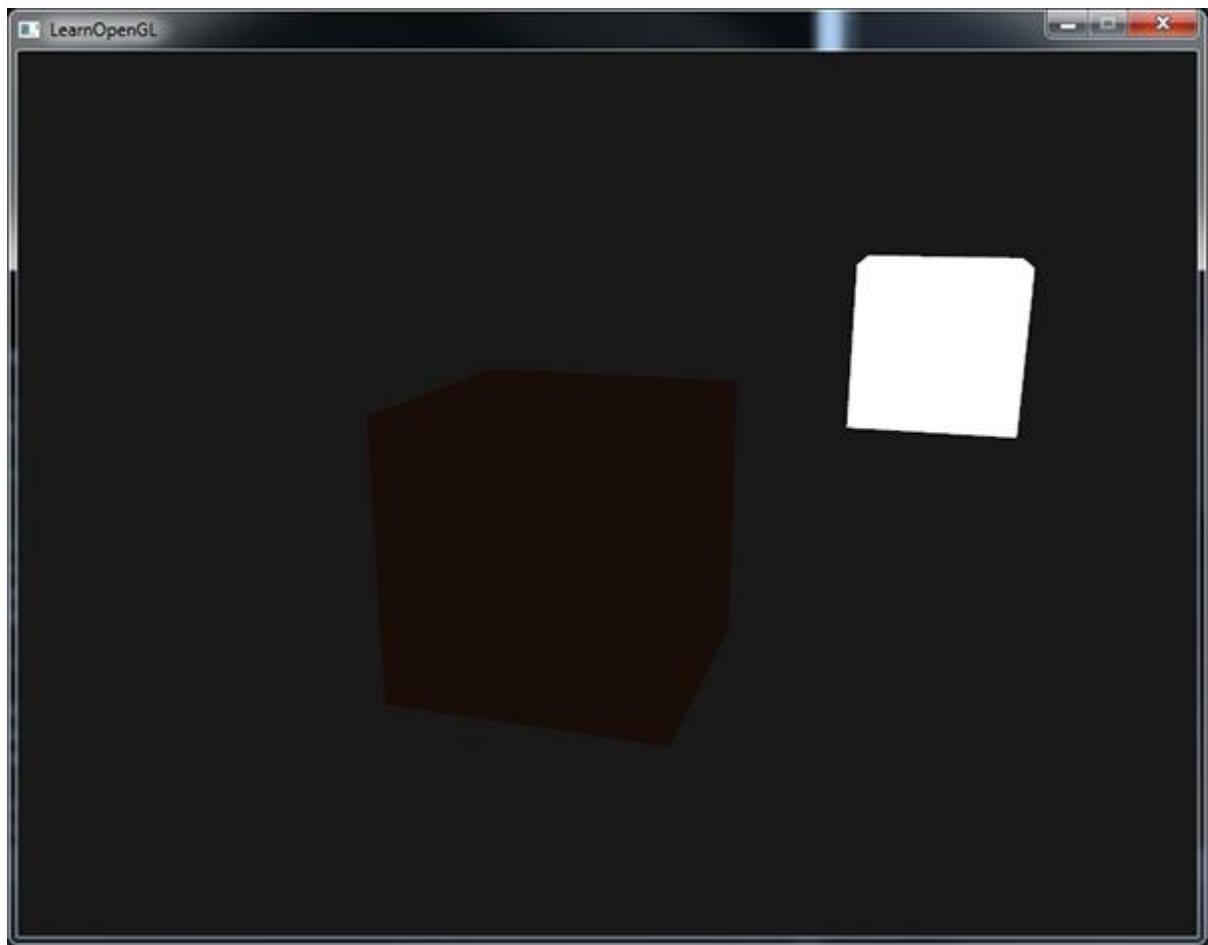
光通常都不是来自于同一光源，而是来自散落于我们周围的很多光源，即使它们可能并不是那么显而易见。光的一个属性是，它可以向很多方向发散和反弹，所以光最后到达的地点可能并不是它所临近的直射方向；光能够像这样**反射**(Reflect)到其他表面，一个物体的光照可能受到来自一个非直射的光源影响。考虑到这种情况的算法叫做**全局照明(Global Illumination)**算法，但是这种算法既开销高昂又极其复杂。

因为我们不是复杂和昂贵算法的死忠粉丝，所以我们将会使用一种简化的全局照明模型，叫做环境光照。如你在前面章节所见，我们使用一个(数值)很小的常量(光)颜色添加进物体片段(**Fragment**，指当前讨论的光线在物体上的照射点)的最终颜色里，这看起来就像即使没有直射光源也始终存在着一些发散的光。

把环境光添加到场景里非常简单。我们用光的颜色乘以一个(数值)很小常量环境因子，再乘以物体的颜色，然后使用它作为片段的颜色：

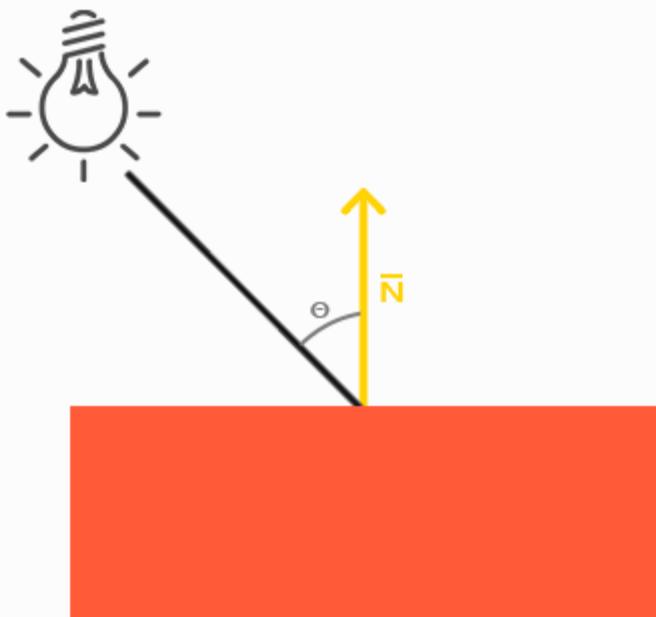
```
void main()
{
    float ambientStrength = 0.1f;
    vec3 ambient = ambientStrength * lightColor;
    vec3 result = ambient * objectColor;
    color = vec4(result, 1.0f);
}
```

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，但不是完全的黑暗，因为我们应用了环境光照(注意发光立方体没被环境光照影响是因为我们对它使用了另一个着色器)。它看起来应该像这样：



漫反射光照(Diffuse Lighting)

环境光本身不提供最明显的光照效果，但是漫反射光照会对物体产生显著的视觉影响。漫反射光使物体上与光线排布越近的片段越能从光源处获得更多的亮度。为了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与它所接触片段之间的角度。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种向量(这里以黄色箭头表示)，我们在后面再讲这个东西。两个向量之间的角度就能够根据点乘计算出来。

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角度越小，它们点乘的结果越倾向于 1。当两个向量的角度是 90 度的时候，点乘会变为 0。这同样适用于 θ ， θ 越大，光对片段颜色的影响越小。

Important

注意，我们使用的是单位向量(Unit Vector，长度是 1 的向量)取得两个向量夹角的余弦值，所以我们需要确保所有的向量都被标准化，否则点乘返回的值就不仅仅是余弦值了(如果你不明白，可以复习[变换](#)那一节的点乘部分)。

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，基于不同片段所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，我们需要些什么来计算漫反射光照？

- 法向量：一个垂直于顶点表面的向量。
- 定向的光线：作为光的位置和片段的位置之间的向量差的方向向量。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(Normal Vector)

法向量是垂直于顶点表面的(单位)向量。由于顶点自身并没有表面(它只是空间中一个独立的点)，我们利用顶点周围的顶点计算出这个顶点的表面。我们能够使用叉乘这个技巧为立方体所有的顶点计算出法线，但是由于 3D 立方体不是一个复杂的形状，所以我们可以简单的把法线数据手工添加到顶点数据中。更新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的各个面的表面的(一个立方体由 6 个面组成)。

因为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 position;  
  
layout (location = 1) in vec3 normal;  
  
...
```

现在我们已经向每个顶点添加了一个法向量，已经更新了顶点着色器，我们还要更新顶点属性指针(**Vertex Attribute Pointer**)。注意，发光物使用同样的顶点数组作为它的顶点数据，然而发光物的着色器没有使用新添加的法向量。我们不会更新发光物的着色器或者属性配置，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(GLfloat),  
                      (GLvoid * )0);  
  
glEnableVertexAttribArray(0);
```

我们只想使用每个顶点的前三个浮点数，并且我们忽略后三个浮点数，所以我们只需要把步长参数改成 `GLfloat` 尺寸的 6 倍就行了。

Important

发光物着色器顶点数据的不完全使用看起来有点低效，但是这些顶点数据已经从立方体对象载入到 GPU 的内存里了，所以 GPU 内存不是必须再储存新数据。相对于重新给发光物分配 VBO，实际上却是更高效了。

所有光照的计算需要在片段着色器里进行，所以我们需要把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(position, 1.0f);  
  
    Normal = normal;  
}
```

剩下要做的事情是，在片段着色器中定义相应的输入值：

```
in vec3 Normal;
```

计算漫反射光照

每个顶点现在都有了法向量，但是我们仍然需要光的位置向量和片段的位置向量。由于光的位置是一个静态变量，我们可以简单的在片段着色器中把它声明为 uniform：

```
uniform vec3 lightPos;
```

然后再游戏循环中(外面也可以，因为它不会变)更新 uniform。我们使用在前面教程中声明的 `lightPos` 向量作为光源位置：

```
GLint lightPosLoc = glGetUniformLocation(lightingShader.Program,  
                                         "lightPos");  
  
glUniform3f(lightPosLoc, lightPos.x, lightPos.y, lightPos.z);
```

最后，我们还需要片段的位置(**Position**)。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(**Model Matrix**,只用模型矩阵不需要用观察和投影矩阵)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，所以让我们就声明一个输出(**out**)变量，然后计算它的世界空间坐标：

```

out vec3 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);

    FragPos = vec3(model * vec4(position, 1.0f));
    Normal = normal;
}

```

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开始光照的计算了。

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，光的方向向量是光的位置向量与片段的位置向量之间的向量差。你可能记得，在变换教程中，我们简单的通过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，所以我们把法线和方向向量这个结果都进行标准化：

```

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

```

Important

当计算光照时我们通常不关心一个向量的“量”或它的位置，我们只关心它们的方向。所有的计算都使用单位向量完成，因为这会简化了大多数计算(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，这样它们才会保证自身为单位向量。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，来计算光对当前片段的实际的散射影响。结果值再乘以光的颜色，得到散射因子。两个向量之间的角度越大，散射因子就会越小：

```
float diff = max(dot(norm, lightDir), 0.0);
```

```
vec3 diffuse = diff * lightColor;
```

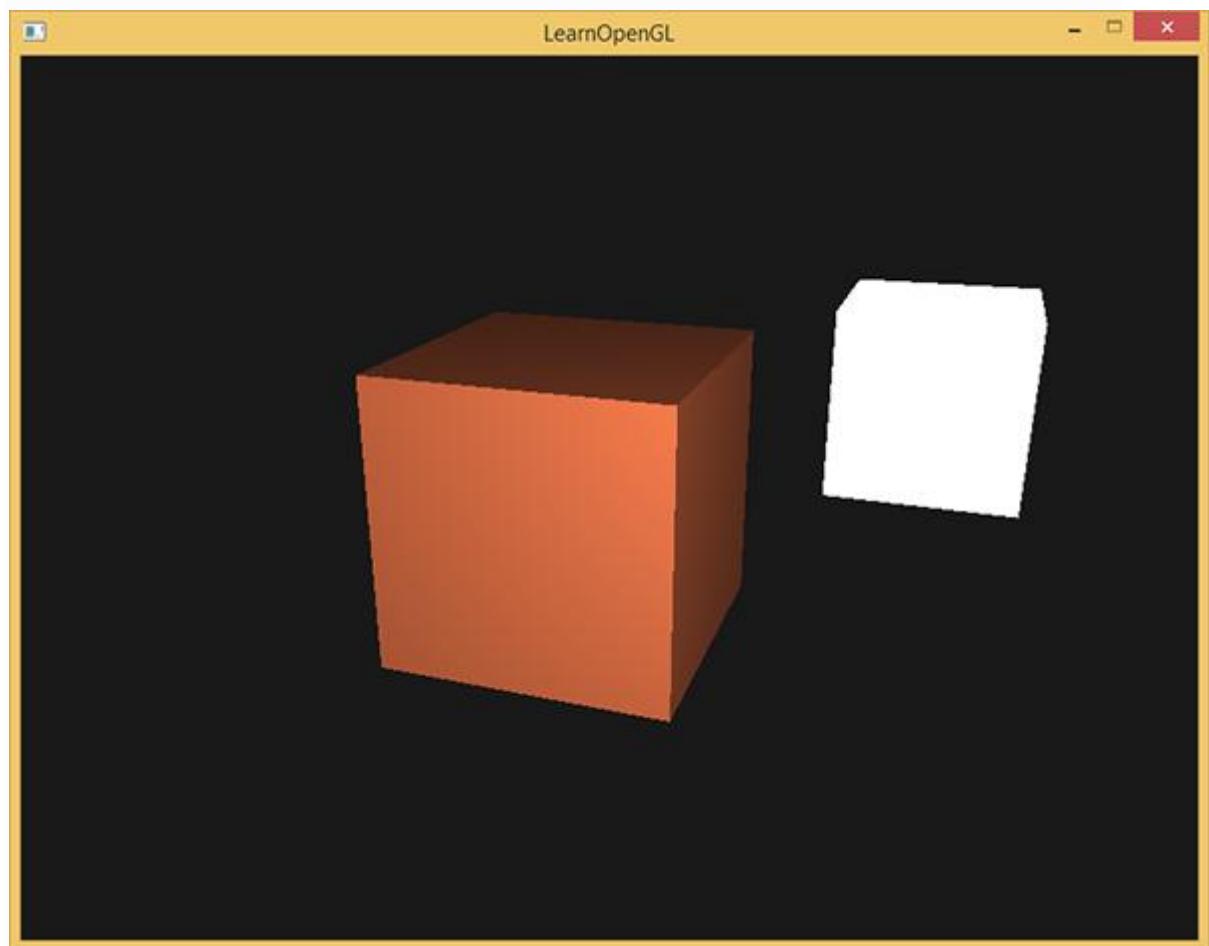
如果两个向量之间的角度大于 90 度，点乘的结果就会变成负数，这样会导致散射因子变为负数。为此，我们使用 `max` 函数返回两个参数之间较大的参数，从而保证散射因子不会变成负数。负数的颜色是没有实际定义的，所以最好避免它，除非你是那种古怪的艺术家。

既然我们有了一个环境光颜色和一个散射光颜色，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;
```

```
color = vec4(result, 1.0f);
```

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：



你可以看到使用了散射光照，立方体看起来就真的像个立方体了。尝试在你的脑中想象，通过移动正方体，法向量和光的方向向量之间的夹角增大，片段变得更暗。

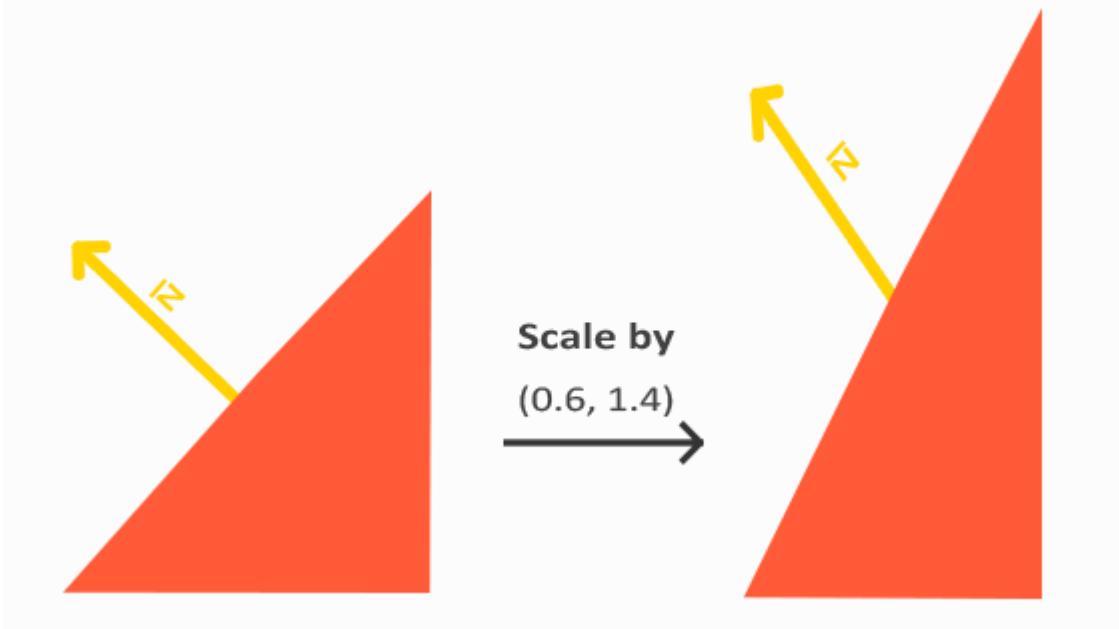
如果你遇到很多困难，可以对比[完整的源代码](#)以及[片段着色器代码](#)。

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，我们都是在世界空间坐标中进行计算的，所以，我们不是应该把法向量转换为世界空间坐标吗？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，平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，我们就要把模型矩阵左上角的 3×3 矩阵从模型矩阵中移除(译注：所谓移除就是设置为 0)，它是模型矩阵的平移部分(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；同样可以移除平移)。对于法向量，我们只能对它应用缩放(Scale)和旋转(Rotation)变换。

其次，如果模型矩阵执行了不等比缩放，法向量就不再垂直于表面了，顶点就会以这种方式被改变了。因此，我们不能用这样的模型矩阵去乘以法向量。下面的图展示了应用了不等比缩放的矩阵对法向量的影响：



无论何时当我们提交一个不等比缩放(注意：等比缩放不会破坏法线，因为法线的方向没被改变，而法线的长度很容易通过标准化进行修复)，法向量就不会再垂直于它们的表面了，这样光照会被扭曲。

修复这个行为的诀窍是使用另一个为法向量专门定制的模型矩阵。这个矩阵称之为正规矩阵(**Normal Matrix**)，它是进行了一点线性代数操作移除了对法向量的错误缩放效果。如果你想知道这个矩阵是如何计算出来的，我建议看[这个文章](#)。

正规矩阵被定义为“模型矩阵左上角的逆矩阵的转置矩阵”。真拗口，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，定义正规矩阵的大多资源就像应用到模型观察矩阵(**Model-view Matrix**)上的操作一样，但是由于我们只在世界空间工作(而不是在观察空间)，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` 函数自己生成正规矩阵，`inverse` 和 `transpose` 函数对所有类型矩阵都有效。注意，我们也要把这个被处理过的矩阵强制转换为 3×3 矩阵，这是为了保证它失去了平移属性，之后它才能乘以法向量。

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

在环境光照部分，光照表现没问题，这是因为我们没有对物体本身执行任何缩放操作，因而不是非得使用正规矩阵不可，用模型矩阵乘以法线也没错。可是，如果你进行了不等比缩放，使用正规矩阵去乘以法向量就是必不可少的了。

Attention

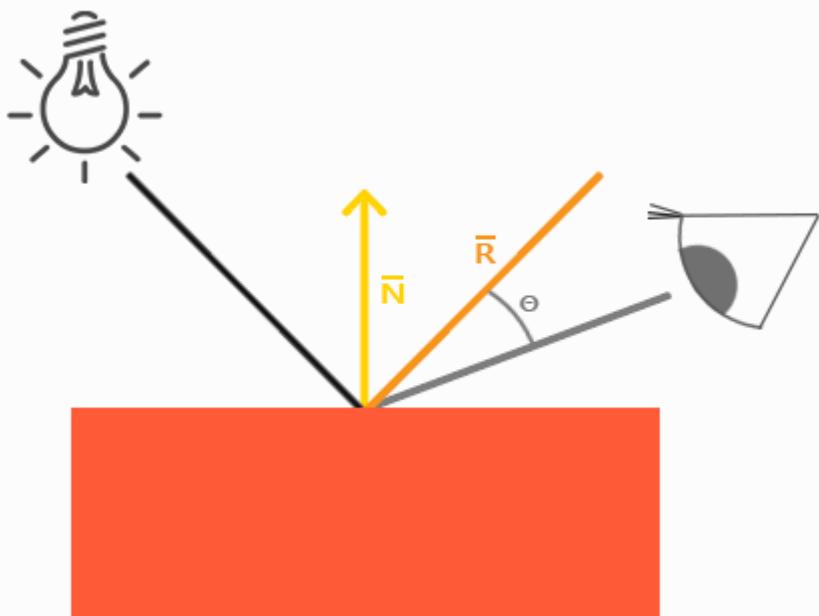
对于着色器来说，逆矩阵也是一种开销比较大的操作，因此，无论何时，在着色器中只要可能就应该尽量避免逆操作，因为它们必须为你场景中的每个顶点进行这样的处理。以学习的目的这样做很好，但是对于一个对于效率有要求的应用来说，在绘制之前，你最好用 **CPU** 计算出正规矩阵，然后通过 **uniform** 把值传递给着色器(和模型矩阵一样)。

镜面光照(**Specular Lighting**)

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。

和环境光照一样，镜面光照同样依据光的方向向量和物体的法向量，但是这次它也会依据观察方向，例如玩家是从什么方向看着这个片段的。镜面光照根据光的

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计算反射向量。然后我们计算反射向量和视线方向的角度，如果之间的角度越小，那么镜面光的作用就会越大。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的一个附加变量，我们可以使用观察者世界空间位置 (Viewer's World Space Position) 和片段的位置来计算。之后，我们计算镜面光亮度，用它乘以光的颜色，在用它加上作为之前计算的光照颜色。

Important

我们选择在世界空间(World Space)进行光照计算，但是大多数人趋向于在观察空间(View Space)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，我发现出于学习的目的，在世界空间计算光照更符合直觉。如果你仍然希望在视野空间计算光照的话，那就使用观察矩阵应用到所有相关的需要变换的向量(不要忘记，也要改变正规矩阵)。

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;
```

```
GLint viewPosLoc = glGetUniformLocation(lightingShader.Program,  
"viewPos");  
  
glUniform3f(viewPosLoc, camera.Position.x, camera.Position.y,  
camera.Position.z);
```

现在我们已经获得所有需要的变量，可以计算高光亮度了。首先，我们定义一个镜面强度(Specular Intensity)变量 `specularStrength`，给镜面高光一个中等亮度颜色，这样就不会产生过度的影响了。

```
float specularStrength = 0.5f;
```

如果我们把它设置为 `1.0f`，我们会得到一个对于珊瑚色立方体来说过度明亮的镜面亮度因子。下一节教程，我们会讨论所有这些光照射度的合理设置，以及它们是如何影响物体的。下一步，我们计算视线方向坐标，和沿法线轴的对应的反射坐标：

```
vec3 viewDir = normalize(viewPos - FragPos);
```

```
vec3 reflectDir = reflect(-lightDir, norm);
```

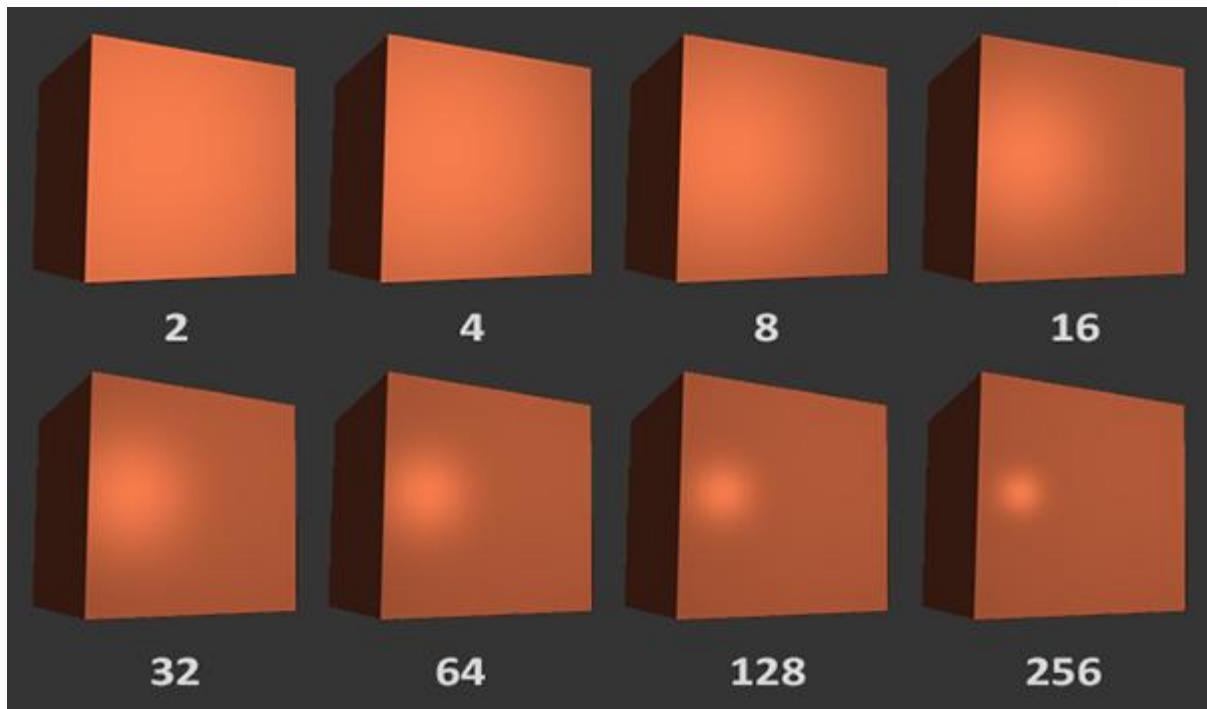
需要注意的是我们使用了 `lightDir` 向量的相反数。`reflect` 函数要求的第一个是从光源指向片段位置的向量，但是 `lightDir` 当前是从片段指向光源的向量(由先前我们计算 `lightDir` 向量时，(减数和被减数)减法的顺序决定)。为了保证我们得到正确的 `reflect` 坐标，我们通过 `lightDir` 向量的相反数获得它的方向的反向。第二个参数要求是一个法向量，所以我们提供的是已标准化的 `norm` 向量。

剩下要做的是计算镜面亮度分量。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(确保它不是负值)，然后得到它的 32 次幂。这个 32 是高光的发光值(Shininess)。一个物体的发光值越高，反射光的能力

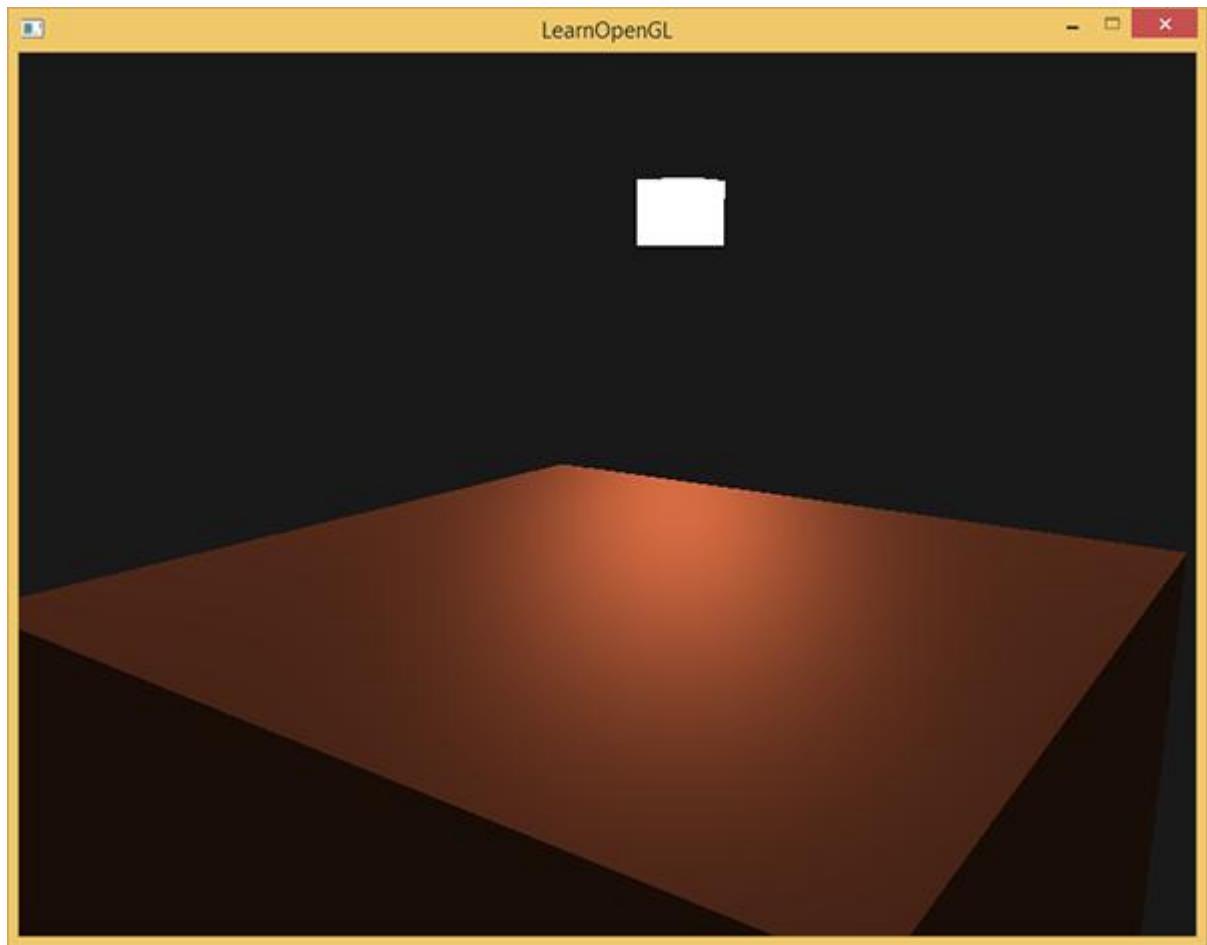
力越强，散射得越少，高光点越小。在下面的图片里，你会看到不同发光值对视觉(效果)的影响：



我们不希望镜面成分过于显眼，所以我们把指数设置为 32。剩下的最后一件事情是把它添加到环境光颜色和散射光颜色里，然后再乘以物体颜色：

```
vec3 result = (ambient + diffuse + specular) * objectColor;  
color = vec4(result, 1.0f);
```

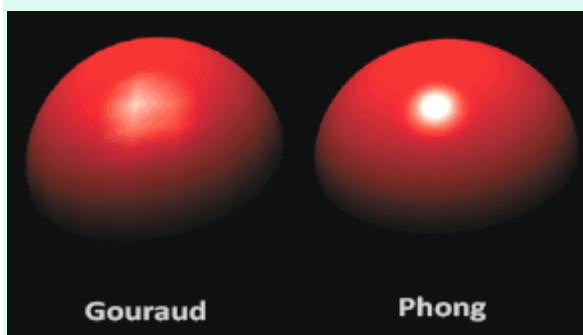
我们现在为冯氏光照计算了全部的光照元素。根据你的观察点，你可以看到类似下面的画面：



你可以[在这里找到完整源码](#)，在这里有[顶点](#)和[片段](#)着色器。

Important

早期的光照着色器，开发者在顶点着色器中实现冯氏光照。在顶点着色器中做这件事的优势是，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，顶点着色器中的颜色值是只是顶点的颜色值，片段的颜色值是它与周围的颜色值的插值。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，而不是冯氏着色。记住由于插值，这种光照连起来有点逊色。冯氏着色能产生更平滑的光照效果。

现在你可以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，影响到为我们所有物体的片段颜色。[下一个教程](#)，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，我们的光源时静止的，你可以尝试使用 `sin` 和 `cos` 函数让光源在场景中来回移动，此时再观察光照效果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、散射镜面强度，观察光照效果。改变镜面光照的 `shininess` 因子试试。
- 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** 光照来模拟冯氏光照，[参考解答](#)。

材质(Material)

原文	Materials
作者	JoeyDeVries
翻译	Django
校对	Geequlim

在真实世界里，每个物体会对光产生不同的反应。钢看起来比陶瓷花瓶更闪闪发光，一个木头箱子不会像钢箱子一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。有些物体不会散射(Scatter)很多光却会反射(Reflect)很多光，结果看起来就有一个较小的高光点(Highlight)，有些物体散射了很多，它们就会产生一个半径更大的高光。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材质(Material)属性。

在前面的教程中，我们指定一个物体和一个光的颜色来定义物体的图像输出，并使之结合环境(Ambient)和镜面强度(Specular Intensity)元素。当描述物体的时候，我们可以使用 3 种光照元素：环境光照(Ambient Lighting)、漫反射光照(Diffuse Lighting)、镜面光照(Specular Lighting)定义一个材质颜色。通过为每个元素指定一个颜色，我们已经对物体的颜色输出有了精密的控制。现在把一个镜面高光元素添加到这三个颜色里，这是我们所有的所有材质属性：

```
#version 330 core

struct Material

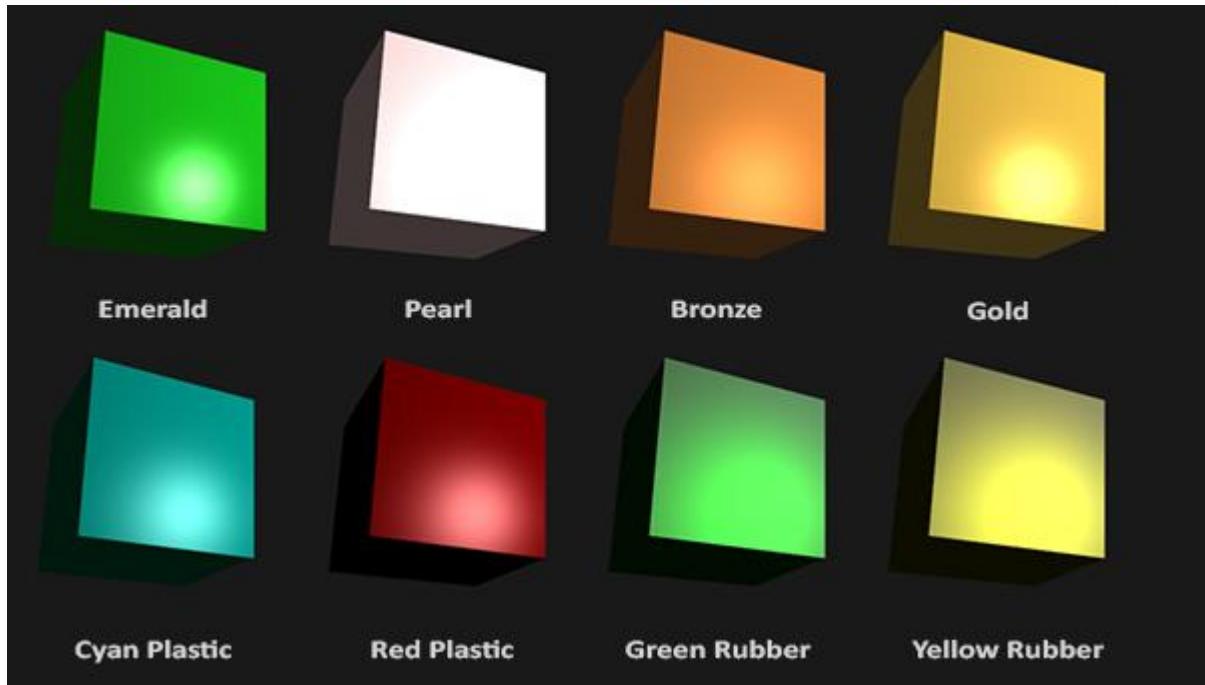
{
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

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可以更有条理。我们首先定义结构体的布局，然后简单声明一个 **uniform** 变量，以新创建的结构体作为它的类型。

就像你所看到的，我们为每个冯氏光照模型的元素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的是什么颜色；通常这是和物体颜色相同颜色。**diffuse** 材质向量定义了在漫反射光照下物体的颜色。漫反射颜色被设置为(和环境光照一样)我们需要的物体颜色。**specular** 材质向量设置的是物体受到的镜面光照的影响的颜色(或者可能是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真实世界的材质。这里有一个列表 devernay.free.fr 展示了几种材质属性，这些材质属性模拟外部世界的真实材质。下面的图片展示了几种真实世界材质对我们的立方体的影响：



如你所见，正确地指定一个物体的材质属性，似乎就是改变我们物体的相关属性的比例。效果显然很引人注目，但是对于大多数真实效果，我们最终需要更加复杂的形状，而不单单是一个立方体。在[下面的教程](#)中，我们会讨论更复杂的形状。

为一个物体赋予一款正确的材质是非常困难的，这需要大量实验和丰富的经验，所以由于错误的设置材质而毁了物体的画面质量是件经常发生的事。

让我们试试在着色器中实现这样的一个材质系统。

设置材质

我们在片段着色器中创建了一个 `uniform` 材质结构体，所以下面我们希望改变光照计算来顺应新的材质属性。由于所有材质元素都储存在结构体中，我们可以从 `uniform` 变量 `material` 取得它们：

```
void main()
{
    // 环境光
    vec3 ambient = lightColor * material.ambient;
```

```
// 漫反射光  
  
vec3 norm = normalize(Normal);  
  
vec3 lightDir = normalize(lightPos - FragPos);  
  
float diff = max(dot(norm, lightDir), 0.0);  
  
vec3 diffuse = lightColor * (diff * material.diffuse);
```

```
// 镜面高光  
  
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;  
  
color = vec4(result, 1.0f);  
  
}
```

你可以看到，我们现在获得所有材质结构体的属性，无论在哪儿都需要它们，这次通过材质颜色的帮助，计算结果输出的颜色。物体的每个材质属性都乘以它们对应的光照元素。

通过设置适当的 `uniform`，我们可以在应用中设置物体的材质。当设置 `uniform` 时，`GLSL` 中的一个结构体并不会被认为有什么特别之处。一个结构体值扮演 `uniform` 变量的封装体，所以如果我们希望填充这个结构体，我们就仍然必须设置结构体中的各个元素的 `uniform` 值，但是这次带有结构体名字作为前缀：

```
GLint matAmbientLoc = glGetUniformLocation(lightingShader.Program,
                                         "material.ambient");

GLint matDiffuseLoc = glGetUniformLocation(lightingShader.Program,
                                         "material.diffuse");

GLint matSpecularLoc = glGetUniformLocation(lightingShader.Program,
                                         "material.specular");

GLint matShineLoc = glGetUniformLocation(lightingShader.Program,
                                         "material.shininess");

glUniform3f(matAmbientLoc, 1.0f, 0.5f, 0.31f);

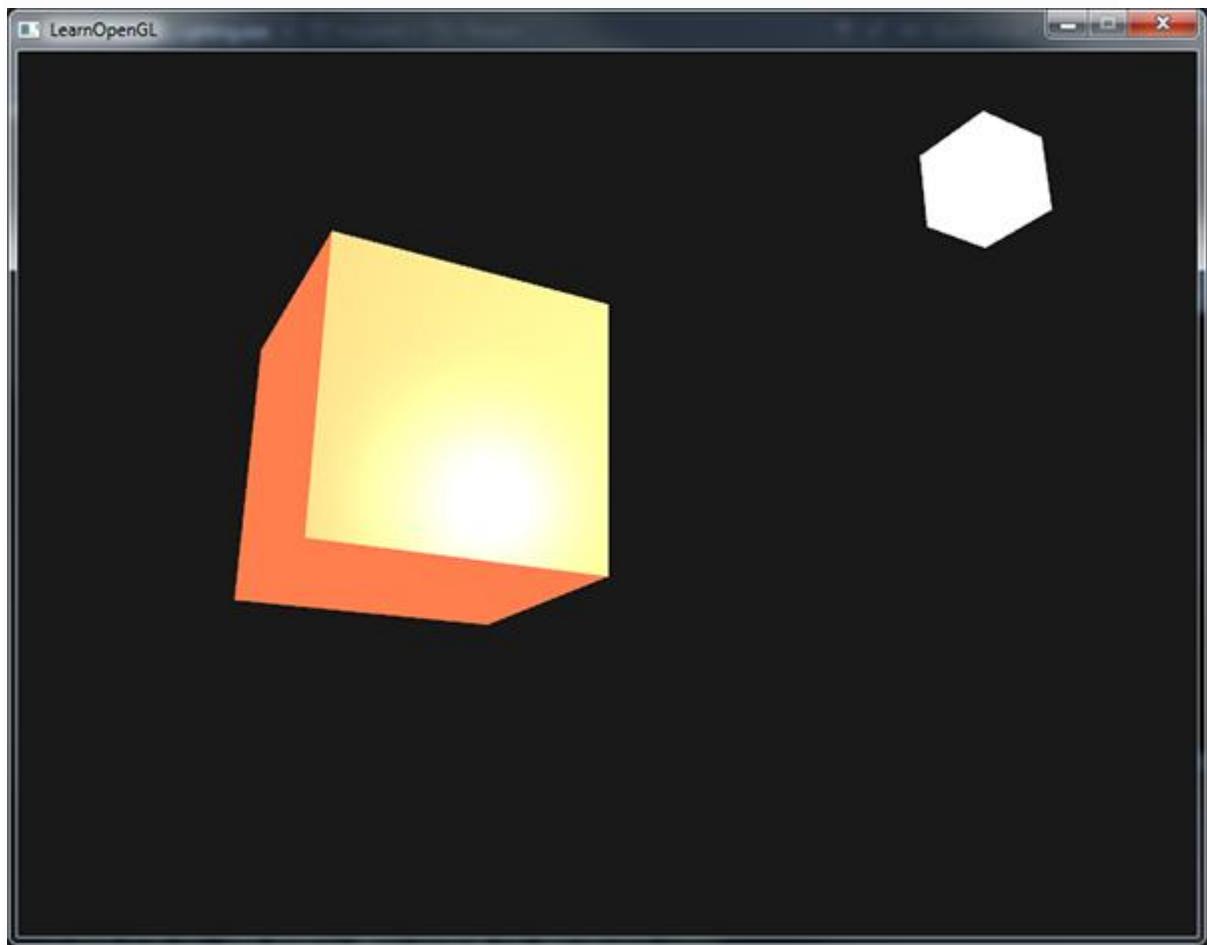
glUniform3f(matDiffuseLoc, 1.0f, 0.5f, 0.31f);

glUniform3f(matSpecularLoc, 0.5f, 0.5f, 0.5f);

glUniform1f(matShineLoc, 32.0f);
```

我们将 `ambient` 和 `diffuse` 元素设置成我们想要让物体所呈现的颜色，设置物体的 `specular` 元素为中等亮度颜色；我们不希望 `specular` 元素对这个指定物体产生过于强烈的影响。我们同样设置 `shininess` 为 32。我们现在可以简单的在应用中影响物体的材质。

运行程序，会得到下面这样的结果：



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、漫反射和镜面三个颜色任何一个光源都会去全力反射。光源对环境、漫反射和镜面元素同时具有不同的强度。前面的教程，我们通过使用一个强度值改变环境和镜面强度的方式解决了这个问题。我们想做一个相同的系统，但是这次为每个光照元素指定了强度向量。如果我们想象 `lightColor` 是 `vec3(1.0)`，代码看起来像是这样：

```
vec3 ambient = vec3(1.0f) * material.ambient;  
  
vec3 diffuse = vec3(1.0f) * (diff * material.diffuse);  
  
vec3 specular = vec3(1.0f) * (spec * material.specular);
```

所以物体的每个材质属性返回了每个光照元素的全强度。这些 `vec3(1.0)` 值可以各自独立的影响各个光源，这通常就是我们想要的。现在物体的 `ambient` 元素完

全地展示了立方体的颜色，可是环境元素不应该对最终颜色有这么大的影响，所以我们要设置光的 `ambient` 亮度为一个小一点的值，从而限制环境色：

```
vec3 result = vec3(0.1f) * material.ambient;
```

我们可以用同样的方式影响光源 `diffuse` 和 `specular` 的强度。这和我们[前面教程](#)所做的极为相似；你可以说我们已经创建了一些光的属性来各自独立地影响每个光照元素。我们希望为光的属性创建一些与材质结构体相似的东西：

```
struct Light  
{  
    vec3 position;  
    vec3 ambient;  
    vec3 diffuse;  
    vec3 specular;  
};  
  
uniform Light light;
```

一个光源的 `ambient`、`diffuse` 和 `specular` 光都有不同的亮度。环境光通常设置为一个比较低的亮度，因为我们不希望环境色太过显眼。光源的 `diffuse` 元素通常设置为我们希望光所具有的颜色；经常是一个明亮的白色。`specular` 元素通常被设置为 `vec3(1.0f)` 类型的全强度发光。要记住的是我们同样把光的位置添加到结构体中。

就像材质 `uniform` 一样，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);
```

然后我们要在应用里设置光的亮度：

```
GLint lightAmbientLoc = glGetUniformLocation(lightingShader.Program,  
"light.ambient");
```

```
GLint lightDiffuseLoc = glGetUniformLocation(lightingShader.Program,  
"light.diffuse");
```

```
GLint lightSpecularLoc = glGetUniformLocation(lightingShader.Program,  
"light.specular");
```

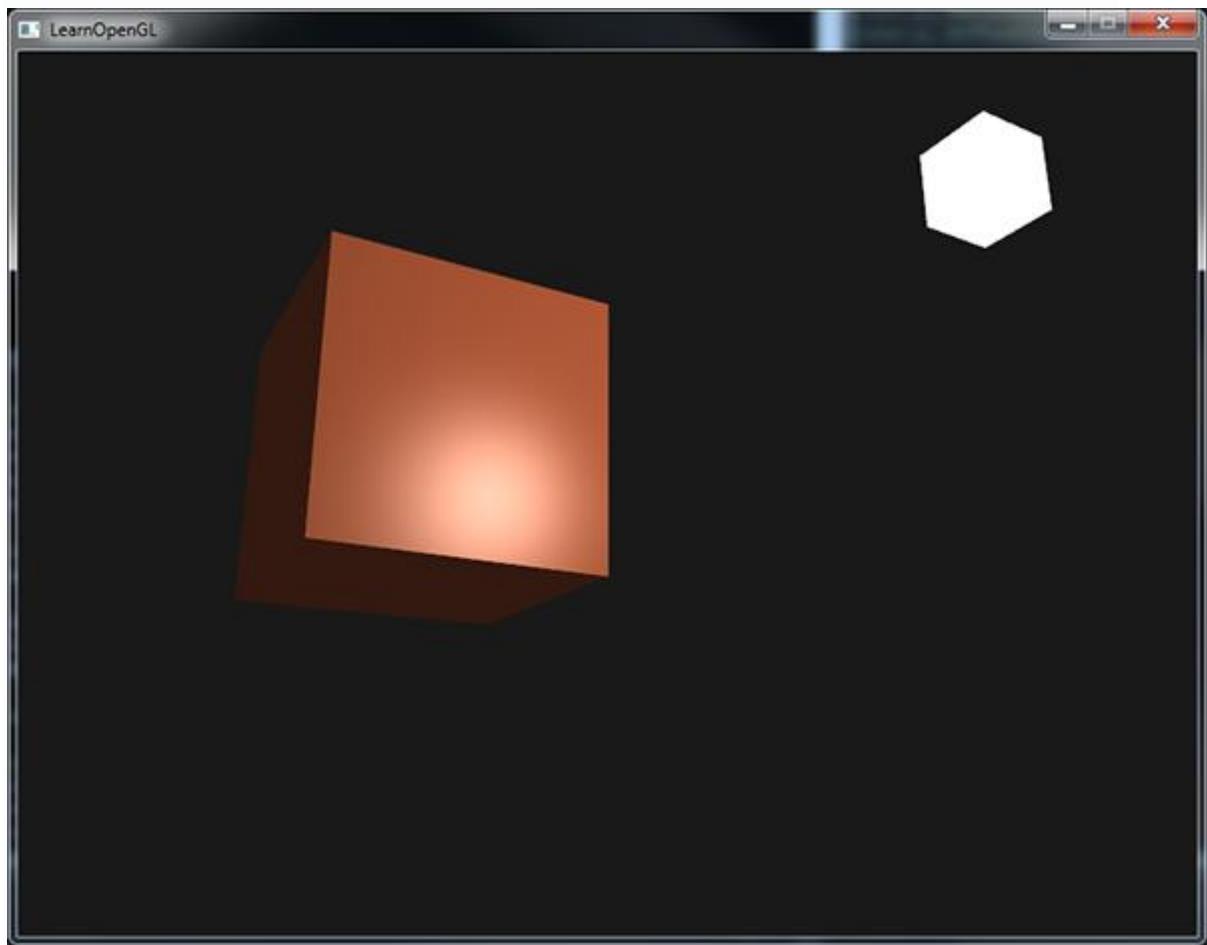
```
glUniform3f(lightAmbientLoc, 0.2f, 0.2f, 0.2f);
```

```
glUniform3f(lightDiffuseLoc, 0.5f, 0.5f, 0.5f); // 让我们把这个光调暗
```

一点，这样会看起来更自然

```
glUniform3f(lightSpecularLoc, 1.0f, 1.0f, 1.0f);
```

现在，我们调整了光是如何影响物体所有的材质的，我们得到一个更像前面教程的视觉输出。这次我们完全控制了物体光照和材质：



现在改变物体的外观相对简单了些。我们做点更有趣的事！

不同的光源颜色

目前为止，我们使用光源的颜色仅仅去改变物体各个元素的强度(通过选用从白到灰到黑范围内的颜色)，并没有影响物体的真实颜色(只是强度)。由于现在能够非常容易地访问光的属性了，我们可以随着时间改变它们的颜色来获得一些有意思的效果。由于所有东西都已经在片段着色器做好了，改变光的颜色很简单，我们可以立即创建出一些有趣的效果：

如你所见，不同光的颜色极大地影响了物体的颜色输出。由于光的颜色直接影响物体反射的颜色(你可能想起在颜色教程中有讨论过)，它对视觉输出有显著的影响。

利用 `sin` 和 `glfwGetTime` 改变光的环境和漫反射颜色，我们可以随着时间流逝简单的改变光源颜色：

```
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);
```

```
glm::vec3 ambientColor = diffuseColor * glm::vec3(0.2f);
```

```
glUniform3f(lightAmbientLoc, ambientColor.x, ambientColor.y,
```

```
ambientColor.z);
```

```
glUniform3f(lightDiffuseLoc, diffuseColor.x, diffuseColor.y,
```

```
diffuseColor.z);
```

尝试和实验使用这些光照和材质值，看看它们怎样影响图像输出的。你可以从这里找到[程序的源码](#)，[片段着色器](#)的源码。

练习

- 你能像我们教程一开始那样根据一些材质的属性来模拟一个真实世界的物体吗？注意[材质表](#)中的环境光颜色与漫反射光的颜色可能不一样，因为他们并没有把光照强度考虑进去来模拟，你需要将光照颜色的强度改为[vec\(1.0f\)](#)来输出正确的结果：[参考解答](#)，我做了一个青色(Cyan)的塑料箱子

光照贴图

原文	Lighting maps
作者	JoeyDeVries
翻译	Django
校对	Geequlim , BLumia

前面的教程，我们讨论了让不同的物体拥有各自不同的材质并对光照做出不同的反应的方法。在一个光照场景中，让每个物体拥有和其他物体不同的外观很棒，但是这仍然不能对一个物体的图像输出提供足够多的灵活性。

前面的教程中我们将一个物体自身作为一个整体为其定义了一个材质，但是现实世界的物体通常不会只有这么一种材质，而是由多种材质组成。想象一辆车：它的外表质地光亮，车窗会部分反射环境，它的轮胎没有 **specular** 高光，轮毂却非常闪亮（在洗过之后）。汽车同样有 **diffuse** 和 **ambient** 颜色，它们在整个车上都不相同；一辆车显示了多种不同的 **ambient/diffuse** 颜色。总之，这样一个物体每个部分都有多种材质属性。

所以，前面的材质系统对于除了最简单的模型以外都是不够的，所以我们需要扩展前面的系统，我们要介绍 **diffuse** 和 **specular** 贴图。它们允许你对一个物体的 **diffuse**（而对于简洁的 **ambient** 成分来说，它们几乎总是是一样的）和 **specular** 成分能够有更精确的影响。

漫反射贴图

我们希望通过某种方式对每个原始像素独立设置 **diffuse** 颜色。有可以让我们基于物体原始像素的位置来获取颜色值的系统吗？

这可能听起来极其相似，坦白来讲我们使用这样的系统已经有一段时间了。听起来很像在一个[之前的教程](#)中谈论的**纹理**，它基本就是一个纹理。我们其实是使用同一个潜在原则下的不同名称：使用一张图片覆盖住物体，以便我们为每个原始像素索引独立颜色值。在光照场景中，通过纹理来呈现一个物体的 **diffuse** 颜色，这个做法被称做**漫反射贴图(Diffuse texture)**(因为 3D 建模师就是这么称呼这个做法的)。

为了演示漫反射贴图，我们将会使用[下面的图片](#)，它是一个有一圈钢边的木箱：



在着色器中使用漫反射贴图和纹理教程介绍的一样。这次我们把纹理以 `sampler2D` 类型储存在 `Material` 结构体中。我们使用 `diffuse` 贴图替代早期定义的 `vec3` 类型的 `diffuse` 颜色。

Attention

要记住的是 `sampler2D` 也叫做模糊类型，这意味着我们不能以某种类型对它实例化，只能用 `uniform` 定义它们。如果我们用结构体而不是 `uniform` 实例化（就像函数的参数那样），`GLSL` 会抛出奇怪的错误；这同样也适用于其他模糊类型。我们也要移除 `ambient` 材质颜色向量，因为 `ambient` 颜色绝大多数情况等于 `diffuse` 颜色，所以不需要分别去储存它：

```
struct Material
```

```
{
```

```
sampler2D diffuse;  
  
vec3 specular;  
  
float shininess;  
  
};  
  
...  
  
in vec2 TexCoords;
```

Important

如果你非把 `ambient` 颜色设置为不同的值不可（不同于 `diffuse` 值），你可以继续保留 `ambient` 的 `vec3`，但是整个物体的 `ambient` 颜色会继续保持不变。为了使每个原始像素得到不同 `ambient` 值，你需要对 `ambient` 值单独使用另一个纹理。

注意，在片段着色器中我们将会再次需要纹理坐标，所以我们声明一个额外输入变量。然后我们简单地从纹理采样，来获得原始像素的 `diffuse` 颜色值：

```
vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse,  
  
TexCoords));
```

同样，不要忘记把 `ambient` 材质的颜色设置为 `diffuse` 材质的颜色：

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse,  
  
TexCoords));
```

这就是 `diffuse` 贴图的全部内容了。就像你看到的，这不是什么新的东西，但是它却极大提升了视觉品质。为了让它工作，我们需要用到纹理坐标更新顶点数据，把它们作为顶点属性传递到片段着色器，把纹理加载并绑定到合适的纹理单元。

更新的顶点数据可以从[这里](#)找到。顶点数据现在包括了顶点位置，法线向量和纹理坐标，每个立方体的顶点都有这些属性。让我们更新顶点着色器来接受纹理坐标作为顶点属性，然后发送到片段着色器：

```
#version 330 core
```

```
layout (location = 0) in vec3 position;  
layout (location = 1) in vec3 normal;  
layout (location = 2) in vec2 texCoords;  
...  
out vec2 TexCoords;  
  
void main()  
{  
    ...  
    TexCoords = texCoords;  
}
```

要保证更新的顶点属性指针，不仅是 VAO 匹配新的顶点数据，也要把箱子图片加载为纹理。在绘制箱子之前，我们希望首选纹理单元被赋为 `material.diffuse` 这个 `uniform` 采样器，并绑定箱子的纹理到这个纹理单元：

```
glUniform1i(glGetUniformLocation(lightingShader.Program,  
    "material.diffuse"), 0);  
...  
glActiveTexture(GL_TEXTURE0);  
glBindTexture(GL_TEXTURE_2D, diffuseMap);
```

现在，使用一个 `diffuse` 贴图，我们在细节上再次获得惊人的提升，这次添加到箱子上的光照开始闪光了（名符其实）。你的箱子现在可能看起来像这样：

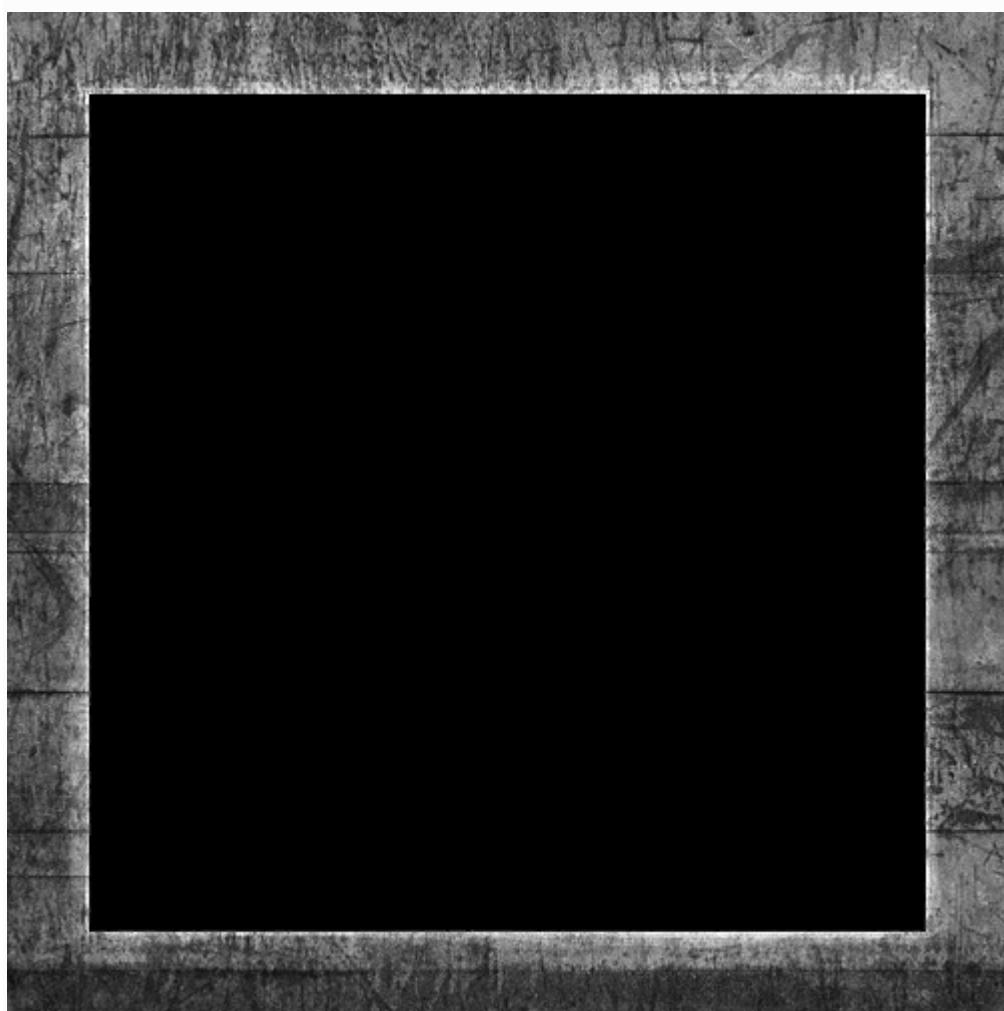


你可以在这里得到应用的[全部代码](#)。

镜面贴图

你可能注意到，**specular** 高光看起来不怎么样，由于我们的物体是个箱子，大部分是木头，我们知道木头是不应该有镜面高光的。我们通过把物体设置 **specular** 材质设置为 `vec3(0.0f)` 来修正它。但是这样意味着铁边会不再显示镜面高光，我们知道钢铁是会显示一些镜面高光的。我们会想要控制物体部分地显示镜面高光，它带有修改了的亮度。这个问题看起来和 **diffuse** 贴图的讨论一样。这是巧合吗？我想不是。

我们同样用一个纹理贴图，来获得镜面高光。这意味着我们需要生成一个黑白（或者你喜欢的颜色）纹理来定义 **specular** 亮度，把它应用到物体的每个部分。下面是一个 [specular 贴图](#)的例子：



一个 **specular** 高光的亮度可以通过图片中每个纹理的亮度来获得。**specular** 贴图的每个像素可以显示为一个颜色向量，比如：在那里黑色代表颜色向量 `vec3(0.0f)`，灰色是 `vec3(0.5f)`。在片段着色器中，我们采样相应的颜色值，把它乘以光的 **specular** 亮度。像素越“白”，乘积的结果越大，物体的 **specular** 部分越亮。

由于箱子几乎是由木头组成，木头作为一个材质不会有镜面高光，整个木头部分的 **diffuse** 纹理被用黑色覆盖：黑色部分不会包含任何 **specular** 高光。箱子的铁边有一个修改的 **specular** 亮度，它自身更容易受到镜面高光影响，木纹部分则不会。

从技术上来讲，木头也有镜面高光，尽管这个闪亮值很小（更多的光被散射），影响很小，但是为了学习目的，我们可以假装木头不会有任何 **specular** 光反射。

使用 Photoshop 或 Gimp 之类的工具，通过将图片进行裁剪，将某部分调整成黑白图样，并调整亮度/对比度的做法，可以非常容易将一个 **diffuse** 纹理贴图处理为 **specular** 贴图。

镜面贴图采样

一个 **specular** 贴图和其他纹理一样，所以代码和 **diffuse** 贴图的代码也相似。确保合理的加载了图片，生成一个纹理对象。由于我们在同样的片段着色器中使用另一个纹理采样器，我们必须为 **specular** 贴图使用一个不同的纹理单元(参见[纹理](#))，所以在渲染前让我们把它绑定到合适的纹理单元

```
glUniform1i(glGetUniformLocation(lightingShader.Program,  
"material.specular"), 1);  
...  
glActiveTexture(GL_TEXTURE1);  
glBindTexture(GL_TEXTURE_2D, specularMap);
```

然后更新片段着色器材质属性，接受一个 **sampler2D** 作为这个 **specular** 部分的类型，而不是 **vec3**:

```
struct Material  
{  
    sampler2D diffuse;  
    sampler2D specular;  
    float shininess;  
};
```

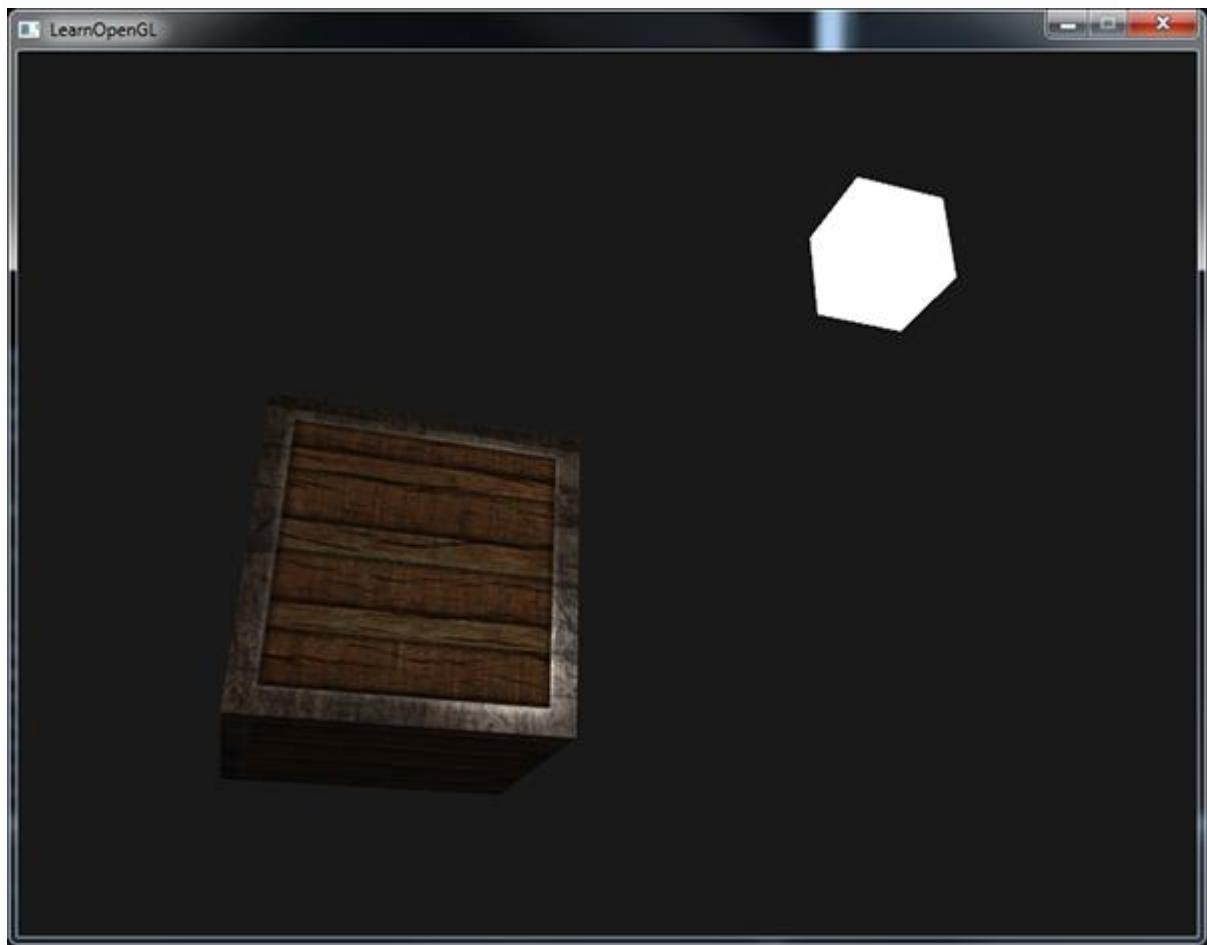
最后我们希望采样这个 **specular** 贴图，来获取原始像素相应的 **specular** 亮度：

```
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));  
  
color = vec4(ambient + diffuse + specular, 1.0f);
```

通过使用一个 **specular** 贴图我们可以定义极为精细的细节，物体的这个部分会获得闪亮的属性，我们可以设置它们相应的亮度。**specular** 贴图给我们一个附加的高于 **diffuse** 贴图的控制权限。

如果你不想成为主流，你可以在 **specular** 贴图里使用颜色，不单单为每个原始像素设置 **specular** 亮度，同时也设置 **specular** 高光的颜色。从真实角度来说，**specular** 的颜色基本是由光源自身决定的，所以它不会生成真实的图像（这就是为什么图片通常是黑色和白色的：我们只关心亮度）。

如果你现在运行应用，你可以清晰地看到箱子的材质现在非常类似真实的铁边的木头箱子了：



你可以在这里找到[全部源码](#)。也对比一下你的[顶点着色器](#)和[片段着色器](#)。

使用 `diffuse` 和 `specular` 贴图，我们可以给相关但简单物体添加一个极为明显的细节。我们可以使用其他纹理贴图，比如法线/`bump` 贴图或者反射贴图，给物体添加更多的细节。但是这些在后面教程才会涉及。把你的箱子给你所有的朋友和家人看，有一天你会很满足，我们的箱子会比现在更漂亮！

练习

- 调整光源的 `ambient`, `diffuse` 和 `specular` 向量值，看看它们如何影响实际输出的箱子外观。
- 尝试在片段着色器中反转镜面贴图(Specular Map)的颜色值，然后木头就会变得反光而边框不会反光了（由于贴图中钢边依然有一些残余颜色，所以钢边依然会有一些高光，不过反光明显小了很多）。[参考解答](#)
- 使用漫反射纹理(Diffuse Texture)原本的颜色而不是黑白色来创建镜面贴图，并观察，你会发现结果显得并不那么真实了。如果你不会处理图片，你可以使用这个[带颜色的镜面贴图](#)。[最终效果](#)

- 添加一个叫做**放射光贴图(Emission Map)**的东西，即记录每个片段发光值(Emission Value)大小的贴图，发光值是(模拟)物体自身发光(Emit)时可能产生的颜色。这样的话物体就可以忽略环境光自身发光。通常在你看到游戏里某个东西(比如[机器人的](#)眼,或是[箱子上的小灯](#))在发光时，使用的就是放射光贴图。使用[这个](#)贴图(作者为 [creativesam](#))作为放射光贴图并使用在箱子上，你就会看到箱子上有会发光的字了。[参考解答](#),[片段着色器](#), [最终效果](#)

投光物

原文	Light casters
作者	JoeyDeVries
翻译	Django
校对	Geequlim

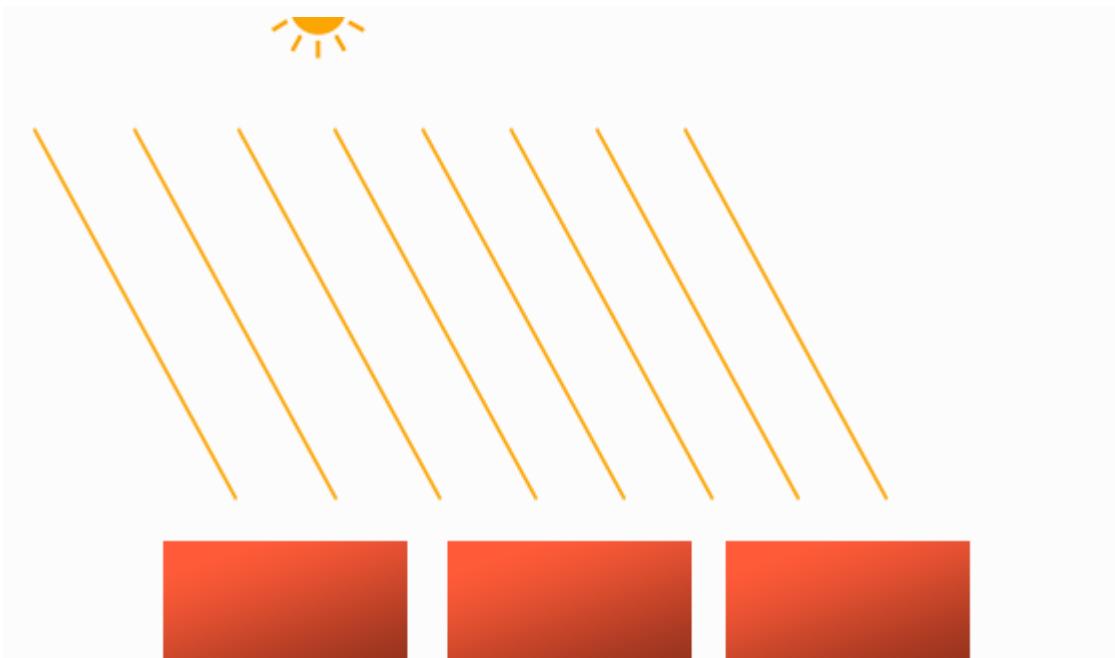
我们目前使用的所有光照都来自于一个单独的光源，这是空间中的一个点。它的效果不错，但是在真实世界，我们有多种类型的光，它们每个表现都不同。一个光源把光投射到物体上，叫做投光。这个教程里我们讨论几种不同的投光类型。学习模拟不同的光源是你未来丰富你的场景的另一个工具。

我们首先讨论定向光(directional light)，接着是作为之前学到知识的扩展的点光(point light)，最后我们讨论聚光灯(Spotlight)。下面的教程我们会把这几种不同的光类型整合到一个场景中。

定向光(Directional Light)

当一个光源很远的时候，来自光源的每条光线接近于平行。这看起来就像所有的光线来自于同一个方向，无论物体和观察者在哪儿。当一个光源被设置为无限远时，它被称为定向光(也被成为平行光)，因为所有的光线都有着同一个方向；它会独立于光源的位置。

我们知道的定向光源的一个好例子是，太阳。太阳和我们不是无限远，但它也足够远了，在计算光照的时候，我们感觉它就像无限远。在下面的图片里，来自于太阳的所有的光线都被定义为平行光：



因为所有的光线都是平行的，对于场景中的每个物体光的方向都保持一致，物体和光源的位置保持怎样的关系都无所谓。由于光的方向向量保持一致，光照计算会和场景中的其他物体相似。

我们可以通过定义一个光的方向向量，来模拟这样一个定向光，而不是使用光的位置向量。着色器计算保持大致相同的要求，这次我们直接使用光的方向向量来代替用 `lightDir` 向量和 `position` 向量的计算：

```
struct Light
{
    // vec3 position; // 现在不在需要光源位置了，因为它无限远的

    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

...
```

```
void main()
{
    vec3 lightDir = normalize(-light.direction);

    ...
}
```

注意，我们首先忽略 `light.direction` 向量。目前我们使用的光照计算需要光的方向作为一个来自片段朝向的光源的方向，但是人们通常更习惯定义一个定向光作为一个全局方向，它从光源发出。所以我们必须忽略全局光的方向向量来改变它的方向；它现在是一个方向向量指向光源。同时，确保对向量进行标准化处理，因为假定输入的向量就是一个单位向量是不明智的。

作为结果的 `lightDir` 向量被使用在 `diffuse` 和 `specular` 计算之前。

为了清晰地强调一个定向光对所有物体都有同样的影响，我们再次访问[坐标系教程](#)结尾部分的箱子场景。例子里我们先定义 10 个不同的箱子位置，为每个箱子生成不同的模型矩阵，每个模型矩阵包含相应的本地到世界变换：

```
for(GLuint i = 0; i < 10; i++)
{
    model = glm::mat4();

    model = glm::translate(model, cubePositions[i]);

    GLfloat angle = 20.0f * i;

    model = glm::rotate(model, angle, glm::vec3(1.0f, 0.3f, 0.5f));

    glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
        glm::value_ptr(model));

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

同时，不要忘记定义光源的方向（注意，我们把方向定义为：从光源处发出的方向；在下面，你可以快速看到光的方向的指向）：

```
GLint lightDirPos = glGetUniformLocation(lightingShader.Program,  
"light.direction");  
  
glUniform3f(lightDirPos, -0.2f, -1.0f, -0.3f);
```

Important

我们已经把光的位置和方向向量传递为 `vec3`，但是有些人去想更喜欢把所有的向量设置为 `vec4`. 当定义位置向量为 `vec4` 的时候，把 `w` 元素设置为 `1.0` 非常重要，这样平移和投影才会合理的被应用。然而，当定义一个方向向量为 `vec4` 时，我们并不想让平移发挥作用（因为它们除了代表方向，其他什么也不是）所以我们把 `w` 元素设置为 `0.0`。

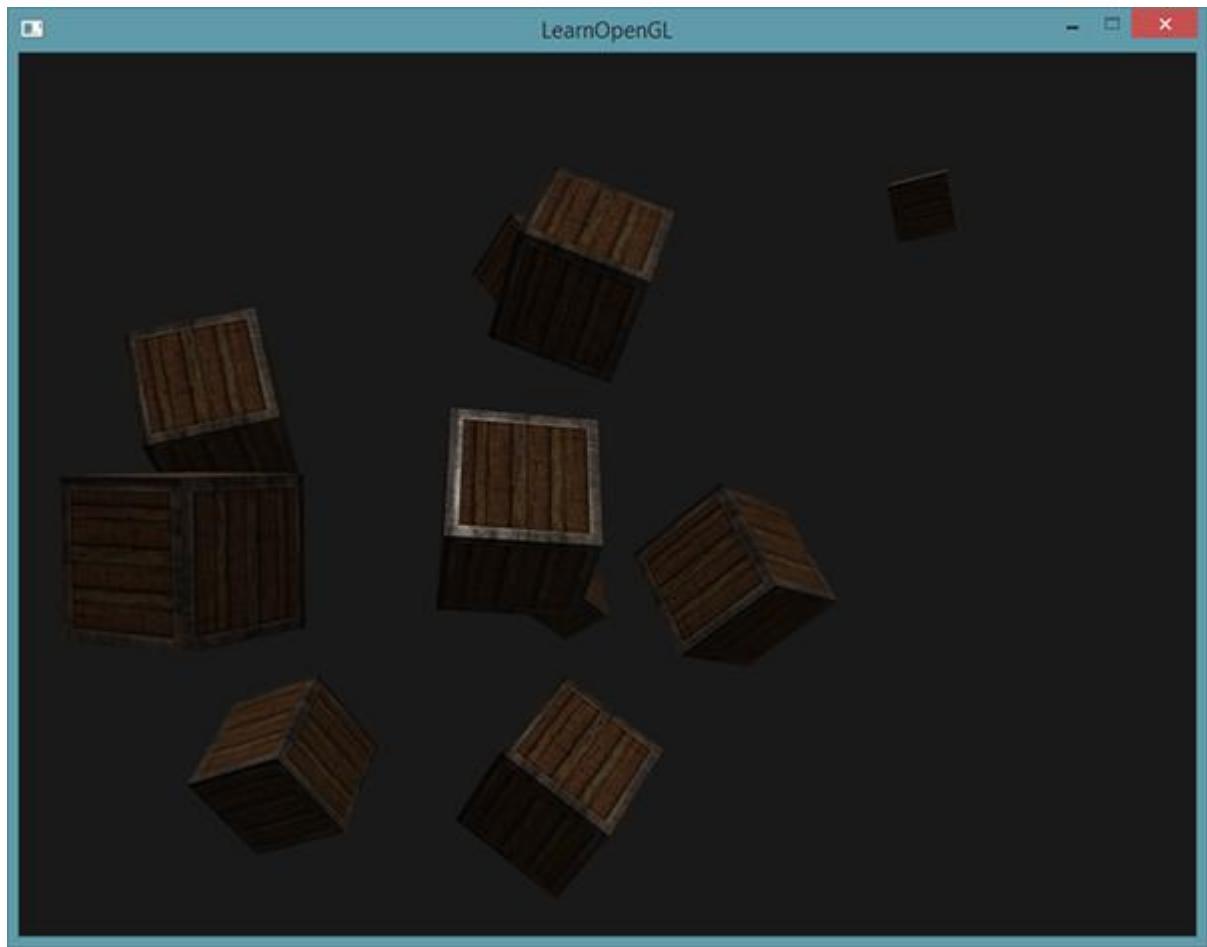
方向向量被表示为：`vec4(0.2f, 1.0f, 0.3f, 0.0f)`。这可以作为简单检查光的类型的方法：你可以检查 `w` 元素是否等于 `1.0`，查看我们现在所拥有的光的位置向量，`w` 是否等于 `0.0`，我们有一个光的方向向量，所以根据那个调整计算方法：

```
```c++ if(lightVector.w == 0.0) // 请留意浮点数错误 // 执行定向光照计算  

else if(lightVector.w == 1.0) // 像上一个教程一样执行顶点光照计算 ```
```

有趣的事：这就是旧 `OpenGL`（固定函数式）决定一个光源是一个定向光还是位置光源，更具这个修改它的光照。

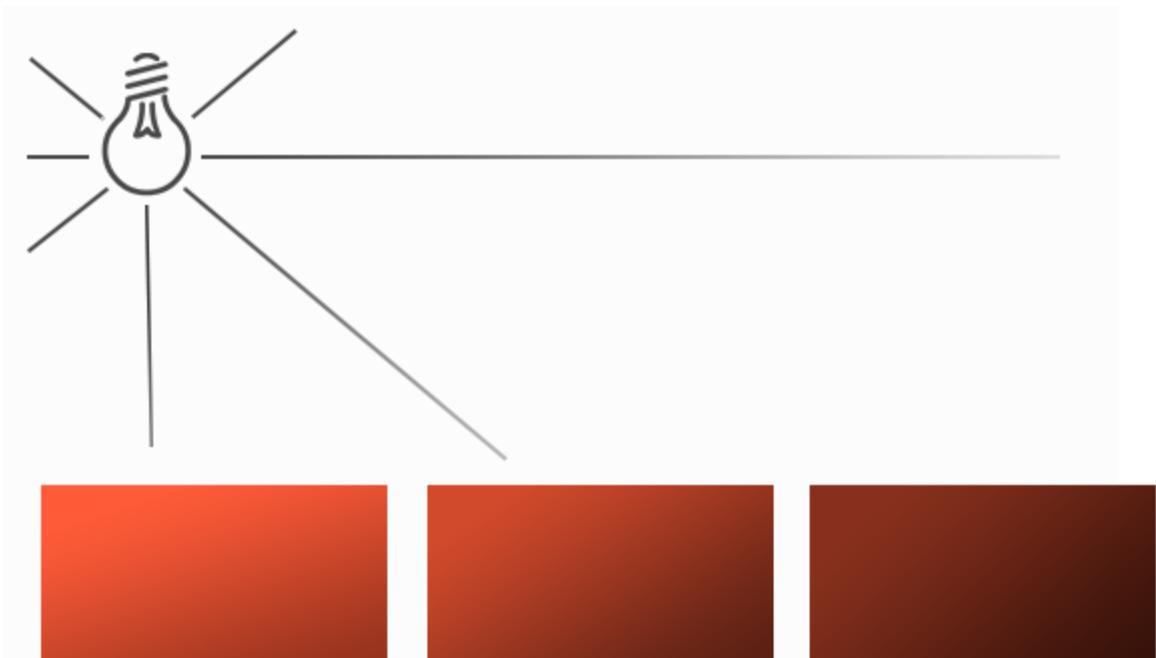
如果你现在编译应用，飞跃场景，它看起来像有一个太阳一样的光源，把光抛到物体身上。你可以看到 `diffuse` 和 `specular` 元素都对该光源进行反射了，就像天空上有一个光源吗？看起来就像这样：



你可以在这里获得[应用的所有代码](#)，这里是[顶点](#)和[片段](#)着色器代码。

## 定点光(**Point Light**)

定向光作为全局光可以照亮整个场景，这非常棒，但是另一方面除了定向光，我们通常也需要几个定点光，在场景里发亮。点光是一个在时间里有位置的光源，它向所有方向发光，光线随距离增加逐渐变暗。想象灯泡和火炬作为投光物，它们可以扮演点光的角色。



之前的教程我们已经使用了（最简单的）点光。我们有一个有位置的光源，它从自身的位置向所有方向发出光线。然而，这个我们定义的光源所模拟光线的从不会衰减，这使得它看起来光源亮度极强。在大多数 3D 模拟中，我们喜欢模拟一个能照亮一个周围确定区域光源，但它不会照亮整个场景。

如果你把 10 个箱子添加到之前教程的光照场景中，你会注意到黑暗中的每个箱子都会有同样的亮度，就像箱子在光照的前面；没有公式定义光的距离衰减。我们想让黑暗中与光源比较近的箱子被轻微地照亮。

## 衰减(Attenuation)

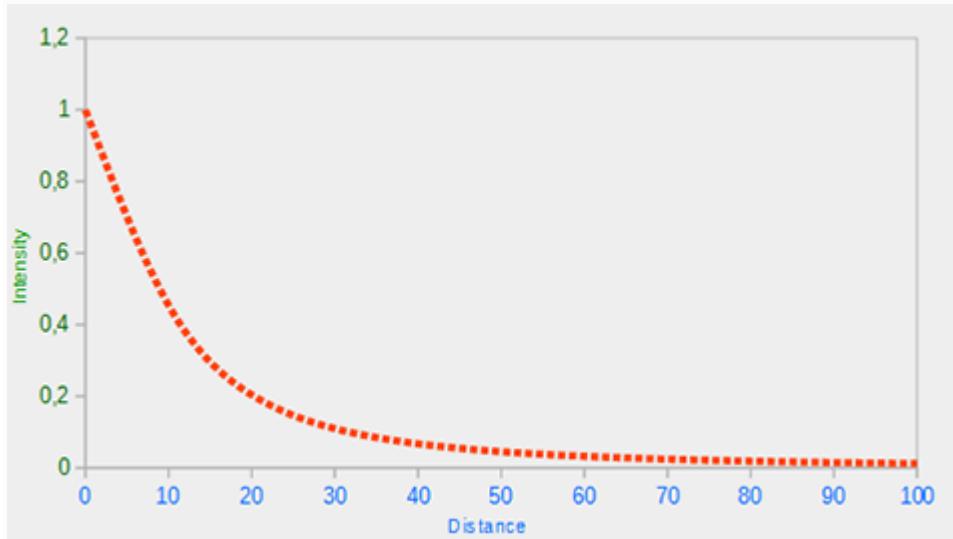
随着光线穿越更远的距离相应地减少亮度，通常被称为衰减(Attenuation)。一种随着距离减少亮度的方式是使用线性等式。这样的一个随着距离减少亮度的线性方程，可以使远处的物体更暗。然而，这样的线性方程效果会有点假。在真实世界，通常光在近处时非常亮，但是一个光源的亮度，开始的时候减少的非常快，之后随着距离的增加，减少的速度会慢下来。我们需要一种不同的方程来减少光的亮度。

幸运的是一些聪明人已经早就把它想到了。下面的方程把一个片段的光的亮度除以一个已经计算出来的衰减值，这个值根据光源的远近得到：

$$F_{att} = \frac{I}{K_c + K_l * d + K_q * d^2}$$

在这里  $I$  是当前片段的光的亮度， $d$  代表片段到光源的距离。为了计算衰减值，我们定义 3 个项：常数项  $K_c$ ，一次项  $K_l$  和二次项  $K_q$ 。

常数项通常是 1.0，它的作用是保证坟墓永远不会比 1 小，因为它可以利用一定的距离增加亮度，这个结果不会影响到我们所寻找的。一次项用于与距离值相称，这回以线性的方式减少亮度。二次项用于与距离的平方相乘，为光源设置一个亮度的二次递减。二次项在距离比较近的时候相比一次项会比一次项更小，但是当距离更远的时候比一次项更大。由于二次项的光会以线性方式减少，指导距离足够大的时候，就会超过一次项，之后，光的亮度会减少的更快。最后的效果就是光在近距离时，非常亮，但是距离变远亮度迅速降低，最后亮度降低速度再次变慢。下面的图展示了在 100 以内的范围，这样的衰减效果。



你可以看到当距离很近的时候光有最强的亮度，但是随着距离增大，亮度明显减弱，大约接近 100 的时候，就会慢下来。这就是我们想要的。

### 选择正确的值

但是，我们把这三个项设置为什么值呢？正确的值的设置由很多因素决定：环境、你希望光所覆盖的距离范围、光的类型等。大多数场合，这是经验的问题，也要适度调整。下面的表格展示一些各项的值，它们模拟现实（某种类型的）光源，

覆盖特定的半径（距离）。第一蓝定义一个光的距离，它覆盖所给定的项。这些值是大多数光的良好开始，它是来自 Ogre3D 的维基的礼物：

<b>Distance</b>	<b>Constant</b>	<b>Linear</b>	<b>Quadratic</b>
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

就像你所看到的，常数项  $K_c$  一直都是 1.0。一次项  $K_l$  为了覆盖更远的距离通常很小，二次项  $K_q$  就更小了。尝试用这些值进行实验，看看它们在你的实现中各自的效果。我们的环境中，32 到 100 的距离对大多数光通常就足够了。

## 实现衰减

为了实现衰减，在着色器中我们会需要三个额外数值：也就是公式的常量、一次项和二次项。最好把它们储存在之前定义的 **Light** 结构体中。要注意的是我们计算 `lightDir`，就是在前面的教程中我们所做的，不是像之前的定向光的那部分。

```
struct Light
```

```
{
```

```
 vec3 position;

 vec3 ambient;

 vec3 diffuse;

 vec3 specular;

 float constant;

 float linear;

 float quadratic;

};
```

然后，我们在 OpenGL 中设置这些项：我们希望光覆盖 50 的距离，所以我们会使用上面的表格中合适的常数项、一次项和二次项：

```
glUniform1f(glGetUniformLocation(lightingShader.Program,
 "light.constant"), 1.0f);

glUniform1f(glGetUniformLocation(lightingShader.Program,
 "light.linear"), 0.09);

glUniform1f(glGetUniformLocation(lightingShader.Program,
 "light.quadratic"), 0.032);
```

在片段着色器中实现衰减很直接：我们根据公式简单的计算衰减值，在乘以 `ambient`、`diffuse` 和 `specular` 元素。

我们需要光源的距离提供给公式；记得我们能够计算向量的长度吗？我们可以通过获取片段和光源之间的不同向量把向量的长度结果作为距离项。我们可以使用 GLSL 的内建 `length` 函数做这件事：

```
float distance = length(light.position - Position);
```

```
float attenuation = 1.0f / (light.constant + light.linear*distance
+light.quadratic*(distance*distance));
```

然后，我们在光照计算中，通过把衰减值乘以 `ambient`、`diffuse` 和 `specular` 颜色，包含这个衰减值。

### Important

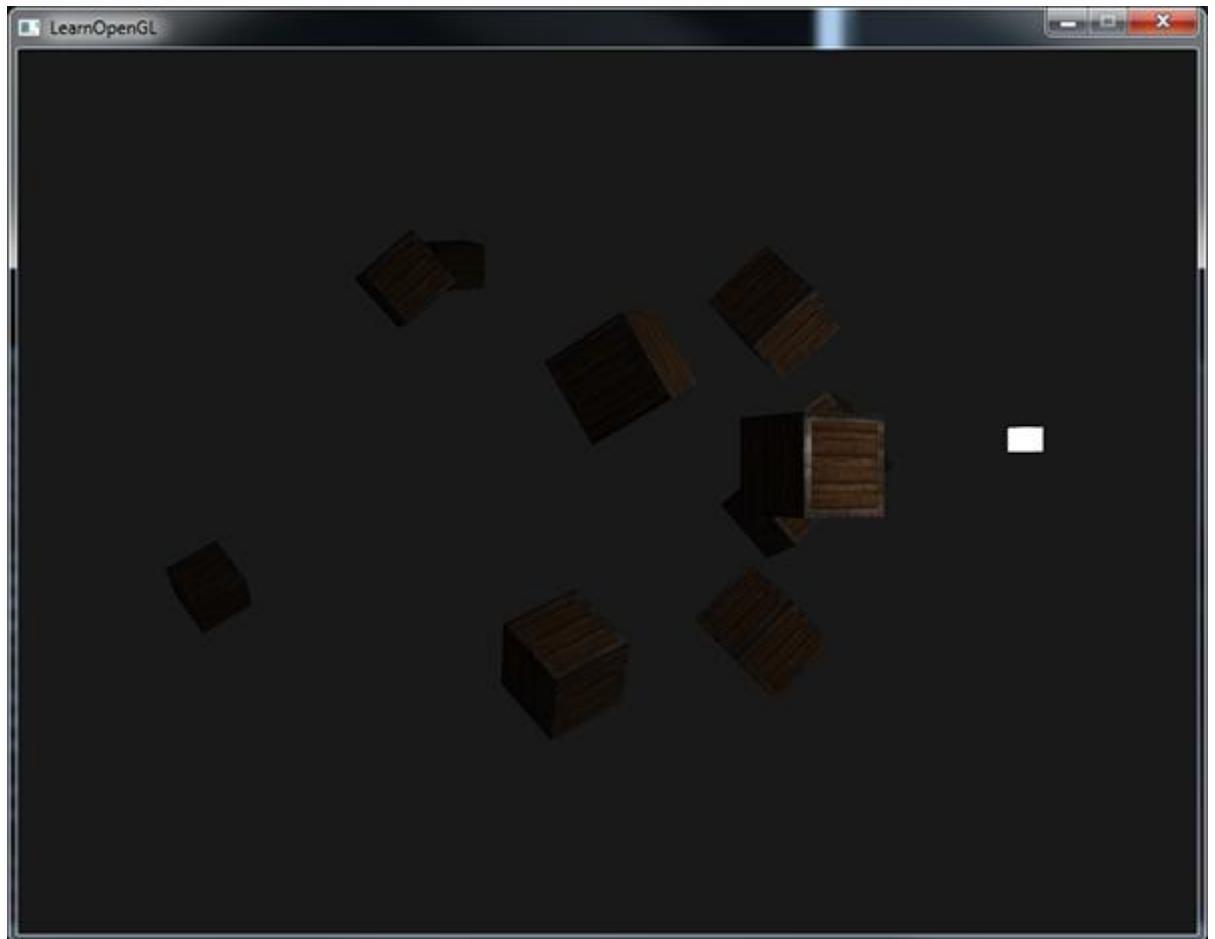
我们可以可以把 `ambient` 元素留着不变，这样 `ambient` 光照就不会随着距离减少，但是如果我们使用多余 1 个的光源，所有的 `ambient` 元素会开始叠加，因此这种情况，我们希望 `ambient` 光照也衰减。简单的调试出对于你的环境来说最好的效果。

```
ambient *= attenuation;

diffuse *= attenuation;

specular *= attenuation;
```

如果你运行应用后获得这样的效果：



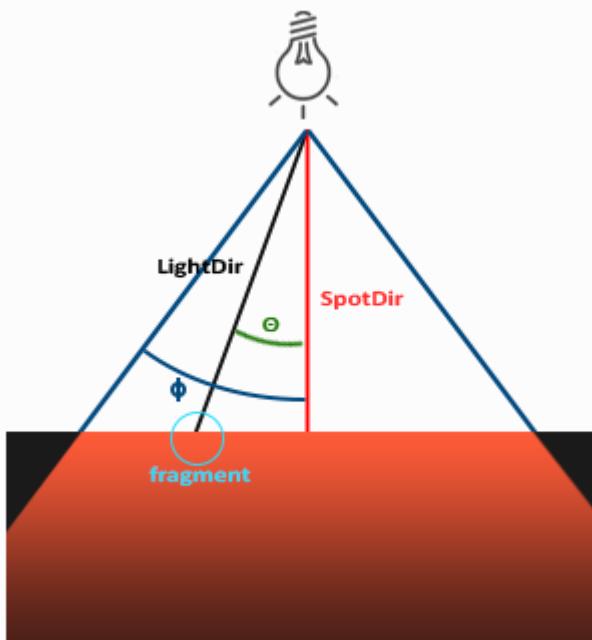
你可以看到现在只有最近处的箱子的前面被照得最亮。后面的箱子一点都没被照亮，因为它们距离光源太远了。你可以在这里找到[应用源码](#)和[片段着色器](#)的代码。

定点光就是一个可配置的位置和衰减值应用到光照计算中。还有另一种类型光可用于我们照明库当中。

## 聚光灯(Spotlight)

我们要讨论的最后一种类型光是聚光灯(Spotlight)。聚光灯是一种位于环境中某处的光源，它不是向所有方向照射，而是只朝某个方向照射。结果是只有一个聚光灯照射方向的确定半径内的物体才会被照亮，其他的都保持黑暗。聚光灯的好例子是路灯或手电筒。

OpenGL 中的聚光灯用世界空间位置，一个方向和一个指定了聚光灯半径的切光角来表示。我们计算的每个片段，如果片段在聚光灯的切光方向之间（就是在圆锥体内），我们就会把片段照亮。下面的图可以让你明白聚光灯是如何工作的：



- **LightDir**: 从片段指向光源的向量。
- **SpotDir**: 聚光灯所指向的方向。
- **Phi $\phi$** : 定义聚光灯半径的切光角。每个落在这个角度之外的，聚光灯都不会照亮。

- $\Theta$ : `LightDir` 向量和 `SpotDir` 向量之间的角度。 $\Theta$  值应该比  $\phi$  值小，这样才会在聚光灯内。

所以我们大致要做的是，计算 `LightDir` 向量和 `SpotDir` 向量的点乘（返回两个单位向量的点乘，还记得吗？），然后和遮光角  $\phi$  对比。现在你应该明白聚光灯是我们下面将创建的手电筒的范例。

## 手电筒

手电筒是一个坐落在观察者位置的聚光灯，通常瞄准玩家透视图的前面。基本上说，一个手电筒是一个普通的聚光灯，但是根据玩家的位置和方向持续的更新它的位置和方向。

所以我们需要为片段着色器提供的值，是聚光灯的位置向量（来计算光的方向坐标），聚光灯的方向向量和遮光角。我们可以把这些值储存在 `Light` 结构体中：

```
struct Light
{
 vec3 position;
 vec3 direction;
 float cutOff;
 ...
};
```

下面我们把这些适当的值传给着色器：

```
glUniform3f(lightPosLoc, camera.Position.x, camera.Position.y,
 camera.Position.z);
glUniform3f(lightSpotdirLoc, camera.Front.x, camera.Front.y,
 camera.Front.z);
```

```
glUniform1f(lightSpotCutOffLoc, glm::cos(glm::radians(12.5f)));
```

你可以看到，我们为遮光角设置一个角度，但是我们根据一个角度计算了余弦值，把这个余弦结果传给了片段着色器。这么做的原因是在片段着色器中，我们计算 `LightDir` 和 `SpotDir` 向量的点乘，而点乘返回一个余弦值，不是一个角度，所以我们不能直接把一个角度和余弦值对比。为了获得这个角度，我们必须计算点乘结果的反余弦，这个操作开销是很大的。所以为了节约一些性能，我们先计算给定切光角的余弦值，然后把结果传递给片段着色器。由于每个角度都被表示为余弦了，我们可以直接对比它们，而不用进行任何开销高昂的操作。

现在剩下要做的是计算  $\theta$  值，用它和  $\phi$  值对比，以决定我们是否在或不在聚光灯的内部：

```
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，使得场景不至于完全黑暗
```

```
color =
```

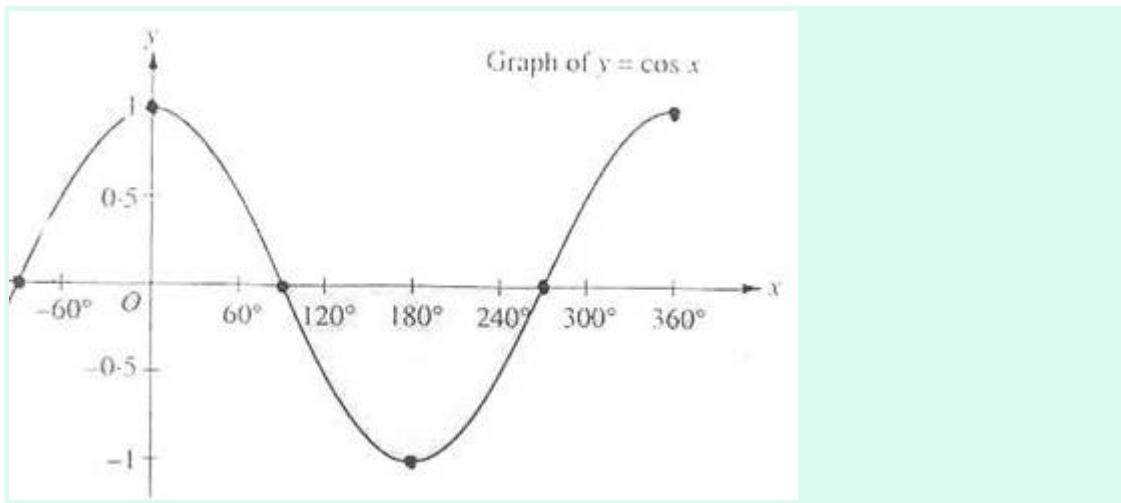
```
vec4(light.ambient*vec3(texture(material.diffuse, TexCoords)),
```

```
1.0f);
```

我们首先计算 `lightDir` 和负方向向量的点乘（它负的是因为我们想要向量指向光源，而不是从光源作为指向出发点。译注：前面的 `specular` 教程中作者却用了相反的表示方法，这里读者可以选择喜欢的表达方式）。确保对所有相关向量进行了标准化处理。

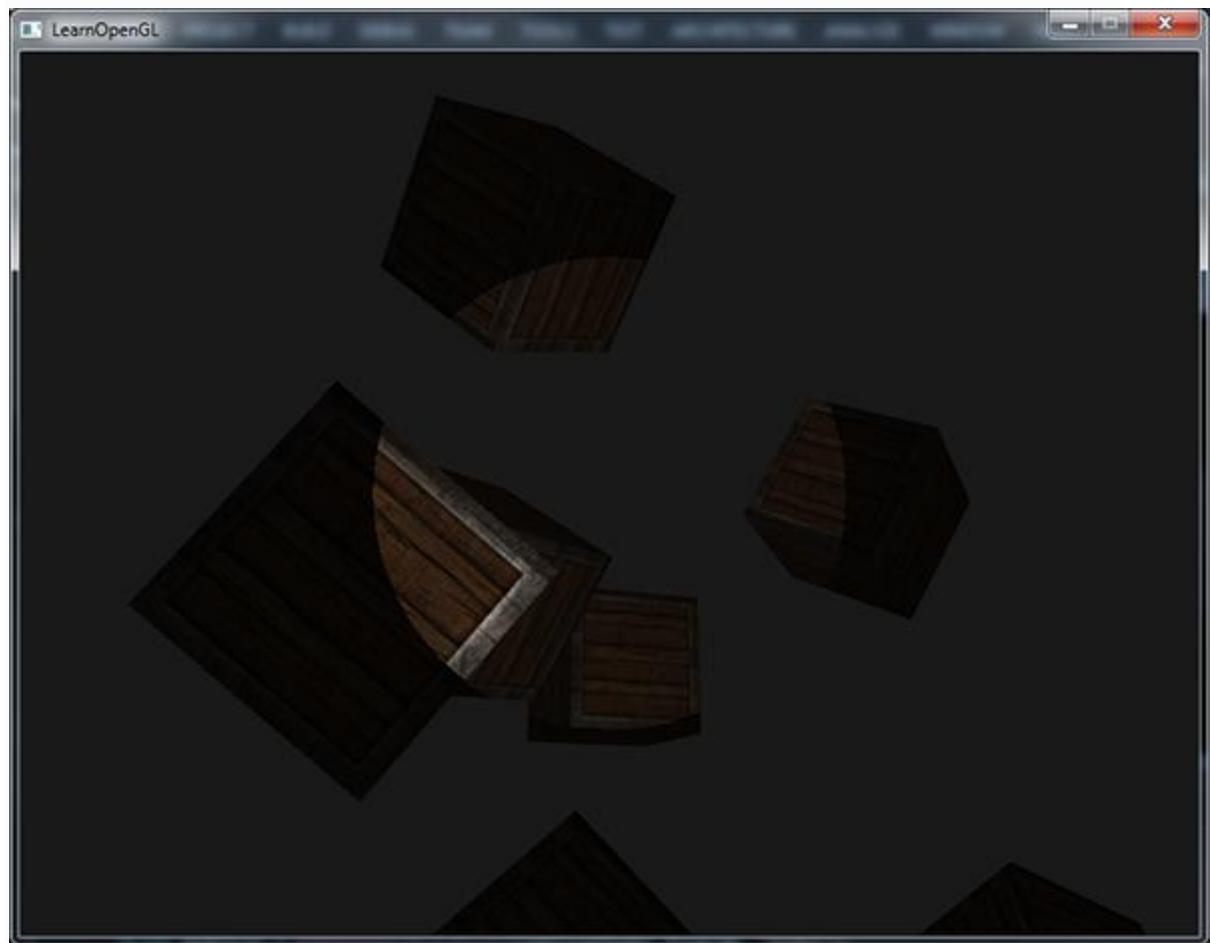
## Important

你可能奇怪为什么 `if` 条件中使用`>`符号而不是`<`符号。为了在聚光灯以内， $\theta$  不是应该比光的遮光值更小吗？这没错，但是不要忘了，角度值是以余弦值来表示的，一个 0 度的角表示为 1.0 的余弦值，当一个角是 90 度的时候被表示为 0.0 的余弦值，你可以在这里看到：



现在你可以看到，余弦越是接近 1.0，角度就越小。这就解释了为什么  $\theta$  需要比切光值更大了。切光值当前被设置为 12.5 的余弦，它等于 0.9978，所以  $\theta$  的余弦值在 0.9979 和 1.0 之间，片段会在聚光灯内，被照亮。

运行应用，在聚光灯内的片段才会被照亮。这看起来像这样：



你可以在这里获得[全部源码](#)和[片段着色器的源码](#)。

它看起来仍然有点假，原因是聚光灯有了一个硬边。片段着色器一旦到达了聚光灯的圆锥边缘，它就立刻黑了下来，却没有任何平滑减弱的过度。一个真实的聚光灯的光会在它的边界处平滑减弱的。

## 平滑/软化边缘

为创建聚光灯的平滑边，我们希望去模拟的聚光灯有一个内圆锥和外圆锥。我们可以把内圆锥设置为前面部分定义的圆锥，我们希望外圆锥从内边到外边逐步的变暗。

为创建外圆锥，我们简单定义另一个余弦值，它代表聚光灯的方向向量和外圆锥的向量（等于它的半径）的角度。然后，如果片段在内圆锥和外圆锥之间，就会给它计算出一个 0.0 到 1.0 之间的亮度。如果片段在内圆锥以内这个亮度就等于 1.0，如果在外面就是 0.0。

我们可以使用下面的公式计算这样的

$$F_{att} = \frac{I}{K_c + K_l * d + K_q * d^2}$$

值：这里  $\epsilon$  是内部 ( $\phi$ ) 和外部圆锥 ( $\gamma$ ) 的差。结果  $I$  的值是聚光灯在当前片段的亮度。

很难用图画描述出这个公式是怎样工作的，所以我们尝试使用一个例子：

$\theta$	$\theta$ in degree s	$\phi$ (inner cutoff )	$\phi$ in degree s	$\gamma$ (outer cutoff )	$\gamma$ in degree s	$\epsilon$	$I$
0.87	30	0.91	25	0.82	35	0.91 - 0.82 = 0.09 = 0.56	0.87 - 0.82 / 0.09 = 0.56
0.9	26	0.91	25	0.82	35	0.91 - 0.82 = 0.09 = 0.89	0.9 - 0.82 / 0.09 = 0.89
0.97	14	0.91	25	0.82	35	0.91 -	0.97 -

$\theta$	$\theta$ in degree s	$\varphi$ (inner cutoff )	$\varphi$ in degree s	$\gamma$ (outer cutoff )	$\gamma$ in degree s	$\epsilon$	I
						0.82 = 0.09	0.82 / 0.09 = 1.67
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
0.96 6	15	0.9978	12.5	0.953	17.5	0.966 - 0.953 = 0.044 8	0.966 - 0.953 / 0.044 8 = 0.29

就像你看到的那样我们基本是根据  $\theta$  在外余弦和内余弦之间插值。如果你仍然不明白怎么继续，不要担心。你可以简单的使用这个公式计算，当你更加老道和明白的时候再来看。

由于我们现在有了一个亮度值，当在聚光灯外的时候是个负的，当在内部圆锥以内大于 1。如果我们适当地把这个值固定，我们在片段着色器中就再不需要 if-else 了，我们可以简单地用计算出的亮度值乘以光的元素：

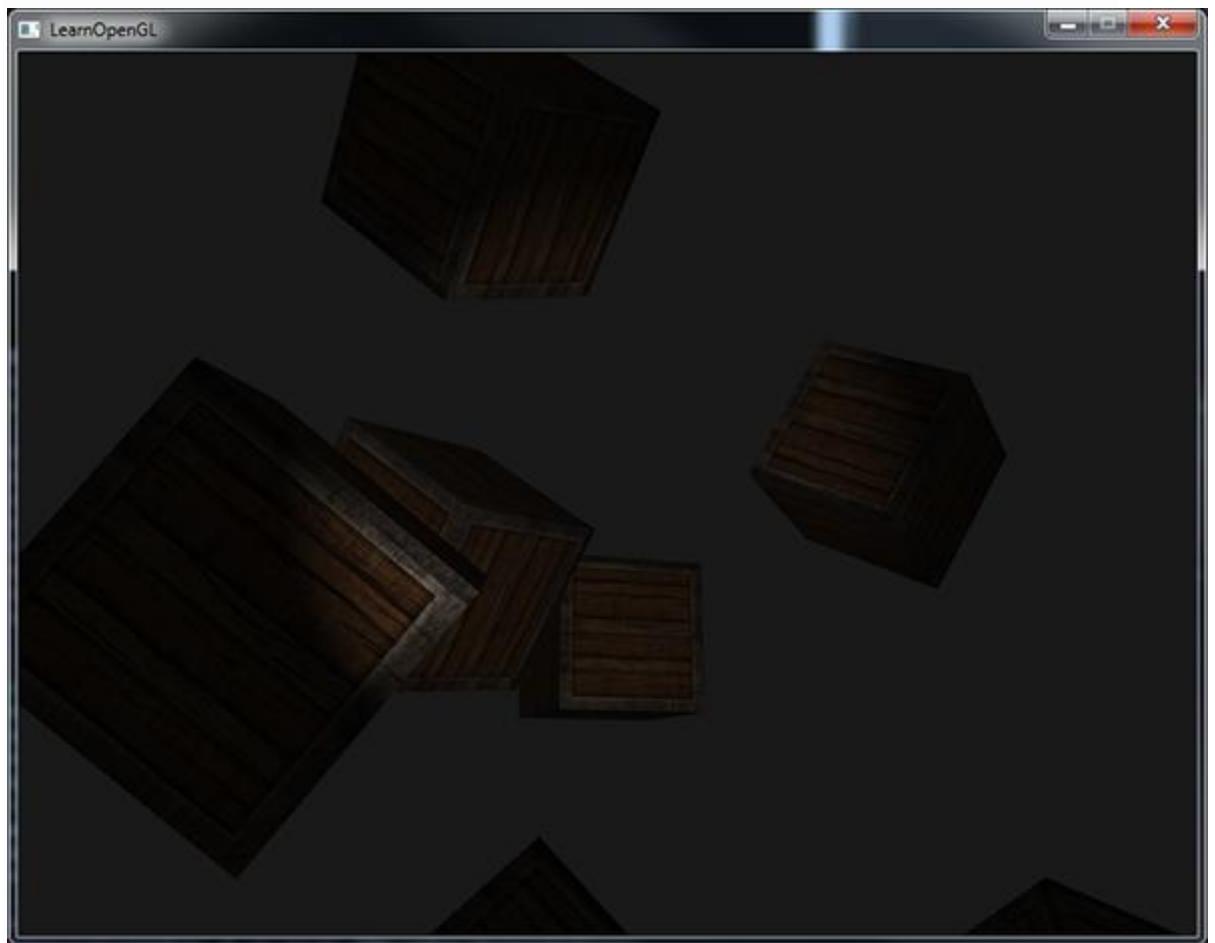
```
float theta = dot(lightDir, normalize(-light.direction));
```

```
float epsilon = light.cutOff - light.outerCutOff;
```

```
float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0,
1.0);
...
// We'll leave ambient unaffected so we always have a little
light.diffuse* = intensity;
specular* = intensity;
...
```

注意，我们使用了 `clamp` 函数，它把第一个参数固定在 0.0 和 1.0 之间。这保证了亮度值不会超出 [0, 1] 以外。

确定你把 `outerCutOff` 值添加到了 `Light` 结构体，并在应用中设置了它的 `uniform` 值。对于下面的图片，内部遮光角 `12.5f`，外部遮光角是 `17.5f`：



看起来好多了。仔细看看内部和外部遮光角，尝试创建一个符合你求的聚光灯。可以在这里找到应用源码，以及片段的源代码。

这样的一个手电筒/聚光灯类型的灯光非常适合恐怖游戏，结合定向和点光，环境会真的开始被照亮了。[下一个教程](#)，我们会结合所有我们目前讨论了的光和技巧。

## 多光源(Multiple lights)

原文	<a href="#">Multiple lights</a>
作者	JoeyDeVries
翻译	<a href="#">Geequlim</a>
校对	<a href="#">Geequlim</a>

我们在前面的教程中已经学习了许多关于 OpenGL 光照的知识，其中包括冯氏照明模型（Phong shading）、光照材质（Materials）、光照图（Lighting maps）以及各种投光物（Light casters）。本教程将结合上述所学的知识，创建一个包含六个光源的场景。我们将模拟一个类似阳光的平行光（Directional light）和 4 个定点光（Point lights）以及一个手电筒(Flashlight).

要在场景中使用多光源我们需要封装一些 GLSL 函数用来计算光照。如果我们对每个光源都去写一遍光照计算的代码，这将是一件令人恶心的事情，并且这些放在 main 函数中的代码将难以理解，所以我们将一些操作封装为函数。

GLSL 中的函数与 C 语言的非常相似，它需要一个函数名、一个返回值类型。并且在调用前必须提前声明。接下来我们将为下面的每一种光照来写一个函数。

当我们在场景中使用多个光源时一般使用以下途径：创建一个代表输出颜色的向量。每一个光源都对输出颜色贡献一些颜色。因此，场景中的每个光源将进行独立运算，并且运算结果都对最终的输出颜色有一定影响。下面是使用这种方式进行多光源运算的一般结构：

```
out vec4 color;
```

```
void main()
```

```

{
 // 定义输出颜色

 vec3 output;

 // 将平行光的运算结果颜色添加到输出颜色

 output += someFunctionToCalculateDirectionalLight();

 // 同样，将定点光的运算结果颜色添加到输出颜色

 for(int i = 0; i < nr_of_point_lights; i++)
 output += someFunctionToCalculatePointLight();

 // 添加其他光源的计算结果颜色（如投射光）

 output += someFunctionToCalculateSpotLight();

 color = vec4(output, 1.0);
}

```

即使对每一种光源的运算实现不同，但此算法的结构一般是与上述出入不大的。我们将定义几个用于计算各个光源的函数，并将这些函数的结算结果（返回颜色）添加到输出颜色向量中。例如，靠近被照射物体的光源计算结果将返回比远离被照射物体的光源更明亮的颜色。

## 平行光（Directional light）

我们要在片段着色器中定义一个函数用来计算平行光在对应的照射点上的光照颜色，这个函数需要几个参数并返回一个计算平行光照结果的颜色。

首先我们需要设置一系列用于表示平行光的变量，正如上一节中所讲过的，我们可以将这些变量定义在一个叫做 **DirLight** 的结构体中，并定义一个这个结构体类型的 **uniform** 变量。

```
struct DirLight {
```

```
 vec3 direction;

 vec3 ambient;

 vec3 diffuse;

 vec3 specular;
};

uniform DirLight dirLight;
```

之后我们可以将 `dirLight` 这个 `uniform` 变量作为下面这个函数原型的参数。

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
```

## Important

和 C/C++一样，我们调用一个函数的前提是这个函数在调用前已经被声明过（此例中我们是在 `main` 函数中调用）。通常情况下我们都将函数定义在 `main` 函数之后，为了能在 `main` 函数中调用这些函数，我们就必须在 `main` 函数之前声明这些函数的原型，这就和我们写 C 语言是一样的。

你已经知道，这个函数需要一个 `DirLight` 和两个其他的向量作为参数来计算光照。如果你看过之前的教程的话，你会觉得下面的函数定义得一点也不意外：

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
 vec3 lightDir = normalize(-light.direction);
 // 计算漫反射强度
 float diff = max(dot(normal, lightDir), 0.0);
 // 计算镜面反射强度
 vec3 reflectDir = reflect(-lightDir, normal);
```

```

float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);

// 合并各个光照分量

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);
}

```

我们从之前的教程中复制了代码，并用两个向量来作为函数参数来计算出平行光的光照射颜色向量，该结果是一个由该平行光的环境反射、漫反射和镜面反射的各个分量组成的一个向量。

## 定点光 (Point light)

和计算平行光一样，我们同样需要定义一个函数用于计算定点光照。同样的，我们定义一个包含定点光源所需属性的结构体：

```

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];
```

如你所见，我们在 **GLSL** 中使用预处理器指令来定义定点光源的数目。之后我们使用这个 **NR\_POINT\_LIGHTS** 常量来创建一个 **PointLight** 结构体的数组。和 C 语言一样，**GLSL** 也是用一对中括号来创建数组的。现在我们有了 4 个 **PointLight** 结构体对象了。

## Important

我们同样可以简单粗暴地定义一个大号的结构体（而不是为每一种类型的光源定义一个结构体），它包含所有类型光源所需要属性变量。并且将这个结构体应用与所有的光照计算函数，在各个光照结算时忽略不需要的属性变量。然而，就我个人来说更喜欢分开定义，这样可以省下一些内存，因为定义一个大号的光源结构体在计算过程中会有用不到的变量。

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3
viewDir);
```

这个函数将所有用得到的数据作为它的参数并使用一个 **vec3** 作为它的返回值类表示一个顶点光的结算结果。我们再一次聪明地从之前的教程中复制代码来把它定义成下面的样子：

```
// 计算定点光在确定位置的光照颜色

vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3
viewDir)
{
 vec3 lightDir = normalize(light.position - fragPos);
```

```
// 计算漫反射强度

float diff = max(dot(normal, lightDir), 0.0);

// 计算镜面反射

vec3 reflectDir = reflect(-lightDir, normal);

float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);

// 计算衰减

float distance = length(light.position - fragPos);

float attenuation = 1.0f / (light.constant + light.linear *
distance +
light.quadratic * (distance * distance));

// 将各个分量合并

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);
```

```
}
```

有了这个函数我们就可以在 `main` 函数中调用它来代替写很多个计算点光源的代码了。通过循环调用此函数就能实现同样的效果，当然代码更简洁。

## 把它们放到一起

我们现在定义了用于计算平行光和定点光的函数，现在我们把这些代码放到一起，写入文开始的一般结构中：

```
void main()
{
 // 一些属性

 vec3 norm = normalize(Normal);

 vec3 viewDir = normalize(viewPos - FragPos);

 // 第一步，计算平行光照
 vec3 result = CalcDirLight(dirLight, norm, viewDir);

 // 第二步，计算顶点光照
 for(int i = 0; i < NR_POINT_LIGHTS; i++)
 result += CalcPointLight(pointLights[i], norm, FragPos,
 viewDir);

 // 第三部，计算 Spot Light
 //result += CalcSpotLight(spotLight, norm, FragPos, viewDir);

 color = vec4(result, 1.0);
}
```

每一个光源的运算结果都添加到了输出颜色上，输出颜色包含了此场景中的所有光源的影响。如果你想实现手电筒的光照效果，同样的把计算结果添加到输出颜色上。我在这里就把 `CalcSpotLight` 的实现留作个读者们的练习吧。

设置平行光结构体的 `uniform` 值和之前所讲过的方式没什么两样，但是你可能想知道如何设置场景中 `PointLight` 结构体的 `uniforms` 变量数组。我们之前并未讨论过如何做这件事。

庆幸的是，这并不是什么难题。设置 `uniform` 变量数组和设置单个 `uniform` 变量值是相似的，只需要用一个合适的下标就能够检索到数组中我们想要的 `uniform` 变量了。

```
glUniform1f(glGetUniformLocation(lightingShader.Program,
"pointLights[0].constant"), 1.0f);
```

这样我们检索到 `pointLights` 数组中的第一个 `PointLight` 结构体元素，同时也可以获取到该结构体中的各个属性变量。不幸的是这一位置我们还需要手动对这个四个光源的每一个属性都进行设置，这样手动设置这 28 个 `uniform` 变量是相当乏味的工作。你可以尝试去定义个光源类来为你设置这些 `uniform` 属性来减少你的工作，但这依旧不能改变去设置每个 `uniform` 属性的事实。

别忘了，我们还需要为每个光源设置它们的位置。这里，我们定义一个 `glm::vec3` 类的数组来包含这些点光源的坐标：

```
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)
};
```

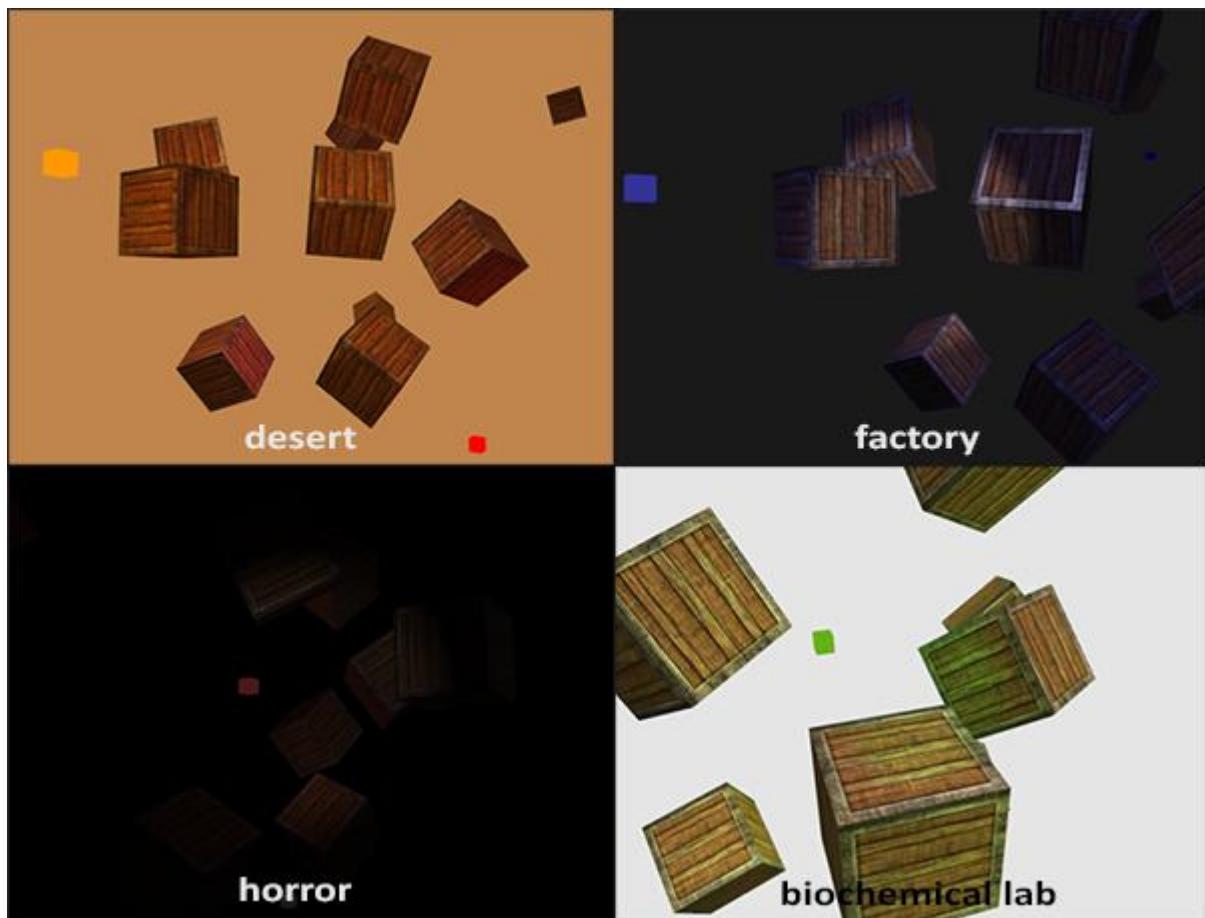
同时我们还需要根据这些光源的位置在场景中绘制 4 个表示光源的立方体，这样的工作我们在之前的教程中已经做过了。

如果你在还是用了手电筒的话，将所有的光源结合起来看上去应该和下图差不多：



你可以在此处获取本教程的[源代码](#)，同时可以查看[顶点着色器](#)和[片段着色器](#)的代码。

上面的图片的光源都是使用默认的属性的效果，如果你尝试对光源属性做出各种修改尝试的话，会出现很多有意思的画面。很多艺术家和场景编辑器都提供大量的按钮或方式来修改光照以使用各种环境。使用最简单的光照属性的改变我们就足已创建有趣的视觉效果：



相信你现在已经对 OpenGL 的光照有很好的理解了。有了这些知识我们便可以创建丰富有趣的环境和氛围了。快试试改变所有的属性的值来创建你的光照环境吧！

## 练习

- 创建一个表示手电筒光的结构体 `Spotlight` 并实现 `CalcSpotLight(...)` 函数：[解决方案](#)
- 你能通过调节不同的光照属性来重新创建一个不同的氛围吗？[解决方案](#)

## 复习

原文	<a href="#">Review</a>
作者	JoeyDeVries
翻译	Meow J

原文

[Review](#)

校对

[Geequlim](#)

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学习的并不是 OpenGL 本身，当然我们仍然学习了一些细枝末节的知识(像访问 uniform 数组)。

到现在的所有教程都是关于用一些技巧和公式来操作着色器从而达到真实的光照效果。这同样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形这一点。

在你学过的最后几个教程中，你学习了有关颜色，冯氏光照模型(包括环境，漫反射，镜面反射光照)，对象材质，可配置的光照属性，漫反射和镜面反射贴图，不同种类的光，并且学习了怎样将所有所学知识融会贯通。

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[下一个教程](#)当中，我们将加入更高级的形状到我们的场景中，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. 一个对象的颜色实际上是该对象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，漫反射，和镜面反射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，从而对全局光照的估计。
- **漫反射着色法(Diffuse Shading):** 光照随着更多的顶点/片段排列在光源上变强. 该方法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。
- **正规矩阵(Normal Matrix):** 一个  $3 \times 3$  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)从而当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失真。

- **镜面光照(Specular Lighting):** (sets a specular highlight the closer the viewer is looking at the reflection of a light source on a surface. 待翻译). 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。
- **冯氏着色法(Phong Shading):** 冯氏光照模型应用在片段着色器。
- **高氏着色法(Gouraud shading):** 冯氏光照模型应用在顶点着色器上. 在使用很少树木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，漫反射，镜面反射光照. 这些东西设定了物体的颜色。
- **光照(性质)(Light(properties)):** 一个光的环境，漫反射，镜面反射的强度. 可以应用任何颜色值并对每一个冯氏分量(Phong Component)都定义一个光源闪烁的颜色/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。
- **镜面贴图(Specular Map):** 一个设定了每一个片段的镜面强度/颜色的纹理贴图. 仅在物体的特定区域允许镜面高光。
- **平行光(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):** 一个场景中光线逐渐淡出的光源。
- **衰减(Attenuation):** 光减少强度的过程，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):** 一个数组的 uniform 值. 就像 C 语言数组一样工作，只是不能被动态调用。

## Assimp 开源模型导入库

原文	<a href="#">Assimp</a>
作者	JoeyDeVries
翻译	Cocoons
校对	<a href="#">Geequlim</a>

到目前为止，我们已经在所有的场景中大面积滥用了我们的容器盒小盆友，但就是容器盒是我们的好朋友，时间久了我们也会喜新厌旧。一些图形应用里经常会使用很多复杂且好玩儿的模型，它们看起来比静态的容器盒可爱多了。但是，我们无法像定义容器盒一样手动地去指定房子、货车或人形角色这些复杂模型的顶点、法线和纹理坐标。我们需要做的也是应该要做的，是把这些模型导入到应用

程序中，而设计制作这些 3D 模型的工作应该交给像 [Blender](#)、[3DS Max](#) 或者 [Maya](#) 这样的工具软件。

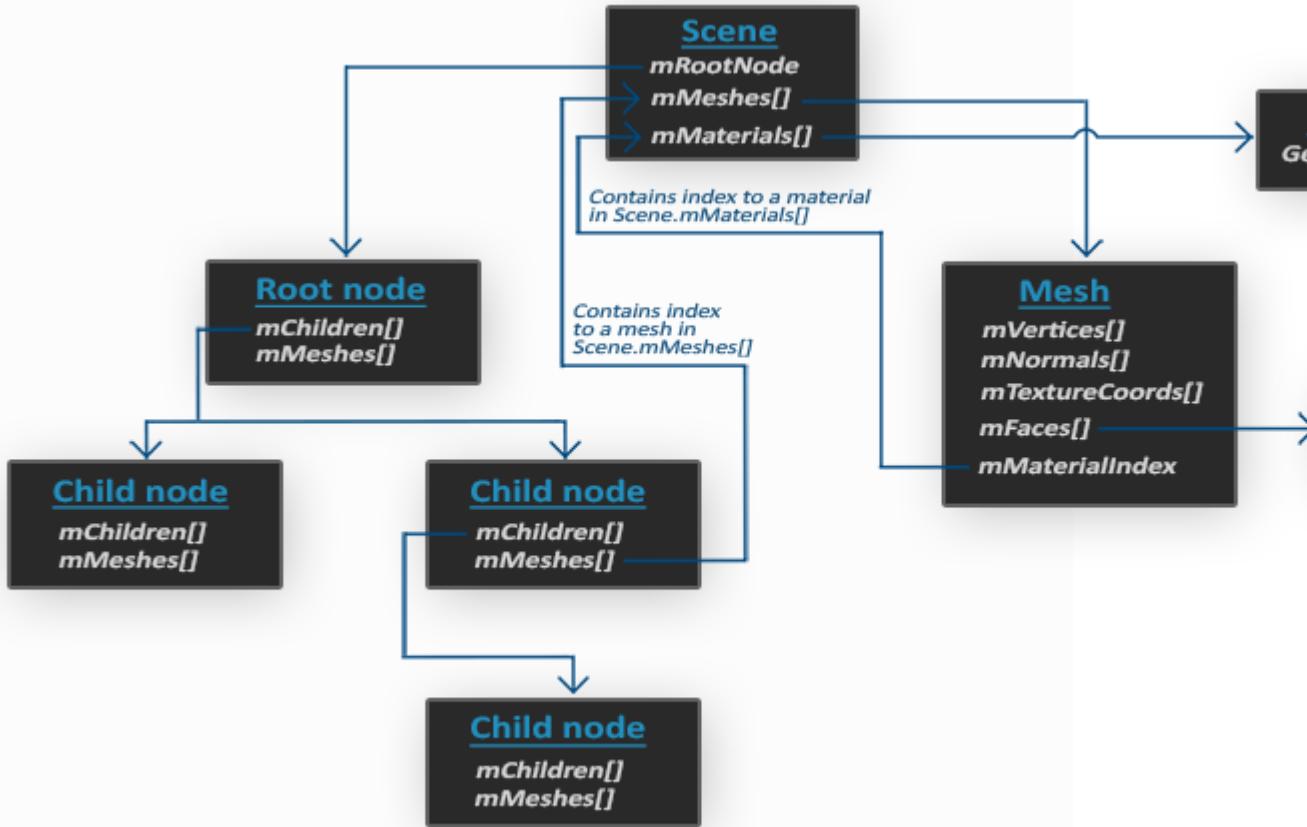
那些 3D 建模工具，可以让美工们构建一些复杂的形状，并将贴图应用到形状上去，即纹理映射。然后，在导出模型文件时，建模工具会自己生成所有的顶点坐标、顶点法线和纹理坐标。这样，美工们可以不用了解大量的图像技术细节，就能有大量的工具集去随心地构建高品质的模型。所有的技术细节内容都隐藏在里导出的模型文件里。而我们，这些图形开发者，就必须得去关注这些技术细节了。

因此，我们的工作就是去解析这些导出的模型文件，并将其中的模型数据存储为 OpenGL 能够使用的数据。一个常见的问题是，导出的模型文件通常有几十种格式，不同的工具会根据不同的文件协议把模型数据导出到不同格式的模型文件中。有的模型文件格式只包含模型的静态形状数据和颜色、漫反射贴图、高光贴图这些基本的材质信息，比如 Wavefront 的.obj 文件。而有的模型文件则采用 XML 来记录数据，且包含了丰富的模型、光照、各种材质、动画、摄像机信息和完整的场景信息等，比如 Collada 文件格式。Wavefront 的 obj 格式是为了考虑到通用性而设计的一种便于解析的模型格式。建议去 Wavefront 的 Wiki 上看看 obj 文件格式是如何封装的。这会给你形成一个对模型文件格式的一个基本概念和印象。

## 模型加载库

现在市面上有一个很流行的模型加载库，叫做 Assimp，全称为 Open Asset Import Library。Assimp 可以导入几十种不同格式的模型文件（同样也可以导出部分模型格式）。只要 Assimp 加载完了模型文件，我们就可以从 Assimp 上获取所有我们需要的模型数据。Assimp 把不同的模型文件都转换为一个统一的数据结构，所有无论我们导入何种格式的模型文件，都可以用同一个方式去访问我们需要的模型数据。

当导入一个模型文件时，即 Assimp 加载一整个包含所有模型和场景数据的模型文件到一个 scene 对象时，Assimp 会为这个模型文件中的所有场景节点、模型节点都生成一个具有对应关系的数据结构，且将这些场景中的各种元素与模型数据对应起来。下图展示了一个简化的 Assimp 生成的模型文件数据结构：



- 所有的模型、场景数据都包含在 `scene` 对象中，如所有的材质和 `Mesh`。同样，场景的根节点引用也包含在这个 `scene` 对象中
- 场景的根节点可能也会包含很多子节点和一个指向保存模型点云数据 `mMeshes[]` 的索引集合。根节点上的 `mMeshes[]` 里保存了实际了 `Mesh` 对象，而每个子节点上的 `mMeshes[]` 都只是指向根节点中的 `mMeshes[]` 的一个引用(译者注：C/C++称为指针，Java/C#称为引用)
- 一个 `Mesh` 对象本身包含渲染所需的所有相关数据，比如顶点位置、法线向量、纹理坐标、面片及物体的材质
- 一个 `Mesh` 会包含多个面片。一个 `Face`（面片）表示渲染中的一个最基本的形状单位，即图元（基本图元有点、线、三角面片、矩形面片）。一个面片记录了一个图元的顶点索引，通过这个索引，可以在 `mMeshes[]` 中寻找到对应的顶点位置数据。顶点数据和索引分开存放，可以便于我们使用缓存（VBO、NBO、TBO、IBO）来高速渲染物体。（详见 [Hello Triangle](#)）
- 一个 `Mesh` 还会包含一个 `Material`（材质）对象用于指定物体的一些材质属性。如颜色、纹理贴图（漫反射贴图、高光贴图等）

所以我们要做的第一件事，就是加载一个模型文件为 `scene` 对象，然后获取每个节点对应的 `Mesh` 对象（我们需要递归搜索每个节点的子节点来获取所有的节

点），并处理每个 **Mesh** 对象对应的顶点数据、索引以及它的材质属性。最终我们得到一个只包含我们需要的数据的 **Mesh** 集合。

## Important

### Mesh(网格,或被译为“模型点云”)

用建模工具构建物体时，美工通常不会直接使用单个形状来构建一个完整的模型。一般来说，一个模型会由几个子模型/形状组合拼接而成。而模型中的那些子模型/形状就是我们所说的一个 **Mesh**。例如一个人形模型，美工通常会把头、四肢、衣服、武器这些组件都分别构建出来，然后在把所有的组件拼合在一起，形成最终的完整模型。一个 **Mesh**(包含顶点、索引和材质属性)是我们在 **OpenGL** 中绘制物体的最小单位。一个模型通常有多个 **Mesh** 组成。

下一节教程中，我们将用上述描述的数据结构来创建我们自己的 **Model** 类和 **Mesh** 类，用于加载和保存那些导入的模型。如果我们想要绘制一个模型，我们不会去渲染整个模型，而是去渲染这个模型所包含的所有独立的 **Mesh**。不管怎样，我们开始导入模型之前，我们需要先把 **Assimp** 导入到我们的工程中。

## 构建 Assimp

你可以在 [Assimp 的下载页面](#)选择一个想要的版本去下载 **Assimp** 库。到目前为止，**Assimp** 可用的最新版本是 3.1.1。我们建议你自己编译 **Assimp** 库，因为 **Assimp** 官方的已编译库不能很好地覆盖在所有平台上运行。如果你忘记怎样使用 **CMake** 编译一个库，请详见 [Creating a window\(创建一个窗口\)](#) 教程。

这里我们列出一些编译 **Assimp** 时可能遇到的问题，以便大家参考和排除：

- **CMake** 在读取配置列表时，报出与 **DirectX** 库丢失相关的一些错误。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：如果你之前没有安装过 DirectX SDK，那么请安装。下载地址：[DirectX SDK](#) - 安装 DirectX SDK 时，可以遇到一个错误码为 **S1023** 的错误。遇到这个问题，请在安装 DirectX SDK 前，先安装 **C++ Redistributable package(s)**。问题解释：[已知问题：DirectX SDK \(June 2010\) 安装及 S1023 错误](#) - 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（编译为 Debug 版本还是 Release 版本，根据你的需要和心情来定吧） - 使用默认配置构建的 Assimp 是一个动态库，所以我们需要把编译出来的 `assimp.dll` 文件拷贝到我们自己程序的可执行文件的同一目录里 - 编译出来的 Assimp 的 LIB 文件和 DLL 文件可以在 code/Debug 或者 code/Release 里找到 - 把编译好的 LIB 文件和 DLL 文件拷贝到工程的相应目录下，并链接到你的解决方案中。同时还要记得把 Assimp 的头文件也拷贝到工程里去（Assimp 的头文件可以在 include 目录里找到）

如果你还遇到了其他问题，可以在下面给出的链接里获取帮助。

## Important

如果你想要让 Assimp 使用多线程支持来提高性能，你可以使用 **Boost** 库来编译 Assimp。在 [Boost 安装页面](#)，你能找到关于 Boost 的完整安装介绍。

现在，你应该已经能够编译 Assimp 库，并链接 Assimp 到你的工程里去了。下一节内容：[导入完美的 3D 物件！](#)

## 网格(Mesh)

原文	<a href="#">Mesh</a>
作者	<a href="#">JoeyDeVries</a>
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

使用 Assimp 可以把多种不同格式的模型加载到程序中，但是一旦载入，它们就都被储存为 Assimp 自己的数据结构。我们最终的目的是把这些数据转变为 OpenGL 可读的数据，才能用 OpenGL 来渲染物体。我们从前面的教程了解到，一个网格(Mesh)代表一个可绘制实体，现在我们就定义一个自己的网格类。

先来复习一点目前学到知识，考虑一个网格最少需要哪些数据。一个网格应该至少需要一组顶点，每个顶点包含一个位置向量，一个法线向量，一个纹理坐标向

量。一个网格也应该包含一个索引绘制用的索引，以纹理(`diffuse/specular map`)形式表现的材质数据。

为了在 OpenGL 中定义一个顶点，现在我们设置有最少需求一个网格类：

```
struct Vertex
{
 glm::vec3 Position;
 glm::vec3 Normal;
 glm::vec2 TexCoords;
};
```

我们把每个需要的向量储存到一个叫做 `Vertex` 的结构体中，它被用来索引每个顶点属性。另外除了 `Vertex` 结构体外，我们也希望组织纹理数据，所以我们定义一个 `Texture` 结构体：

```
struct Texture
{
 GLuint id;
 String type;
};
```

我们储存纹理的 `id` 和它的类型，比如 `diffuse` 纹理或者 `specular` 纹理。

知道了顶点和纹理的实际表达，我们可以开始定义网格类的结构：

```
class Mesh
{
 Public:
```

```

vector<Vertex> vertices;

vector<GLuint> indices;

vector<Texture> textures;

Mesh(vector<Vertex> vertices, vector<GLuint> indices,
 vector<Texture> texture);

Void Draw(Shader shader);

private:

GLuint VAO, VBO, EBO;

void setupMesh();

}

```

如你所见这个类一点都不复杂，构造方法里我们初始化网格所有必须数据。在 `setupMesh` 函数里初始化缓冲。最后通过 `Draw` 函数绘制网格。注意，我们把 `shader` 传递给 `Draw` 函数。通过把 `shader` 传递给 `Mesh`，在绘制之前我们设置几个 `uniform`（就像链接采样器到纹理单元）。

构造函数的内容非常直接。我们简单设置类的公有变量，使用的是构造函数相应的参数。我们在构造函数中也调用 `setupMesh` 函数：

```

Mesh(vector<Vertex> vertices, vector<GLuint> indices,
 vector<Texture> textures)

{

 this->vertices = vertices;

 this->indices = indices;

 this->textures = textures;
}

```

```
this->setupMesh();
```

```
}
```

这里没什么特别的，现在让我们研究一下 `setupMesh` 函数。

## 初始化

现在我们有一大列的网格数据可用于渲染，这要感谢构造函数。我们确实需要设置合适的缓冲，通过顶点属性指针（vertex attribute pointers）定义顶点着色器 layout。现在你应该对这些概念很熟悉，但是我们通过介绍了结构体中使用顶点数据，所以稍微有点不一样：

```
void setupMesh()
```

```
{
```

```
 glGenVertexArrays(1, &this->VAO);
```

```
 glGenBuffers(1, &this->VBO);
```

```
 glGenBuffers(1, &this->EBO);
```

```
 glBindVertexArray(this->VAO);
```

```
 glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
```

```
 glBufferData(GL_ARRAY_BUFFER, this->vertices.size() *
```

```
 sizeof(Vertex),
```

```
 &this->vertices[0], GL_STATIC_DRAW);
```

```
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
```

```
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() *
```

```
 sizeof(GLuint),
```

```
&this->indices[0], GL_STATIC_DRAW);

// 设置顶点坐标指针
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)0);

// 设置法线指针
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, Normal));

// 设置顶点的纹理坐标
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, TexCoords));

 glBindVertexArray(0);
}
}
```

如你所想代码没什么特别不同的地方，在 `Vertex` 结构体的帮助下有了一些小把戏。

C++的结构体有一个重要的属性，那就是在内存中它们是连续的。如果我们用结构体表示一列数据，这个结构体只包含结构体的连续的变量，它就会直接转变为一个 `float`（实际上是 `byte`）数组，我们就能用于一个数组缓冲（array buffer）中了。比如，如果我们填充一个 `Vertex` 结构体，它在内存中的排布等于：

```
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];
```

感谢这个有用的特性，我们能直接把一个作为缓冲数据的一大列 `Vertex` 结构体的指针传递过去，它们会翻译成 `glBufferData` 能用的参数：

```
glBufferData(GL_ARRAY_BUFFER, this->vertices.size() *
 sizeof(Vertex),
 &this->vertices[0], GL_STATIC_DRAW);
```

自然地，`sizeof` 函数也可以使用于结构体来计算字节类型的大小。它应该是 32 字节 ( $8\text{float} * 4$ )。

一个预处理指令叫做 `offsetof(s,m)` 把结构体作为它的第一个参数，第二个参数是这个结构体名字的变量。这是结构体另外的一个重要用途。函数返回这个变量从结构体开始的字节偏移量（`offset`）。这对于定义 `glVertexAttribPointer` 函数偏移量参数效果很好：

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
 (GLvoid*)offsetof(Vertex, Normal));
```

偏移量现在使用 `offsetof` 函数定义了，在这个例子里，设置法线向量的字节偏移量等于法线向量在结构体的字节偏移量，它是 `3float`，也就是 12 字节（一个 `float` 占 4 字节）。注意，我们同样设置步长参数等于 `Vertex` 结构体的大小。

使用一个像这样的结构体，不仅能提供可读性更高的代码同时也是我们可以轻松的扩展结构体。如果我们想要增加另一个顶点属性，我们把它可以简单的添加到结构体中，由于它的可扩展性，渲染代码不会被破坏。

## 渲染

我们需要为 `Mesh` 类定义的最后一个函数，是它的 `Draw` 函数。在真正渲染前我们希望绑定合适的纹理，然后调用 `glDrawElements`。可因为我们从一开始不知道这个网格有多少纹理以及它们应该是什么类型的，所以这件事变得很困难。所以我们该怎样在着色器中设置纹理单元和采样器呢？

解决这个问题，我们需要假设一个特定的名称惯例：每个 `diffuse` 纹理被命名为 `texture_diffuseN`，每个 `specular` 纹理应该被命名为 `texture_specularN`。`N` 是一个从 1 到纹理才抢其允许使用的最大值之间的数。可以说，在一个网格中我们有 3 个 `diffuse` 纹理和 2 个 `specular` 纹理，它们的纹理采样器应该这样被调用：

```
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_diffuse2;
uniform sampler2D texture_diffuse3;
uniform sampler2D texture_specular1;
uniform sampler2D texture_specular2;
```

使用这样的惯例，我们能定义我们在着色器中需要的纹理采样器的数量。如果一个网格真的有（这么多）纹理，我们就知道它们的名字应该是什么。这个惯例也使我们能够处理一个网格上的任何数量的纹理，通过定义合适的采样器开发者可以自由使用希望使用的数量（虽然定义少的话就会有点浪费绑定和 `uniform` 调用了）。

像这样的问题有很多不同的解决方案，如果你不喜欢这个方案，你可以自己创造一个你自己的方案。 最后的绘制代码：

```
void Draw(Shader shader)
{
 GLuint diffuseNr = 1;
 GLuint specularNr = 1;
 for(GLuint i = 0; i < this->textures.size(); i++)
 {
```

```
glActiveTexture(GL_TEXTURE0 + i); // 在绑定纹理前需要激活适当的纹理单元

// 检索纹理序列号 (N in diffuse_textureN)

stringstream ss;

string number;

string name = this->textures[i].type;

if(name == "texture_diffuse")

 ss << diffuseNr++; // 将GLuin 输入到string stream

else if(name == "texture_specular")

 ss << specularNr++; // 将GLuin 输入到string stream

number = ss.str();

glUniform1f(glGetUniformLocation(shader.Program,
("material." + name + number).c_str()), i);

glBindTexture(GL_TEXTURE_2D, this->textures[i].id);

}

glActiveTexture(GL_TEXTURE0);

// 绘制Mesh

glBindVertexArray(this->VAO);

glDrawElements(GL_TRIANGLES, this->indices.size(),

GL_UNSIGNED_INT, 0);

glBindVertexArray(0);
```

}

这不是最漂亮的代码，但是这主要归咎于 C++ 转换类型时的丑陋，比如 `int` 转 `string` 时。我们首先计算 `N`-元素每个纹理类型，把它链接到纹理类型字符串来获取合适的 `uniform` 名。然后查找合适的采样器位置，给它位置值对应回当前激活纹理单元，绑定纹理。这也是我们需要在 `Draw` 方法是用 `shader` 的原因。我们添加 `material` 到作为结果的 `uniform` 名，因为我们通常把纹理储存进材质结构体（对于每个实现也许会有不同）。

## Important

注意，当我们把 `diffuse` 和 `specular` 传递到字符串流（`stringstream`）的时候，计数器会增加，在 C++ 自增叫做：变量`++`，它会先返回自身然后加 1，而`++`变量，先加 1 再返回自身，我们的例子里，我们先传递原来的计数器值到字符串流，然后再加 1，下一轮生效。

你可以从这里得到 [Mesh 类的源码](#)。

`Mesh` 类是对我们前面的教程里讨论的很多话题的抽象在下面的教程里，我们会创建一个模型，它用作存放多个网格物体的容器，真正的实现 Assimp 的加载接口。

## 模型(Model)

原文	<a href="#">Model</a>
作者	<a href="#">JoeyDeVries</a>
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

现在是时候着手启用 Assimp，并开始创建实际的加载和转换代码了。本教程的目标是创建另一个类，这个类可以表达模型的全部。更确切的说，一个模型包含多个网格(`Mesh`)，一个网格可能带有多个对象。一个别墅，包含一个木制阳台，一个尖顶或许也有一个游泳池，它仍然被加载为一个单一模型。我们通过 Assimp 加载模型，把它们转变为多个网格 (`Mesh`) 对象，这些对象是先前教程里创建的。

闲话少说，我把 `Model` 类的结构呈现给你：

```
class Model
{
public:
 /* 成员函数 */
 Model(GLchar* path)
 {
 this->loadModel(path);
 }
 void Draw(Shader shader);

private:
 /* 模型数据 */
 vector<Mesh> meshes;
 string directory;

 /* 私有成员函数 */
 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` 类包含一个 `Mesh` 对象的向量，我们需要在构造函数中给出文件的位置。之后，在构造其中，它通过 `loadModel` 函数加载文件。私有方法都被设计为处理一

部分的 Assimp 导入的常规动作，我们会简单讲讲它们。同样，我们储存文件路径的目录，这样稍后加载纹理的时候会用到。

函数 `Draw` 没有什么特别之处，基本上是循环每个网格，调用各自的 `Draw` 函数。

```
void Draw(Shader shader)
{
 for(GLuint i = 0; i < this->meshes.size(); i++)
 this->meshes[i].Draw(shader);
}
```

## 把一个 3D 模型导入到 OpenGL

为了导入一个模型，并把它转换为我们自己的数据结构，第一件需要做的事是包含合适的 Assimp 头文件，这样编译器就不会对我们抱怨了。

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

我们将要调用的第一个函数是 `loadModel`，它被构造函数直接调用。在 `loadModel` 函数里面，我们使用 Assimp 加载模型到 Assimp 中被称为 `scene` 对象的数据结构。你可能还记得模型加载系列的第一个教程中，这是 Assimp 的数据结构的根对象。一旦我们有了场景对象，我们就能从已加载模型中获取所有所需数据了。

Assimp 最大优点是，它简约的抽象了所加载所有不同格式文件的技术细节，用一行可以做到这一切：

```
Assimp::Importer importer;
const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate
| aiProcess_FlipUVs);
```

我们先来声明一个 `Importer` 对象，它的名字空间是 `Assimp`，然后调用它的 `ReadFile` 函数。这个函数需要一个文件路径，第二个参数是后处理（post-processing）选项。除了可以简单加载文件外，`Assimp` 允许我们定义几个选项来强制 `Assimp` 去对导入数据做一些额外的计算或操作。通过设置 `aiProcess_Triangulate`，我们告诉 `Assimp` 如果模型不是（全部）由三角形组成，应该转换所有的模型的原始几何形状为三角形。`aiProcess_FlipUVs` 基于 y 轴翻转纹理坐标，在处理的时候是必须的（你可能记得，我们在纹理教程中，我们说过在 `OpenGL` 大多数图像会被沿着 y 轴反转，所以这个小小的后处理选项会为我们修正这个）。一些部分其他有用的选项如下：

- `aiProcess_GenNormals`：如果模型没有包含法线向量，就为每个顶点创建法线。
- `aiProcess_SplitLargeMeshes`：把大的网格成几个小的的下级网格，当你渲染有一个最大数量顶点的限制时或者只能处理小块网格时很有用。
- `aiProcess_OptimizeMeshes`：和上个选项相反，它把几个网格结合为一个更大的网格。以减少绘制函数调用的次数的方式来优化。

`Assimp` 提供了后处理说明，你可以从这里找到所有内容。事实上通过 `Assimp` 加载一个模型超级简单。困难的是使用返回的场景对象把加载的数据变换到一个 `Mesh` 对象的数组。

完整的 `loadModel` 函数在这里列出：

```
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: " << import.GetErrorString() << endl;
 }
}
```

```

 cout << "ERROR::ASSIMP::" << import.GetErrorString() << endl;

 return;
}

this->directory = path.substr(0, path.find_last_of('/'));

this->processNode(scene->mRootNode, scene);

}

```

在我们加载了模型之后，我们检验是否场景和场景的根节点为空，查看这些标记中的一个来看看返回的数据是否完整。如果发生了任何一个错误，我们通过导入器（importer）的 `GetErrorString` 函数返回错误报告。我们同样重新获取文件的目录路径。

如果没什么错误发生，我们希望处理所有的场景节点，所以我们传递第一个节点（根节点）到递归函数 `processNode`。因为每个节点（可能）包含多个子节点，我们希望先处理父节点再处理子节点，以此类推。这符合递归结构，所以我们定义一个递归函数。递归函数就是一个做一些什么处理之后，用不同的参数调用它自身的函数，此种循环不会停止，直到一个特定条件发生。在我们的例子里，特定条件是所有的节点都被处理。

也许你记得，Assimp 的结构，每个节点包含一个网格集合的索引，每个索引指向一个在场景对象中特定的网格位置。我们希望获取这些网格索引，获取每个网格，处理每个网格，然后对其他的节点的子节点做同样的处理。`processNode` 函数的内容如下：

```

void processNode(aiNode* node, const aiScene* scene)

{
 // 添加当前节点中的所有 Mesh

 for(GLuint i = 0; i < node->mNumMeshes; i++)
 {
 aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
 }
}

```

```

 this->meshes.push_back(this->processMesh(mesh, scene));
}

// 递归处理该节点的子孙节点

for(GLuint i = 0; i < node->mNumChildren; i++)
{
 this->processNode(node->mChildren[i], scene);
}
}

```

我们首先利用场景的 `mMeshes` 数组来检查每个节点的网格索引以获取相应的网格。被返回的网格被传递给 `processMesh` 函数，它返回一个网格对象，我们可以把它储存在 `meshes` 的 list 或 vector (STL 里的两种实现链表的数据结构) 中。

一旦所有的网格都被处理，我们遍历所有子节点，同样调用 `processNode` 函数。一旦一个节点不再拥有任何子节点，函数就会停止执行。

## Important

认真的读者会注意到，我们可能基本忘记处理任何的节点，简单循环出场景所有的网格，而不是用索引做这件复杂的事。我们这么做的原因是，使用这种节点的原始的想法是，在网格之间定义一个父-子关系。通过递归遍历这些关系，我们可以真正定义特定的网格作为其他网格的父（节点）。

关于这个系统的一个有用的例子是，当你想要平移一个汽车网格需要确保把它的子（节点）比如，引擎网格，方向盘网格和轮胎网格都进行平移；使用父-子关系这样的系统很容易被创建出来。

现在我们没用这种系统，但是无论何时你想要对你的网格数据进行额外的控制，这通常是一种坚持被推荐的做法。这些模型毕竟是那些定义了这些节点风格的关系的艺术家所创建的。

下一步是用上个教程创建的 `Mesh` 类开始真正处理 Assimp 的数据。

## 从 Assimp 到网格

把一个 `aiMesh` 对象转换为一个我们自己定义的网格对象并不难。我们所要做的全部是获取每个网格相关的属性并把这些属性储存到我们自己的对象。通常 `processMesh` 函数的结构会是这样：

```
Mesh processMesh(aiMesh* mesh, const aiScene* scene)

{

 vector<Vertex> vertices;

 vector<GLuint> indices;

 vector<Texture> textures;

 for(GLuint i = 0; i < mesh->mNumVertices; i++)

 {

 Vertex vertex;

 // 处理顶点坐标、法线和纹理坐标

 ...

 vertices.push_back(vertex);

 }

 // 处理顶点索引

 ...

 // 处理材质

 if(mesh->mMaterialIndex >= 0)

 {

 ...

 }
}
```

```
 return Mesh(vertices, indices, textures);
```

```
}
```

处理一个网格基本由三部分组成：获取所有顶点数据，获取网格的索引，获取相关材质数据。处理过的数据被储存在 3 个向量其中之一里面，一个 **Mesh** 被以这些数据创建，返回到函数的调用者。

获取顶点数据很简单：我们定义一个 **Vertex** 结构体，在每次遍历后我们把这个结构体添加到 **Vertices** 数组。我们为存在于网格中的众多顶点循环（通过 **mesh->mNumVertices** 获取）。在遍历的过程中，我们希望用所有相关数据填充这个结构体。每个顶点位置会像这样被处理：

```
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，为了传输 Assimp 的数据，我们定义一个 **vec3** 的宿主，我们需要它是因为 Assimp 维持它自己的数据类型，这些类型用于向量、材质、字符串等。这些数据类型转换到 **glm** 的数据类型时通常效果不佳。

## Important

Assimp 调用他们的顶点位置数组 **mVertices** 真有点违反直觉。

对应法线的步骤毫无疑问是这样的：

```
vector.x = mesh->mNormals[i].x;

vector.y = mesh->mNormals[i].y;

vector.z = mesh->mNormals[i].z;

vertex.Normal = vector;
```

纹理坐标也基本一样，但是 Assimp 允许一个模型的每个顶点有 8 个不同的纹理坐标，我们可能用不到，所以我们只关系第一组纹理坐标。我们也希望检查网格是否真的包含纹理坐标（可能并不总是如此）：

```
if(mesh->mTextureCoords[0]) // Does the mesh contain texture
coordinates?
{
 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** 结构体现在完全被所需的顶点属性填充了，我们能把它添加到 **vertices** 向量的尾部。要对每个网格的顶点做相同的处理。

## 顶点

Assimp 的接口定义每个网格有一个以面 (**faces**) 为单位的数组，每个面代表一个单独的图元，在我们的例子中（由于 **aiProcess\_Triangulate** 选项）总是三角形，一个面包含索引，这些索引定义我们需要绘制的顶点以在那样的顺序提供给每个图元，所以如果我们遍历所有面，把所有面的索引储存到 **indices** 向量，我们需要这么做：

```
for(GLuint i = 0; i < mesh->mNumFaces; i++)
{
 aiFace face = mesh->mFaces[i];
```

```
for(GLuint j = 0; j < face.mNumIndices; j++)
 indices.push_back(face.mIndices[j]);
}
```

所有外部循环结束后，我们现在有了一个完整点的顶点和索引数据来绘制网格，这要调用 `glDrawElements` 函数。可是，为了结束这个讨论，并向网格提供一些细节，我们同样希望处理网格的材质。

## 材质

如同节点，一个网格只有一个指向材质对象的索引，获取网格实际的材质，我们需要索引场景的 `mMaterials` 数组。网格的材质索引被设置在 `mMaterialIndex` 属性中，通过这个属性我们同样能够检验一个网格是否包含一个材质：

```
 textures.insert(textures.end(), specularMaps.begin(),
specularMaps.end());
}
```

我么先从场景的 `mMaterials` 数组获取 `aiMaterial` 对象，然后，我们希望加载网格的 `diffuse` 或/和 `specular` 纹理。一个材质储存了一个数组，这个数组为每个纹理类型提供纹理位置。不同的纹理类型都以 `aiTextureType_` 为前缀。我们使用一个帮助函数：`loadMaterialTextures` 来从材质获取纹理。这个函数返回一个 `Texture` 结构体的向量，我们在之后储存在模型的 `textures` 坐标的后面。

`loadMaterialTextures` 函数遍历所有给定纹理类型的纹理位置，获取纹理的文件位置，然后加载生成纹理，把信息储存到 `Vertex` 结构体。看起来像这样：

```
vector<Texture> loadMaterialTextures(aiMaterial* mat, aiTextureType
type, string typeName)
{
 vector<Texture> textures;

 for(GLuint i = 0; i < mat->GetTextureCount(type); i++)
 {
 aiString str;
 mat->GetTexture(type, i, &str);
 Texture texture;
 texture.id = TextureFromFile(str.C_Str(), this->directory);
 texture.type = typeName;
 texture.path = str;
 textures.push_back(texture);
 }
}
```

```
 return textures;
```

```
}
```

我们先通过 `GetTextureCount` 函数检验材质中储存的纹理，以期得到我们希望得到的纹理类型。然后我们通过 `GetTexture` 函数获取每个纹理的文件位置，这个位置以 `aiString` 类型储存。然后我们使用另一个帮助函数，它被命名为：`TextureFromFile` 加载一个纹理（使用 **SOIL**），返回纹理的 ID。你可以查看列在最后的完整的代码，如果你不知道这个函数应该怎样写出来的话。

## Important

注意，我们假设纹理文件与模型是在相同的目录里。我们可以简单的链接纹理位置字符串和之前获取的目录字符串（在 `loadModel` 函数中得到的）来获得完整的纹理路径（这就是为什么 `GetTexture` 函数同样需要目录字符串）。

有些在互联网上找到的模型使用绝对路径，它们的纹理位置就不会在每天机器上都有效了。例子里，你可能希望手工编辑这个文件来使用本地路径为纹理所使用（如果可能的话）。

这就是使用 **Assimp** 来导入一个模型的全部了。你可以在这里找到 [Model 类的代码](#)。

## 重大优化

我们现在还没做完。因为我们还想做一个重大的优化（但是不是必须的）。大多数场景重用若干纹理，把它们应用到网格；还是思考那个别墅，它有个花岗岩的纹理作为墙面。这个纹理也可能应用到地板、天花板，楼梯，或者一张桌子、一个附近的小物件。加载纹理需要不少操作，当前的实现中一个新的纹理被加载和生成，来为每个网格使用，即使同样的纹理之前已经被加载了好几次。这会很快转变为你的模型加载实现的瓶颈。

所以我们打算添加一个小小的微调，把模型的代码改成，储存所有的已加载纹理到全局。无论在哪儿我们都要先检查这个纹理是否已经被加载过了。如果加载过了，我们就直接使用这个纹理并跳过整个加载流程来节省处理能力。为了对比纹理我们同样需要储存它们的路径：

```
struct Texture {
```

```
 GLuint id;
```

```
 string type;

 aiString path; // We store the path of the texture to compare with
 other textures
};
```

然后我们把所有加载过的纹理储存到另一个向量中，它是作为一个私有变量声明在模型类的顶部：

```
vector<Texture> textures_loaded;
```

然后，在 `loadMaterialTextures` 函数中，我们希望把纹理路径和所有 `texture_loaded` 向量对比，看看是否当前纹理路径和其中任何一个是否相同，如果是，我们跳过纹理加载/生成部分，简单的使用已加载纹理结构体作为网格纹理。这个函数如下所示：

```
vector<Texture> loadMaterialTextures(aiMaterial* mat, aiTextureType
type, string typeName)
{
 vector<Texture> textures;

 for(GLuint i = 0; i < mat->GetTextureCount(type); i++)
 {
 aiString str;

 mat->GetTexture(type, i, &str);

 GLboolean skip = false;

 for(GLuint j = 0; j < textures_loaded.size(); j++)
 {
 if(textures_loaded[j].path == str)
```

```
{
 textures.push_back(textures_loaded[j]);
 skip = true;
 break;
}
}

if(!skip)
{ // 如果纹理没有被加载过, 加载之
 Texture texture;
 texture.id = TextureFromFile(str.C_Str(),
 this->directory);
 texture.type = typeName;
 texture.path = str;
 textures.push_back(texture);
 this->textures_loaded.push_back(texture); // 添加到纹理

 列表 textures
}
}
}

return textures;
}
```

所以现在我们不仅有了一个通用模型加载系统，同时我们也得到了一个能使加载对象更快的优化版本。

## Attention

有些版本的 **Assimp** 当使用调试版或/和使用你的 **IDE** 的调试模式时，模型加载模型实在慢，所以确保在当你加载得很慢的时候用发布版再测试。

你可以从这里获得优化的 [Model 类的完整源代码](#)。

## 和箱子模型告别！

现在给我们导入一个天才艺术家创建的模型看看效果，不是我这个天才做的（你不得不承认，这个箱子也许是见过的最漂亮的立体图形）。因为我不想过于自夸，所以我会时不时的给其他艺术家进入这个行列的机会，这次我们会加载 **Crytek** 原版的孤岛危机游戏中的纳米铠甲。这个模型被输出为 **obj** 和 **mtl** 文件，**mtl** 包含模型的 **diffuse** 和 **specular** 以及法线贴图（后面会讲）。你可以下载这个模型，注意，所有的纹理和模型文件都应该放在同一个目录，以便载入纹理。

### Important

你从这个站点下载的版本是修改过的版本，每个纹理文件路径已经修改改为本地相对目录，原来的资源是绝对目录。

现在在代码中，声明一个 **Model** 对象，把它模型的文件位置传递给它。模型应该自动加载（如果没有错误的话）在游戏循环中使用它的 **Draw** 函数绘制这个对象。没有更多的缓冲配置，属性指针和渲染命令，仅仅简单的一行。如果你创建几个简单的着色器，像素着色器只输出对象的 **diffuse** 纹理颜色，结果看上去会有点像这样：



你可以从这里找到带有[顶点](#)和[片段](#)着色器的[完整的源码](#)。

我们也可以变得更加有创造力，引入两个点光源到我们之前从光照教程学过的渲染等式，结合高光贴图获得惊艳效果：



虽然我不得不承认这个相比之前用过的容器也太炫了。使用 Assimp，你可以载入无数在互联网上找到的模型。只有很少的资源网站提供多种格式的免费 3D 模型给你下载。一定注意，有些模型仍然不能很好的载入，纹理路径无效或者这种格式 Assimp 不能读。

## 练习

你可以使用两个点光源重建上个场景吗？[方案](#)，[着色器](#)。

## 深度测试(Depth testing)

[原文](#)    [Depth testing](#)

原文	<a href="#">Depth testing</a>
作者	<a href="#">JoeyDeVries</a>
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

在[坐标系的教程](#)中我们呈现了一个 3D 容器, 使用深度缓冲, 以防止被其他面遮挡的面渲染到前面。在本教程中我们将细致地讨论被深度缓冲区(depth-buffer 或 z-buffer)所存储的深度值以及它是如何确定一个片段是否被其他片段遮挡。

深度缓冲就像颜色缓冲(存储所有的片段颜色: 视觉输出)那样存储每个片段的信息, (通常) 和颜色缓冲区有相同的宽度和高度。深度缓冲由窗口系统自动创建并将其深度值存储为 16、 24 或 32 位浮点数。在大多数系统中深度缓冲区为 24 位。

当深度测试启用的时候, OpenGL 测试深度缓冲区内的深度值。OpenGL 执行深度测试的时候, 如果此测试通过, 深度缓冲内的值将被设为新的深度值。如果深度测试失败, 则丢弃该片段。

深度测试在片段着色器运行之后(并且模板测试运行之后, 我们将在[接下来](#)的教程中讨论)在屏幕空间中执行的。屏幕空间坐标直接有关的视区, 由 OpenGL 的 `glViewport` 函数给定, 并且可以通过 GLSL 的片段着色器中内置的 `gl_FragCoord` 变量访问。`gl_FragCoord` 的 X 和 y 表示该片段的屏幕空间坐标 ((0, 0) 在左下角)。`gl_FragCoord` 还包含一个 z 坐标, 它包含了片段的实际深度值。此 z 坐标值是与深度缓冲区的内容进行比较的值。

## Important

现在大多数 GPU 都支持一种称为提前深度测试(Early depth testing)的硬件功能。提前深度测试允许深度测试在片段着色器之前运行。明确一个片段永远不会可见的 (它是其它物体的后面) 我们可以更早地放弃该片段。

片段着色器通常是相当费时的所以我们应该尽量避免运行它们。对片段着色器提前深度测试一个限制是, 你不应该写入片段的深度值。如果片段着色器将写入其深度值, 提前深度测试是不可能的, OpenGL 不能事先知道深度值。

深度测试默认是关闭的, 要启用深度测试的话, 我们需要用 `GL_DEPTH_TEST` 选项来打开它:

```
glEnable(GL_DEPTH_TEST);
```

一旦启用深度测试，如果片段通过深度测试，OpenGL 自动在深度缓冲区存储片段的 z 值，如果深度测试失败，那么相应地丢弃该片段。如果启用深度测试，那么在每个渲染之前还应使用 `GL_DEPTH_BUFFER_BIT` 清除深度缓冲区，否则深度缓冲区将保留上一次进行深度测试时所写的深度值

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

在某些情况下我们需要进行深度测试并相应地丢弃片段，但我们不希望更新深度缓冲区，基本上，可以使用一个只读的深度缓冲区；OpenGL 允许我们通过将其深度掩码设置为 `GL_FALSE` 禁用深度缓冲区写入：

```
glDepthMask(GL_FALSE);
```

注意这只在深度测试被启用的时候有效。

## 深度测试函数

OpenGL 允许我们修改它深度测试使用的比较运算符(*comparison operators*)。这样我们能够控制 OpenGL 通过或丢弃碎片和如何更新深度缓冲区。我们可以通过调用 `glDepthFunc` 来设置比较运算符 (或叫做深度函数(*depth function*))：

```
glDepthFunc(GL_LESS);
```

该函数接受在下表中列出的几个比较运算符：

运算符	描述
GL_ALWAYS	永远通过测试
GL_NEVER	永远不通过测试
GL_LESS	在片段深度值小于缓冲区的深度时通过测试
GL_EQUAL	在片段深度值等于缓冲区的深度时通过测试
GL_LEQUAL	在片段深度值小于等于缓冲区的深度时通过测试
GL_GREATER	在片段深度值大于缓冲区的深度时通过测试

运算符	描述
GL_NOTEQUAL	在片段深度值不等于缓冲区的深度时通过测试
GL_GEQUAL	在片段深度值大于等于缓冲区的深度时通过测试

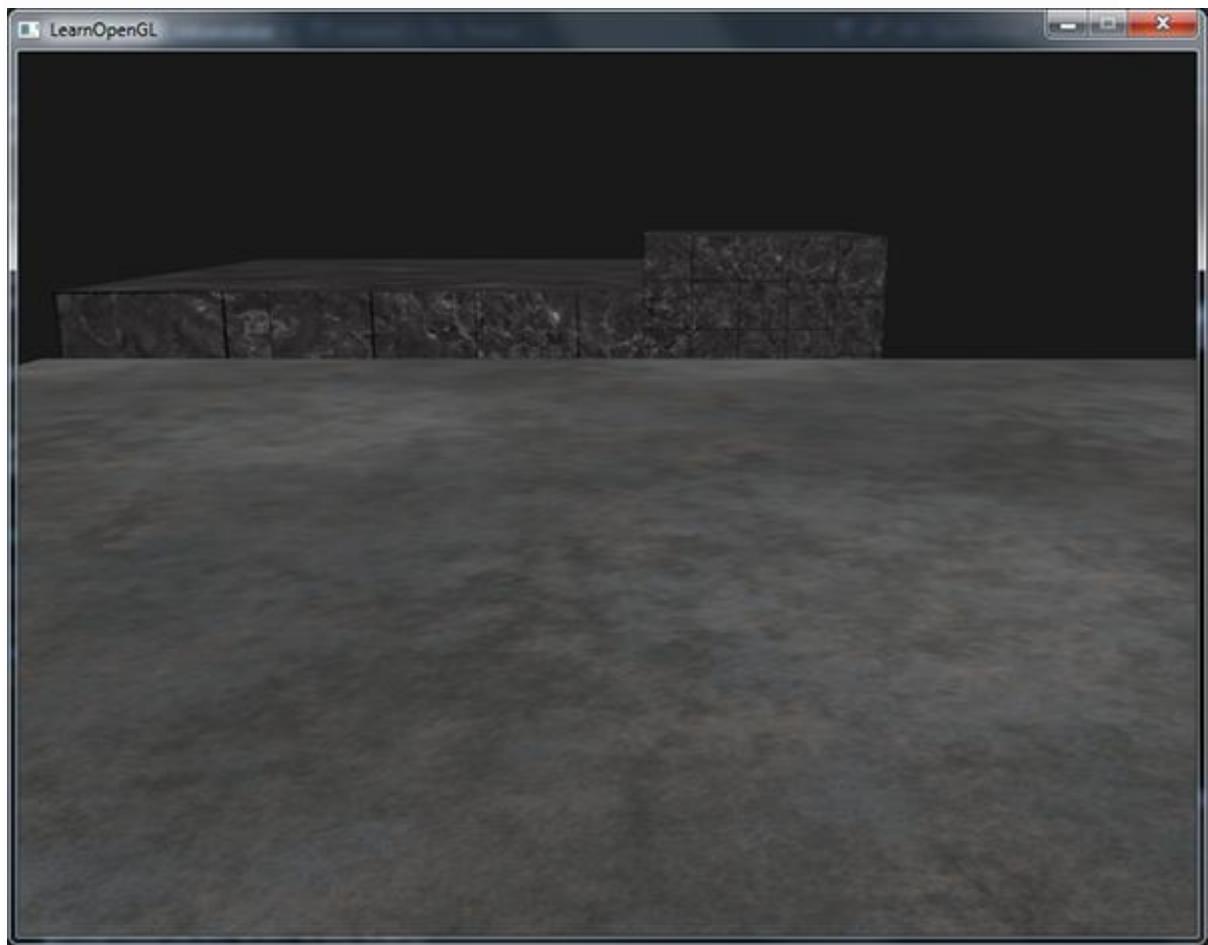
默认情况下使用 `GL_LESS`，这将丢弃深度值高于或等于当前深度缓冲区的值的片段。

让我们看看改变深度函数对输出的影响。我们将使用新鲜的代码安装程序显示一个没有灯光的有纹理地板上的两个有纹理的立方体。你可以在这里找到源代码和其着色器代码。

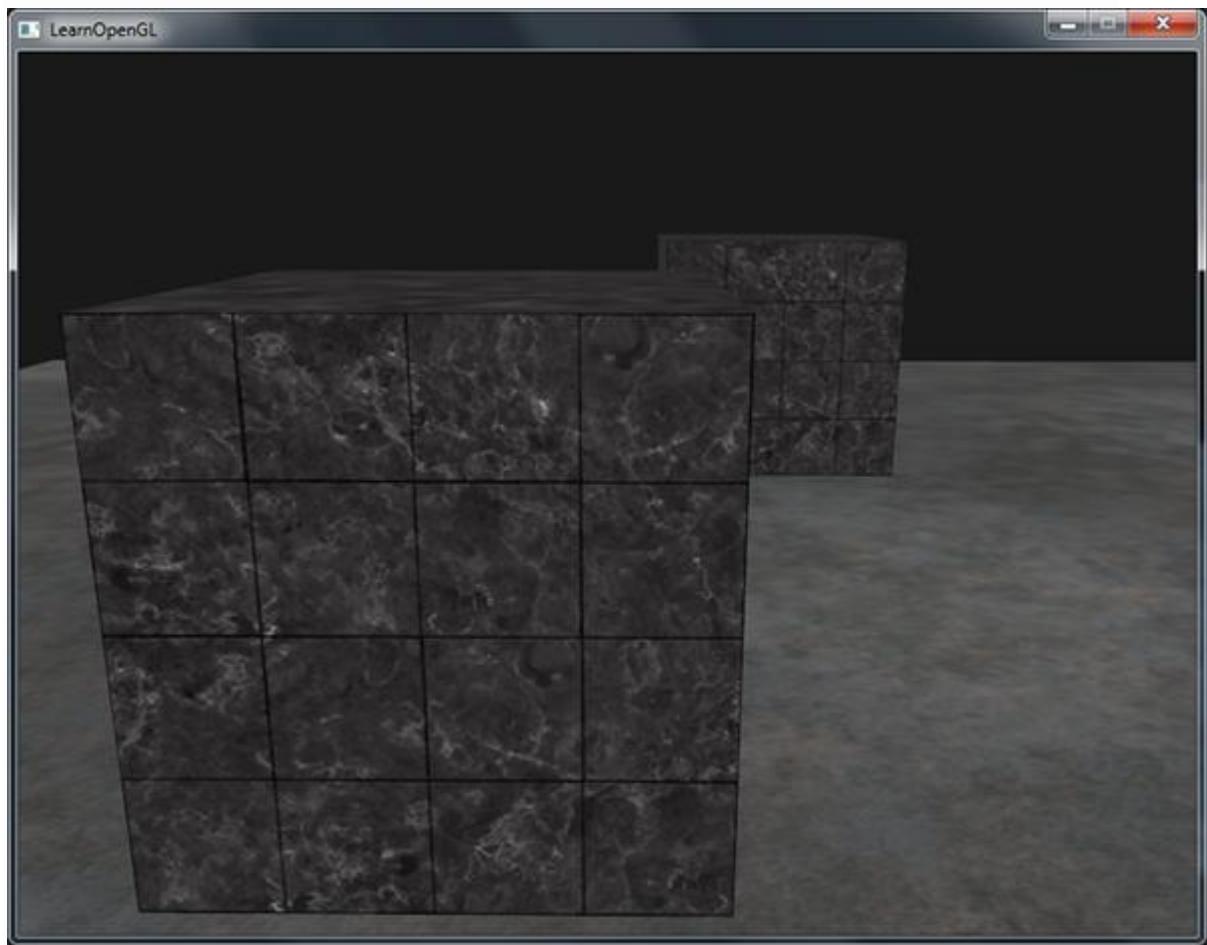
代码中我们将深度函数设为 `GL_ALWAYS`:

```
glDepthFunc(GL_ALWAYS);
```

这和我们没有启用深度测试得到了相同的行为。深度测试只是简单地通过，所以这样最后绘制的片段就会呈现在之前绘制的片段前面，即使他们应该在前面。由于我们最后绘制地板平面，那么平面的片段会覆盖每个容器的片段:



重新设置到 `GL_LESS` 给了我们曾经的场景：

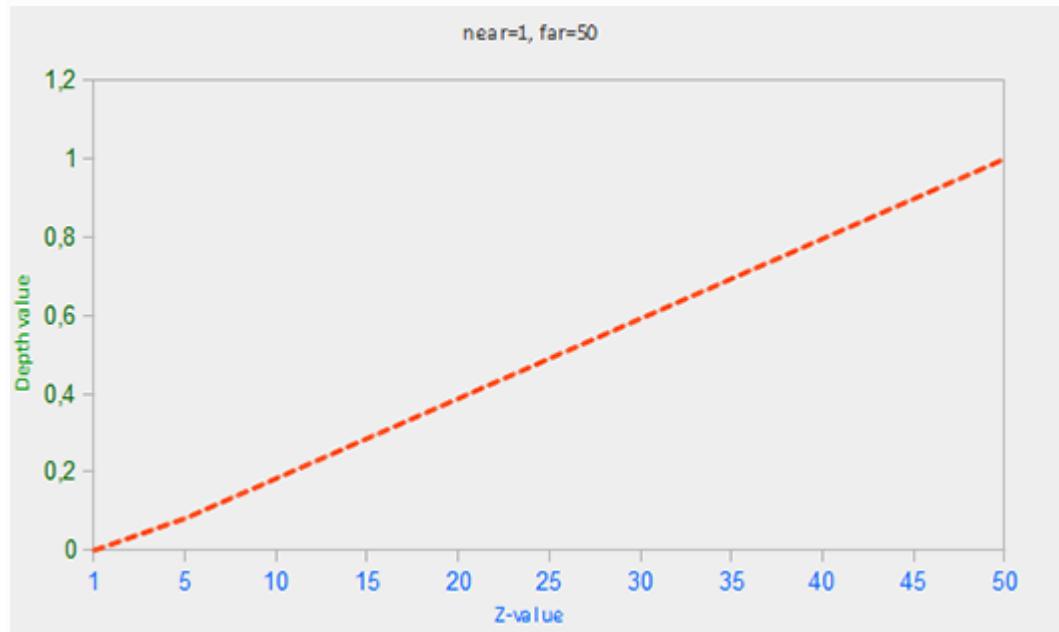


## 深度值精度

在深度缓冲区中包含深度值介于  $0.0$  和  $1.0$  之间，从观察者看到其内容与场景中的所有对象的  $z$  值进行了比较。这些视图空间中的  $z$  值可以在投影平头截体的近平面和远平面之间的任何值。我们因此需要一些方法来转换这些视图空间  $z$  值到  $[0, 1]$  的范围内，方法之一就是线性将它们转换为  $[0, 1]$  范围内。下面的（线性）方程把  $z$  值转换为  $0.0$  和  $1.0$  之间的值：

$$F_{depth} = \frac{z - near}{far - near}$$

这里 `far` 和 `near` 是我们用来提供到投影矩阵设置可见视图截锥的远近值 (见[坐标系](#))。方程带内锥截体的深度值 `z`, 并将其转换到 [0, 1] 范围。在下面的图给出 `z` 值和其相应的深度值的关系:



### Important

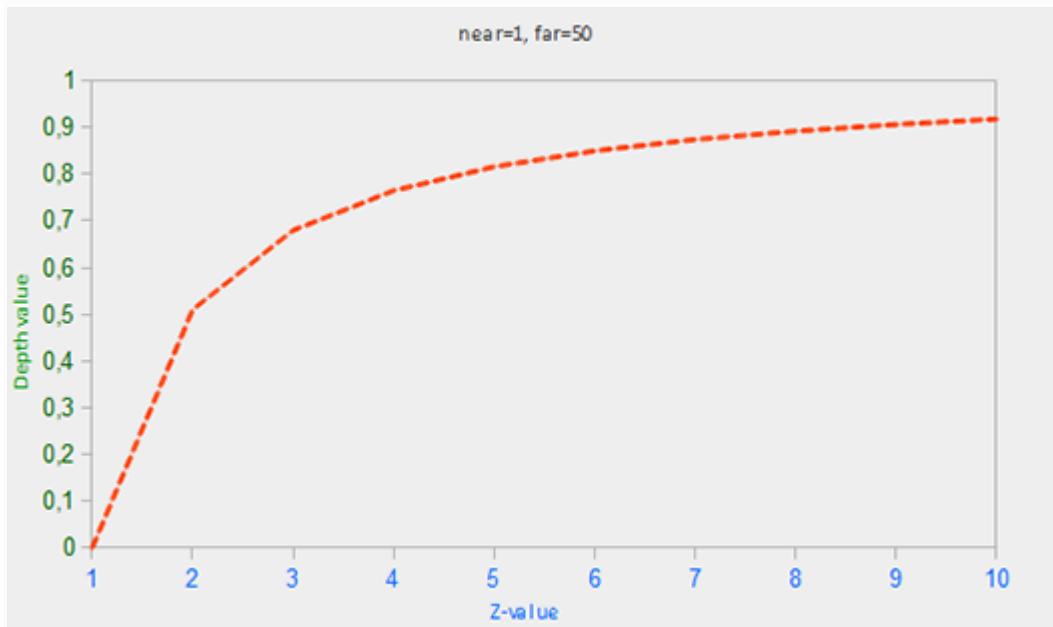
注意在物体接近近平面的时候, 方程给出的深度值接近 0.0, 物体接近远平面时, 方程给出的深度接近 1.0。

然而, 在实践中是几乎从来不使用这样的线性深度缓冲区。正确的投影特性的非线性深度方程是和  $1/z$  成正比的。这样基本上做的是在  $z$  很近是的高精度和  $z$  很远的时候的低精度。用几秒钟想一想: 我们真的需要让 1000 单位远的物体和只有 1 单位远的物体的深度值有相同的精度吗? 线性方程没有考虑这一点。

由于非线性函数是和  $1/z$  成正比，例如 1.0 和 2.0 之间的  $z$  值，将变为 1.0 到 0.5 之间，这样在  $z$  非常小的时候给了我们很高的精度。50.0 和 100.0 之间的  $z$  值将只占 2% 的浮点数的精度，这正是我们想要的。这类方程，也需要近和远距离考虑，下面给出：

$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

如果你不知道这个方程到底怎么回事也不必担心。要记住的重要一点是在深度缓冲区的值不是线性的屏幕空间（它们在视图空间投影矩阵应用之前是线性）。值为 0.5 在深度缓冲区并不意味着该对象的  $z$  值是投影平头截体的中间；顶点的  $z$  值是实际上相当接近近平面！你可以看到  $z$  值和产生深度缓冲区的值在下列图中的非线性关系：



正如你所看到，一个附近的物体的小的  $z$  值因此给了我们很高的深度精度。变换（从观察者的角度）的  $z$  值的方程式被嵌入在投影矩阵，所以当我们变换顶点坐标从视图到裁剪，然后到非线性方程应用了的屏幕空间中。如果你好奇的投影矩阵究竟做了什么我建议阅读[这个文章](#)。

接下来我们看看这个非线性的深度值。

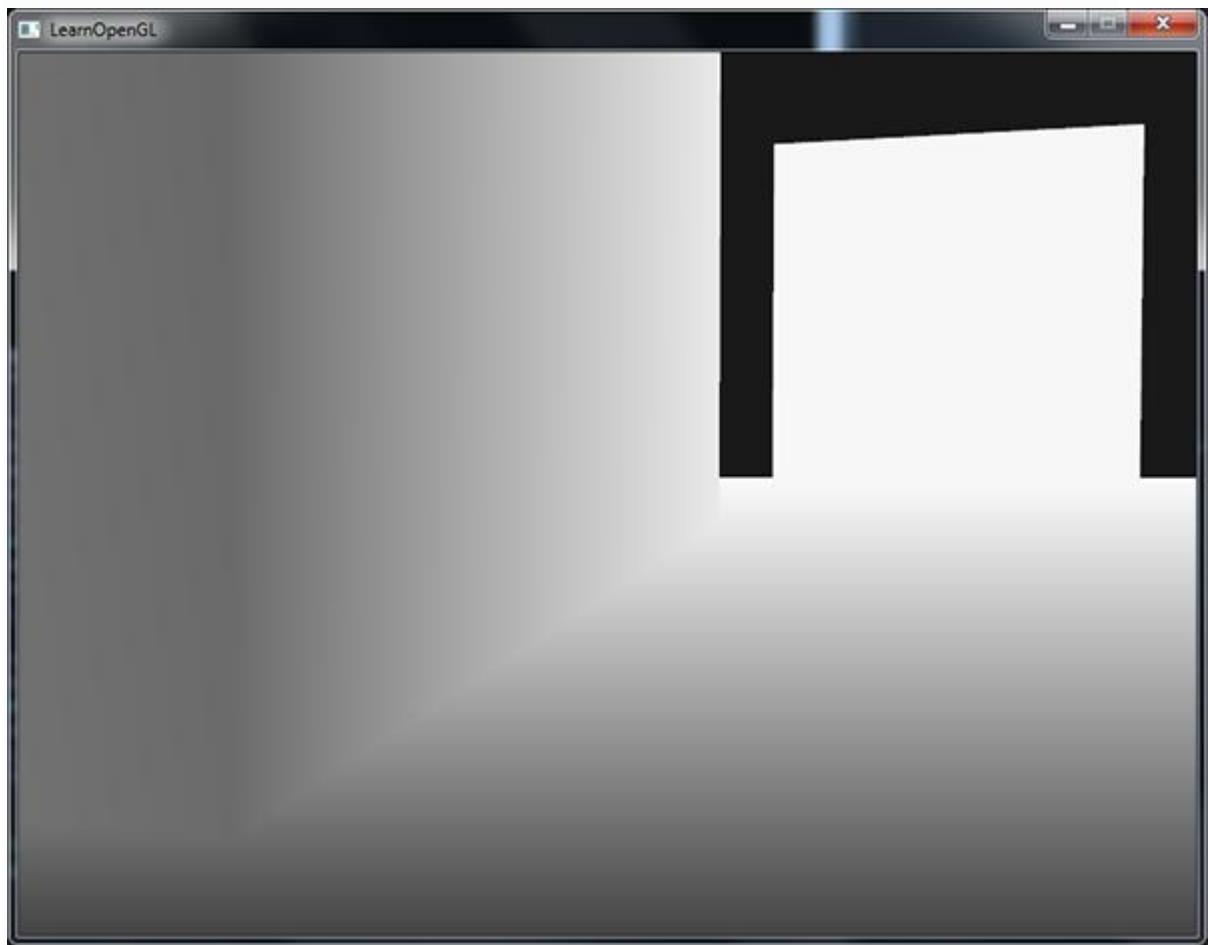
## 深度缓冲区的可视化

我们知道在片段渲染器的内置 `gl_FragCoord` 向量的 `z` 值包含那个片段的深度值。如果我们要吧深度值作为颜色输出，那么我们可以在场景中显示的所有片段的深度值。我们可以返回基于片段的深度值的颜色向量：

```
void main()
{
 color = vec4(vec3(gl_FragCoord.z), 1.0f);
}
```

如果再次运行同一程序你可能会发现一切都是白的，它看起来像我们的深度值都是最大值 `1.0`。那么为什么没有深度值接近 `0.0` 而发暗？

你可能还记得从上一节中的屏幕空间的深度值是非线性如他们在 `z` 很小的时候有很高的精度，， 较大的 `z` 值有较低的精度。该片段的深度值会迅速增加，所以几乎所有顶点的深度值接近 `1.0`。如果我们小心的靠近物体，你最终可能会看到的色彩越来越暗，意味着它们的 `z` 值越来越小：



这清楚地表明深度值的非线性特性。近的物体相对远的物体对的深度值比对象较大的影响。只移动几英寸就能让暗色完全变亮。

但是我们可以让深度值变换回线性。要实现这一目标我们需要让点应用投影变换逆的逆变换，成为单独的深度值的过程。这意味着我们必须首先重新变换范围  $[0, 1]$  中的深度值为单位化的设备坐标(normalized device coordinates)范围内  $[-1, 1]$  (裁剪空间(clip space))。然后，我们想要反转非线性方程 (等式 2) 就像在投影矩阵做的那样并将此反转方程应用于所得到的深度值。然后，结果是一个线性的深度值。听起来能行对吗？

首先，我们需要并不太难的 NDC 深度值转换：

```
float z = depth * 2.0 - 1.0;
```

然后把我们所得到的  $z$  值应用逆转换来检索的线性深度值：

```
float linearDepth = (2.0 * near) / (far + near - z * (far - near));
```

注意此方程不是方程 2 的精确的逆方程。这个方程从投影矩阵中导出，可以从新使用等式 2 将他转换为非线性深度值。这个方程也会考虑使用[0, 1] 而不是[near, far]范围内的 z 值。[math-heavy](#) 为感兴趣的读者阐述了大量详细地投影矩阵的知识;它还表明了方程是从哪里来的。

这不是从投影矩阵推导出的准确公式;这个方程是除以 far 的结果。深度值的范围一直到 far, 这作为一个介于 0.0 和 1.0 之间的颜色值并不合适。除以 far 的值把深度值映射到介于 0.0 和 1.0, 更适合用于演示目的。

这个能够将屏幕空间的非线性深度值转变为线性深度值的完整的片段着色器如下所示:

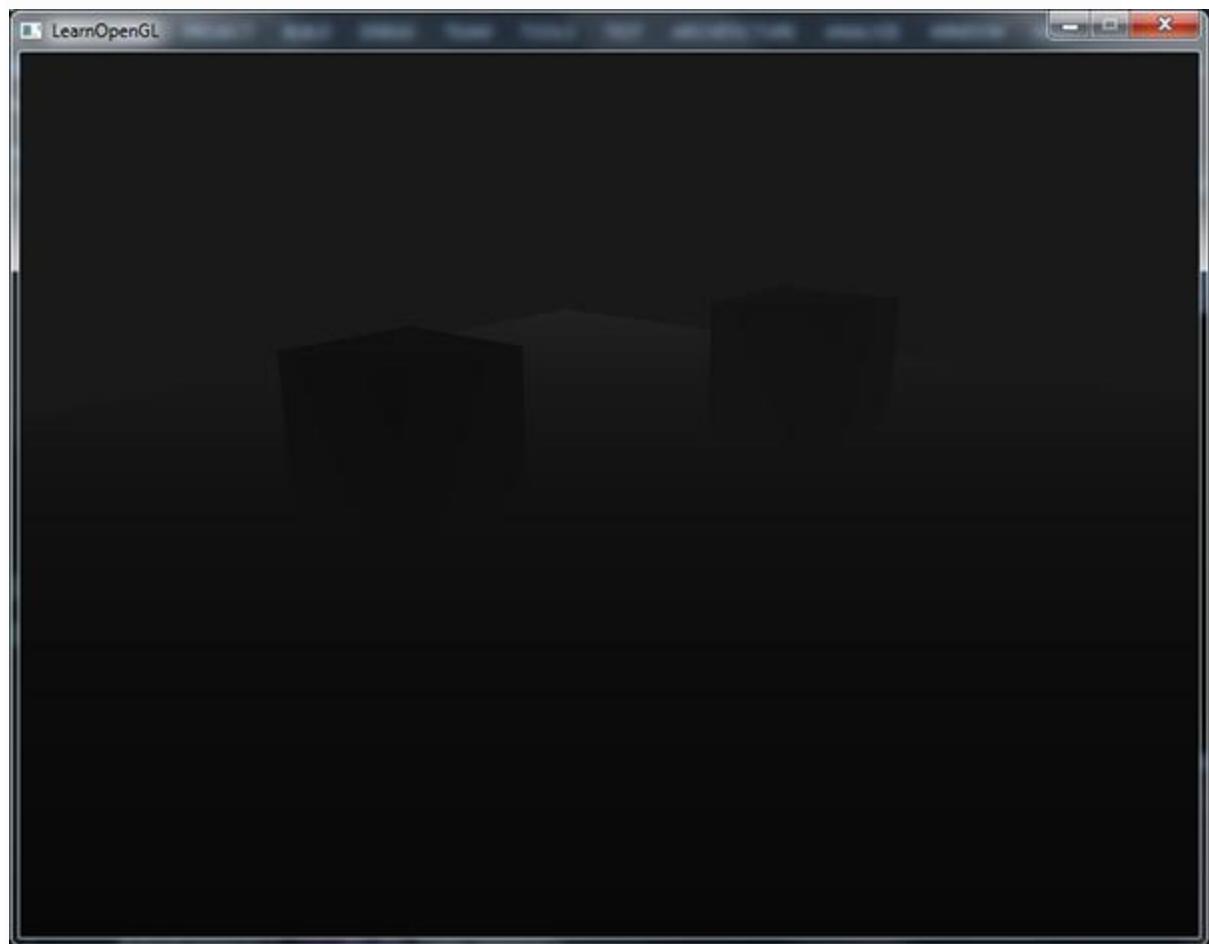
```
#version 330 core

out vec4 color;

float LinearizeDepth(float depth)
{
 float near = 0.1;
 float far = 100.0;
 float z = depth * 2.0 - 1.0; // Back to NDC
 return (2.0 * near) / (far + near - z * (far - near));
}

void main()
{
 float depth = LinearizeDepth(gl_FragCoord.z);
 color = vec4(vec3(depth), 1.0f);
}
```

如果现在运行该应用程序，我们得到在距离实际上线性的深度值。尝试移动现场周围看到深度值线性变化



。

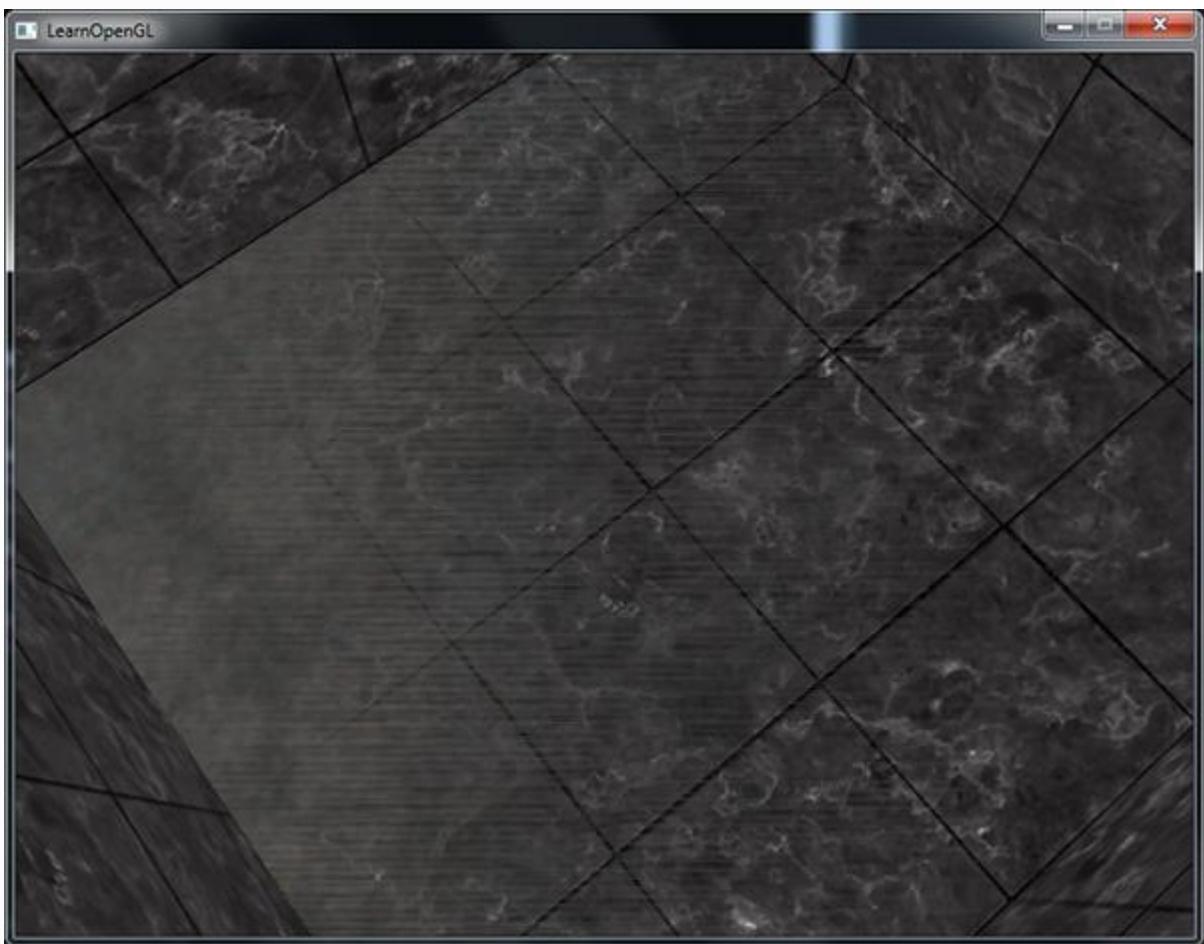
颜色主要是黑色的因为深度值线性范围从 0.1 的近平面到 100 的远平面，那里离我们很远。其结果是，我们相对靠近近平面，从而得到较低（较暗）的深度值。

## 深度冲突

两个平面或三角形如此紧密相互平行深度缓冲区不具有足够的精度以至于无法得到哪一个靠前。结果是，这两个形状不断似乎切换顺序导致怪异出问题。这被称为深度冲突，因为它看上去像形状争夺顶靠前的位置。

我们到目前为止一直在使用的场景中有几个地方深度冲突很显眼。容器被置于确切高度地板被安置这意味着容器的底平面与地板平面共面。两个平面的深度值是相同的，因此深度测试也没有办法找出哪个是正确。

如果您移动摄像机到容器的里面，那么这个影响清晰可，容器的底部不断切换容器的平面和地板的平面：



深度冲突是深度缓冲区的普遍问题，当对象的距离越远一般越强(因为深度缓冲区在 z 值非常大的时候没有很高的精度)。深度冲突还无法完全避免，但有一般的几个技巧，将有助于减轻或完全防止深度冲突在你的场景中的出现：

## 防止深度冲突

第一个也是最重要的技巧是让物体之间不要离得太近，以至于他们的三角形重叠。通过在物体之间制造一点用户无法察觉到的偏移，可以完全解决深度冲突。在容器和平面的条件下，我们可以把容器像 +y 方向上略微移动。这微小的改变可能完全不被注意但是可以有效地减少或者完全解决深度冲突。然而这需要人工的干预每个物体，并进行彻底地测试，以确保这个场景的物体之间没有深度冲突。

另一个技巧是尽可能把近平面设置得远一些。前面我们讨论过越靠近近平面的位置精度越高。所以我们移动近平面远离观察者，我们可以在椎体内很有效的提高

精度。然而把近平面移动的太远会导致近处的物体被裁剪掉。所以不断调整测试近平面的值，为你的场景找出最好的近平面的距离。

另外一个技巧是放弃一些性能来得到更高的深度值的精度。大多数的深度缓冲区都是 **24** 位。但现在显卡支持 **32** 位深度值，这让深度缓冲区的精度提高了一大节。所以牺牲一些性能你会得到更精确的深度测试，减少深度冲突。

我们已经讨论过的 **3** 个技术是最常见和容易实现消除深度冲突的技术。还有一些其他技术需要更多的工作，仍然不会完全消除深度冲突。深度冲突是一个常见的问题，但如果你将列举的技术适当结合你可能不会真的需要处理深度冲突。

## 模板测试(**Stencil testing**)

原文	<a href="#">Stencil testing</a>
作者	JoeyDeVries
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

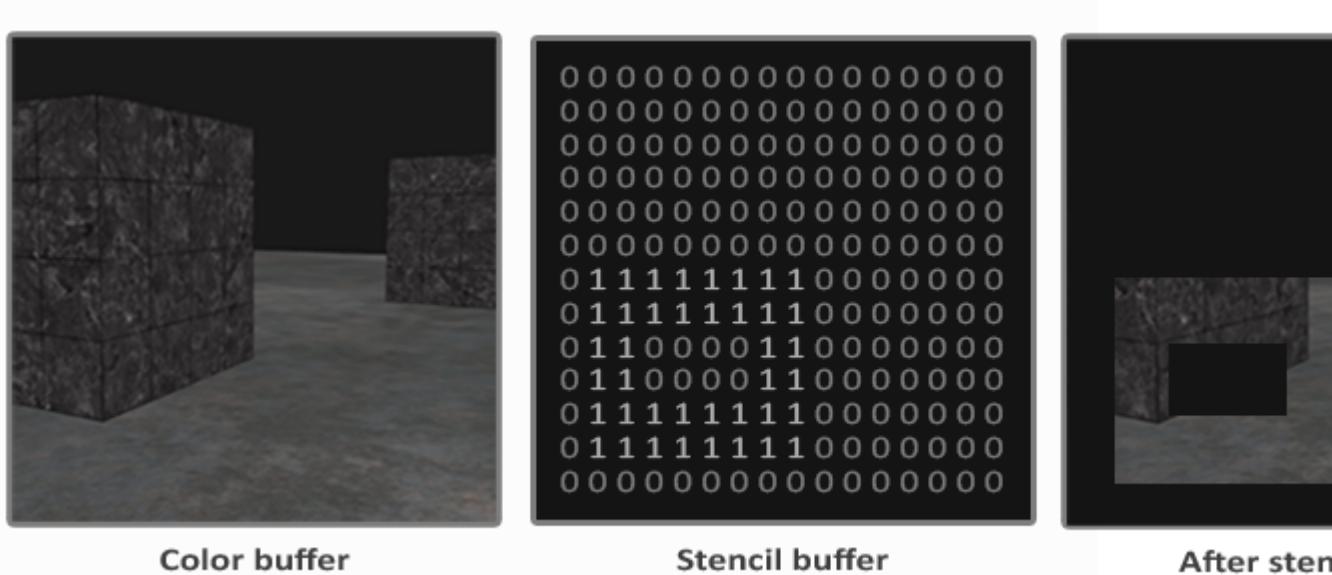
当片段着色器处理完片段之后，**模板测试(stencil test)** 就开始执行了，和深度测试一样，它能丢弃一些片段。仍然保留下来的片段进入深度测试阶段，深度测试可能丢弃更多。模板测试基于另一个缓冲，这个缓冲叫做**模板缓冲(stencil buffer)**，我们被允许在渲染时更新它来获取有意思的效果。

模板缓冲中的模板值（**stencil value**）通常是 **8** 位的，因此每个片段（像素）共有 **256** 种不同的模板值（译注：8 位就是 **1** 字节大小，因此和 **char** 的容量一样是 **256** 个不同值）。这样我们就能将这些模板值设置为我们链接的，然后在模板测试时根据这个模板值，我们就可以决定丢弃或保留它了。

### Important

每个窗口库都需要为你设置模板缓冲。**GLFW** 自动做了这件事，所以你不必告诉 **GLFW** 去创建它，但是其他库可能没默认创建模板库，所以一定要查看你使用的库的文档。

下面是一个模板缓冲的简单例子：



模板缓冲先清空模板缓冲设置所有片段的模板值为 0，然后开启矩形片段用 1 填充。场景中的模板值为 1 的那些片段才会被渲染（其他的都被丢弃）。

无论我们在渲染哪里的片段，模板缓冲操作都允许我们把模板缓冲设置为一个特定值。改变模板缓冲的内容实际上就是对模板缓冲进行写入。在同一次（或接下来的）渲染迭代我们可以读取这些值来决定丢弃还是保留这些片段。当使用模板缓冲的时候，你可以随心所欲，但是需要遵守下面的原则：

- 开启模板缓冲写入。
- 渲染物体，更新模板缓冲。
- 关闭模板缓冲写入。
- 渲染（其他）物体，这次基于模板缓冲内容丢弃特定片段。

使用模板缓冲我们可以基于场景中已经绘制的片段，来决定是否丢弃特定的片段。

你可以开启 `GL_STENCIL_TEST` 来开启模板测试。接着所有渲染函数调用都会以这样或那样的方式影响到模板缓冲。

```
glEnable(GL_STENCIL_TEST);
```

要注意的是，像颜色和深度缓冲一样，在每次循环，你也得清空模板缓冲。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
```

```
GL_STENCIL_BUFFER_BIT);
```

同时，和深度测试的 `glDepthMask` 函数一样，模板缓冲也有一个相似函数。

`glStencilMask` 允许我们给模板值设置一个位遮罩（**bitmask**），它与模板值进行按位与（**and**）运算决定缓冲是否可写。默认设置的位遮罩都是 1，这样就不会影响输出，但是如果我们设置为 0x00，所有写入深度缓冲最后都是 0。这和深度缓冲的 `glDepthMask(GL_FALSE)` 很类似：

```
// 0xFF == 0b11111111
```

```
// 此时，模板值与它进行按位与运算结果是模板值，模板缓冲可写
```

```
glStencilMask(0xFF);
```

```
// 0x00 == 0b00000000 == 0
```

```
// 此时，模板值与它进行按位与运算结果是 0，模板缓冲不可写
```

```
glStencilMask(0x00);
```

大多数情况你的模板遮罩（**stencil mask**）写为 0x00 或 0xFF 就行，但是最好知道有一个选项可以自定义位遮罩。

## 模板函数（**stencil functions**）

和深度测试一样，我们也有几个不同控制权，决定何时模板测试通过或失败以及它怎样影响模板缓冲。一共有两种函数可供我们使用去配置模板测试：

`glStencilFunc` 和 `glStencilOp`。

`void glStencilFunc(GLenum func, GLint ref, GLuint mask)` 函数有三个参数：

- **func:** 设置模板测试操作。这个测试操作应用到已经储存的模板值和 `glStencilFunc` 的 `ref` 值上，可用的选项是：`GL_NEVER`、`GL_EQUAL`、`GL_GREATER`、`GL_GEQUAL`、`GL_EQUAL`、`GL_NOTEQUAL`、`GL_ALWAYS`。它们的语义和深度缓冲的相似。
- **ref:** 指定模板测试的引用值。模板缓冲的内容会与这个值对比。

- **mask:** 指定一个遮罩，在模板测试对比引用值和储存的模板值前，对它们进行按位与（and）操作，初始设置为 1。

在上面简单模板的例子里，方程应该设置为：

```
glStencilFunc(GL_EQUAL, 1, 0xFF)
```

它会告诉 OpenGL，无论何时，一个片段模板值等于(`GL_EQUAL`)引用值 `1`，片段就能通过测试被绘制了，否则就会被丢弃。

但是 `glStencilFunc` 只描述了 OpenGL 对模板缓冲做什么，而不是描述我们如何更新缓冲。这就需要 `glStencilOp` 登场了。

`void glStencilOp(GLenum sfail, GLenum dpfail, GLenum dpass)` 函数包含三个选项，我们可以指定每个选项的动作：

- **sfail:** 如果模板测试失败将采取的动作。
- **dpfail:** 如果模板测试通过，但是深度测试失败时采取的动作。
- **dpass:** 如果深度测试和模板测试都通过，将采取的动作。

每个选项都可以使用下列任何一个动作。

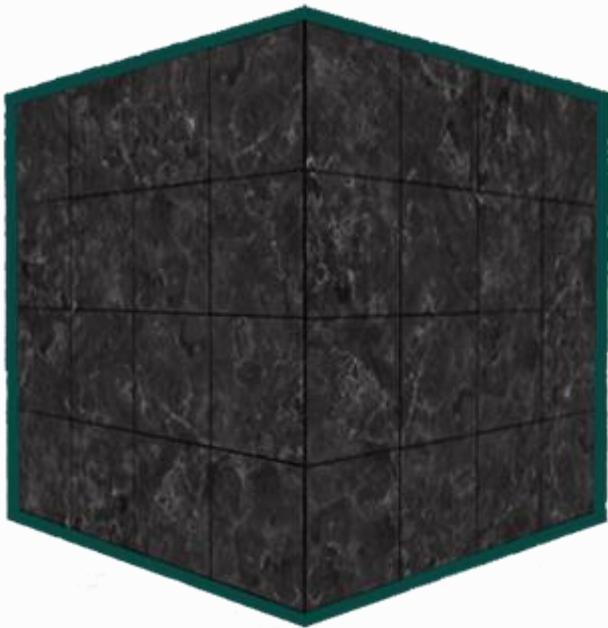
操作	描述
GL_KEEP	保持现有的模板值
GL_ZERO	将模板值置为 0
GL_REPLACE	将模板值设置为用 <code>glStencilFunc</code> 函数设置的 <code>ref</code> 值
GL_INCR	如果模板值不是最大值就将模板值+1
GL_INCR_WRAP	与 <code>GL_INCR</code> 一样将模板值+1，如果模板值已经是最大值则设为 0
GL_DECR	如果模板值不是最小值就将模板值-1
GL_DECR_WRAP	与 <code>GL_DECR</code> 一样将模板值-1，如果模板值已经是最小值则设为最大值
GL_INVERT	Bitwise inverts the current stencil buffer value.

`glStencilOp` 函数默认设置为 `(GL_KEEP, GL_KEEP, GL_KEEP)`，所以任何测试的任何结果，模板缓冲都会保留它的值。默认行为不会更新模板缓冲，所以如果你想写入模板缓冲的话，你必须像任意选项指定至少一个不同的动作。

使用 `glStencilFunc` 和 `glStencilOp`，我们就可以指定在什么时候以及我们打算怎么样去更新模板缓冲了，我们也可以指定何时让测试通过或不通过。什么时候片段会被抛弃。

## 物体轮廓

看了前面的部分你未必能理解模板测试是如何工作的，所以我们会展示一个用模板测试实现的一个特别的和有用的功能，叫做物体轮廓（object outlining）。



物体轮廓就像它的名字所描述的那样，它能够给每个（或一个）物体创建一个有颜色的边。在策略游戏中当你打算选择一个单位的时候它特别有用。给物体加上轮廓的步骤如下：

1. 在绘制物体前，把模板方程设置为 `GL_ALWAYS`，用 1 更新物体将被渲染的片段。
2. 渲染物体，写入模板缓冲。
3. 关闭模板写入和深度测试。
4. 每个物体放大一点点。
5. 使用一个不同的片段着色器用来输出一个纯颜色。
6. 再次绘制物体，但只是当它们的片段的模板值不为 1 时才进行。
7. 开启模板写入和深度测试。

这个过程将每个物体的片段模板缓冲设置为 1，当我们绘制边框的时候，我们基本上绘制的是放大的版本的物体的通过测试的地方，放大的版本绘制后物体就会有

一个边。我们基本会使用模板缓冲丢弃所有的不是原来物体的片段的放大的版本内容。

我们先来创建一个非常基本的片段着色器，它输出一个边框颜色。我们简单地设置一个固定的颜色值，把这个着色器命名为 `shaderSingleColor`:

```
void main()
{
 outColor = vec4(0.04, 0.28, 0.26, 1.0);
}
```

我们只打算给两个箱子加上边框，所以我们不会对地面做什么。这样我们要先绘制地面，然后再绘制两个箱子（同时写入模板缓冲），接着我们绘制放大的箱子（同时丢弃前面已经绘制的箱子的那部分片段）。

我们先开启模板测试，设置模板、深度测试通过或失败时才采取动作：

```
glEnable(GL_DEPTH_TEST);

glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
```

如果任何测试失败我们都什么也不做，我们简单地保持深度缓冲中当前所储存着的值。如果模板测试和深度测试都成功了，我们就将储存着的模板值替换为 `1`，我们要用 `glStencilFunc` 来做这件事。

我们清空模板缓冲为 `0`，为箱子的所有绘制的片段的模板缓冲更新为 `1`:

```
glStencilFunc(GL_ALWAYS, 1, 0xFF); // 所有片段都要写入模板缓冲

glStencilMask(0xFF); // 设置模板缓冲为可写状态

normalShader.Use();

DrawTwoContainers();
```

使用 `GL_ALWAYS` 模板测试函数，我们确保箱子的每个片段用模板值 `1` 更新模板缓冲。因为片段总会通过模板测试，在我们绘制它们的地方，模板缓冲用引用值更新。

现在箱子绘制之处，模板缓冲更新为 1 了，我们将要绘制放大的箱子，但是这次关闭模板缓冲的写入：

```
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00); // 禁止修改模板缓冲
glDisable(GL_DEPTH_TEST);
shaderSingleColor.Use();
DrawTwoScaledUpContainers();
```

我们把模板方程设置为 `GL_NOTEQUAL`，它保证我们只箱子上不等于 1 的部分，这样只绘制前面绘制的箱子外围的那部分。注意，我们也要关闭深度测试，这样放大的的箱子也就是边框才不会被地面覆盖。

做完之后还要保证再次开启深度缓冲。

场景中的物体边框的绘制方法最后看起来像这样：

```
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); // 绘制地板时确保关闭模板缓冲的写入
normalShader.Use();
DrawFloor();

glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);
```

```
DrawTwoContainers();
```

```
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
```

```
glStencilMask(0x00);
```

```
glDisable(GL_DEPTH_TEST);
```

```
shaderSingleColor.Use();
```

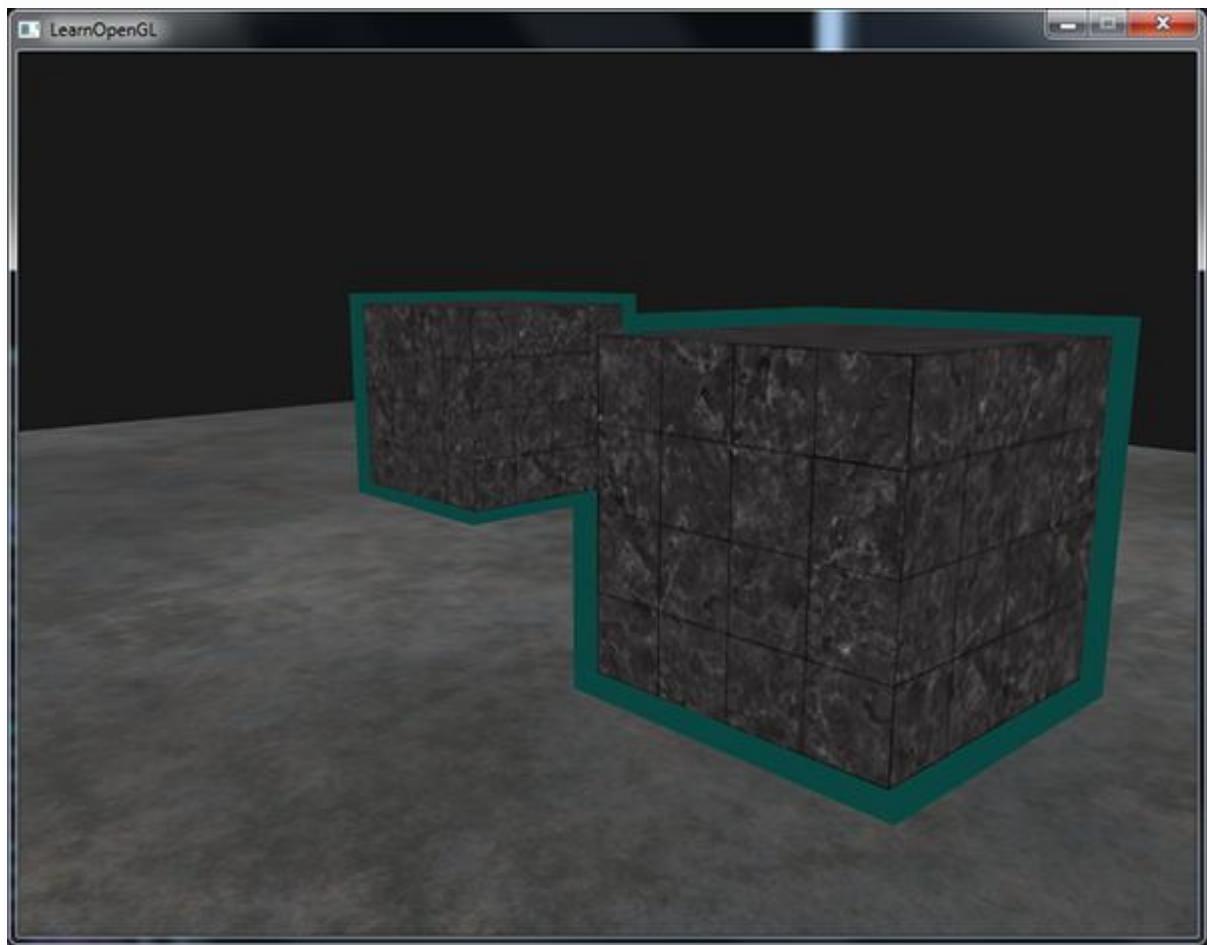
```
DrawTwoScaledUpContainers();
```

```
glStencilMask(0xFF);
```

```
	glEnable(GL_DEPTH_TEST);
```

理解这段代码后面的模板测试的思路并不难以理解。如果还不明白尝试再仔细阅读上面的部分，尝试理解每个函数的作用，现在你已经看到了它的使用方法的例子。

这个边框的算法的结果在深度测试教程的那个场景中，看起来像这样：



在这里[查看源码](#)和[着色器](#)，看看完整的物体边框算法是怎样的。

### Important

你可以看到两个箱子边框重合通常正是我们希望得到的（想想策略游戏中，我们打算选择 10 个单位；我们通常会希望把边界合并）。如果你想要让每个物体都有自己的边界那么你需要为每个物体清空模板缓冲，创造性地使用深度缓冲。

你目前看到的物体边框算法在一些游戏中显示备选物体（想象策略游戏）非常常用，这样的算法可以在一个模型类中轻易实现。你可以简单地在模型类设置一个布尔类型的标识来决定是否绘制边框。如果你想要更多的创造性，你可以使用后处理（post-processing）过滤比如高斯模糊来使边框看起来更自然。

除了物体边框以外，模板测试还有很多其他的应用目的，比如在后视镜中绘制纹理，这样它会很好的适合镜子的形状，比如使用一种叫做 **shadow volumes** 的模板缓冲技术渲染实时阴影。模板缓冲在我们的已扩展的 OpenGL 工具箱中给我们提供了另一种好用工具。

# 混合 (Blending)

原文	<a href="#">Blending</a>
作者	<a href="#">JoeyDeVries</a>
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

在 OpenGL 中，物体透明技术通常被叫做混合(Blending)。透明是物体（或物体的一部分）非纯色而是混合色，这种颜色来自于不同浓度的自身颜色和它后面的物体颜色。一个有色玻璃窗就是一种透明物体，玻璃有自身的颜色，但是最终的颜色包含了所有玻璃后面的颜色。这也正是混合这名称的出处，因为我们将多种（来自于不同物体）颜色混合为一个颜色，透明使得我们可以看穿物体。



Full transparent window

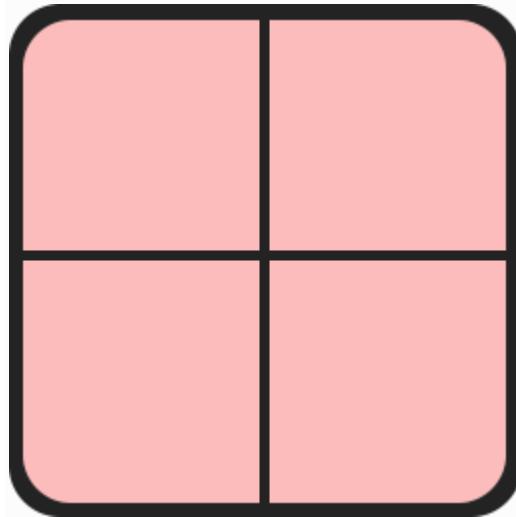


Partially transparent window

透明物体可以是完全透明（它使颜色完全穿透）或者半透明的（它使颜色穿透的同时也显示自身颜色）。一个物体的透明度，被定义为它的颜色的 **alpha** 值。**alpha** 颜色值是一个颜色向量的第四个元素，你可能已经看到很多了。在这个教程前，我们一直把这个元素设置为 **1.0**，这样物体的透明度就是 **0.0**，同样的，当 **alpha** 值是 **0.0** 时就表示物体是完全透明的，**alpha** 值为 **0.5** 时表示物体的颜色由 **50%** 的自身的颜色和 **50%** 的后面的颜色组成。

我们之前所使用的纹理都是由 **3** 个颜色元素组成的：红、绿、蓝，但是有些纹理同样有一个内嵌的 **alpha** 通道，它为每个纹理像素(**Texel**)包含着一个 **alpha** 值。

这个 **alpha** 值告诉我们纹理的哪个部分有透明度，以及这个透明度有多少。例如，下面的窗子纹理的玻璃部分的 **alpha** 值为 0.25(它的颜色是完全红色，但是由于它有 75 的透明度，它会很大程度上反映出网站的背景色，看起来就不那么红了)，角落部分 **alpha** 是 0.0。



我们很快就会把这个窗子纹理加到场景中，但是首先，我们将讨论一点简单的技术来实现纹理的半透明，也就是完全透明和完全不透明。

## 忽略片段

有些图像并不关心半透明度，但也想基于纹理的颜色值显示一部分。例如，创建像草这种物体你不需要花费很大力气，通常把一个草的纹理贴到 2D 四边形上，然后把这个四边形放置到你的场景中。可是，草并不是像 2D 四边形这样的形状，而只需要显示草纹理的一部分而忽略其他部分。

下面的纹理正是这样的纹理，它既有完全不透明的部分（**alpha** 值为 1.0）也有完全透明的部分（**alpha** 值为 0.0），而没有半透明的部分。你可以看到没有草的部分，图片显示了网站的背景色，而不是它自身的那部分颜色。



所以，当向场景中添加像这样的纹理时，我们不希望看到一个方块图像，而是只显示实际的纹理像素，剩下的部分可以被看穿。我们要忽略(丢弃)纹理透明部分的像素，不必将这些片段储存到颜色缓冲中。在此之前，我们还要学一下如何加载一个带有透明像素的纹理。

加载带有 `alpha` 值的纹理我们需要告诉 `SOIL`，去加载 `RGBA` 元素图像，而不再是 `RGB` 元素的。`SOIL` 能以 `RGBA` 的方式加载大多数没有 `alpha` 值的纹理，它会将这些像素的 `alpha` 值设为了 `1.0`。

```
unsigned char * image = SOIL_load_image(path, &width, &height, 0,
SOIL_LOAD_RGBA);
```

不要忘记还要改变 `OpenGL` 生成的纹理：

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, image);
```

保证你在片段着色器中获取了纹理的所有 4 个颜色元素，而不仅仅是 RGB 元素：

```
void main()
{
 // color = vec4(vec3(texture(texture1, TexCoords)), 1.0);
 color = texture(texture1, TexCoords);
}
```

现在我们知道了如何加载透明纹理，是时候试试在深度测试教程里那个场景中添加几根草了。

我们创建一个 `std::vector`，并向里面添加几个 `glm::vec3` 变量，来表示草的位置：

```
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));
```

一个单独的四边形被贴上草的纹理，这并不能完美的表现出真实的草，但是比起加载复杂的模型还是要高效很多，利用一些小技巧，比如在同一个地方添加多个不同朝向的草，还是能获得比较好的效果的。

由于草纹理被添加到四边形物体上，我们需要再次创建另一个 VAO，向里面填充 VBO，以及设置合理的顶点属性指针。在我们绘制完地面和两个立方体后，我们就来绘制草叶：

```
glBindVertexArray(vegetationVAO);

glBindTexture(GL_TEXTURE_2D, grassTexture);

for(GLuint i = 0; i < vegetation.size(); i++)

{

 model = glm::mat4();

 model = glm::translate(model, vegetation[i]);

 glUniformMatrix4fv(modelLoc, 1, GL_FALSE,

glm::value_ptr(model));

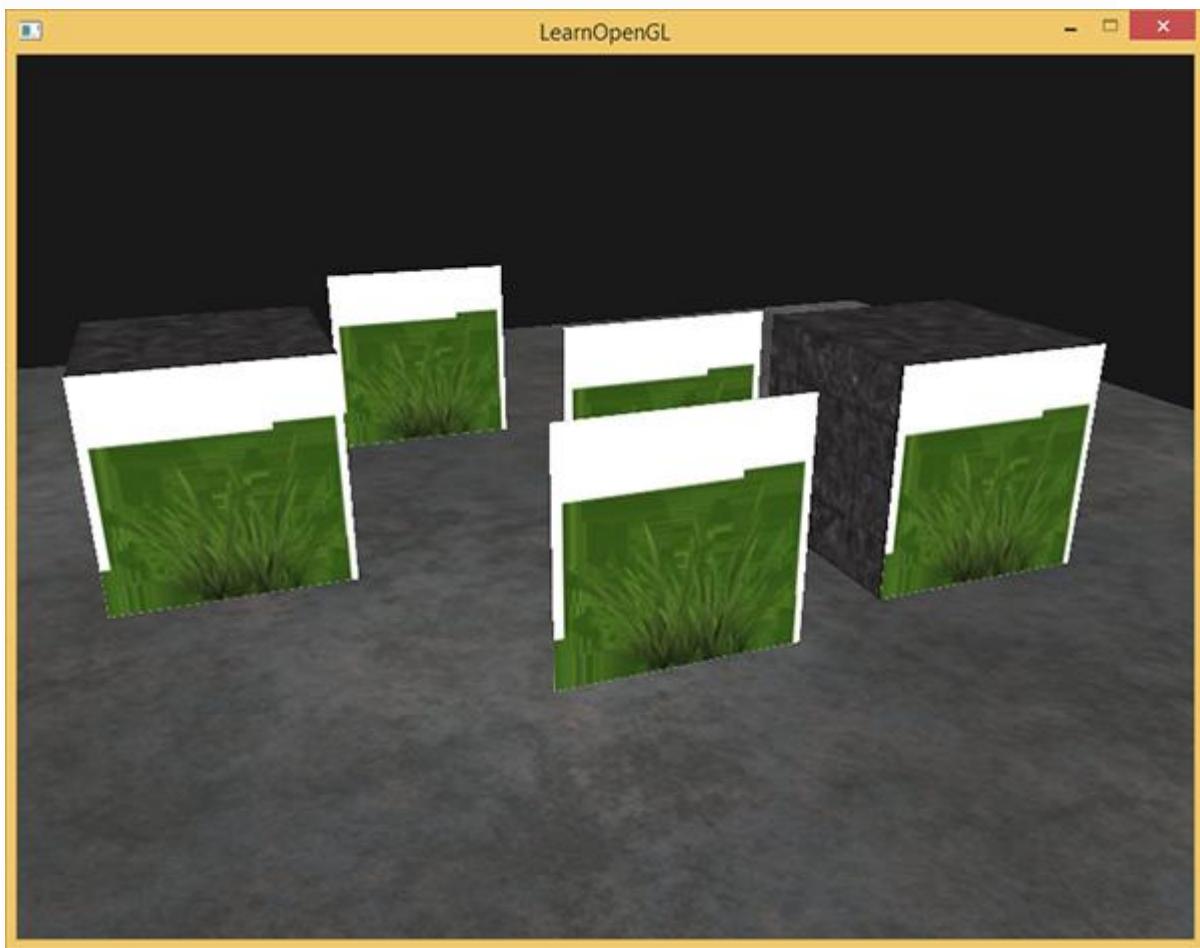
 glDrawArrays(GL_TRIANGLES, 0, 6);

}

glBindVertexArray(0);
```

运行程序你将看

到：



出现这种情况是因为 OpenGL 默认是不知道如何处理 alpha 值的，不知道何时忽略(丢弃)它们。我们不得不手动做这件事。幸运的是这很简单，感谢着色器，GLSL 为我们提供了 `discard` 命令，它保证了片段不会被进一步处理，这样就不会进入颜色缓冲。有了这个命令我们就可以在片段着色器中检查一个片段是否在一定的阈限下的 `alpha` 值，如果有，那么丢弃这个片段，就好像它不存在一样：

```
#version 330 core

in vec2 TexCoords;

out vec4 color;

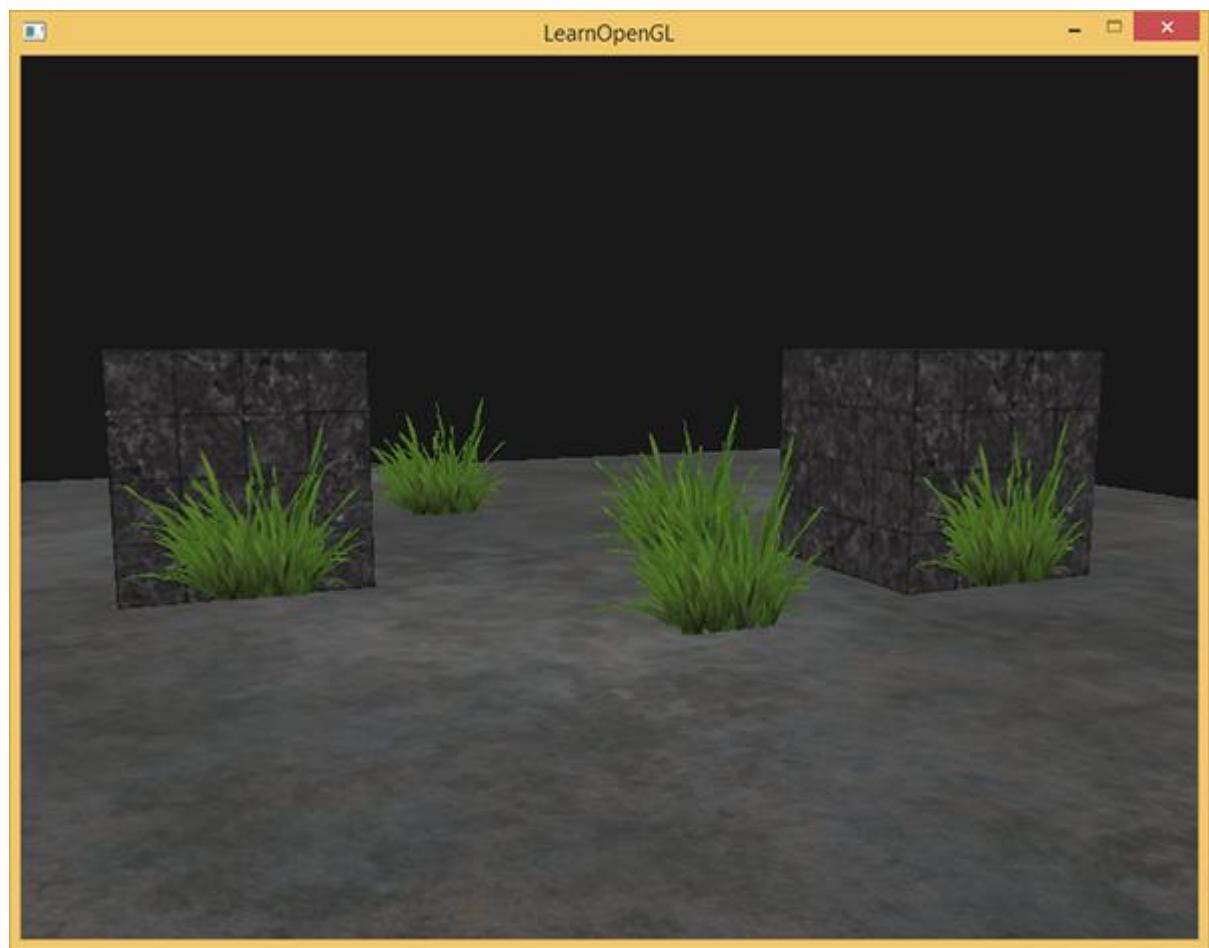
uniform sampler2D texture1;
```

```
void main()
{
 vec4 texColor = texture(texture1, TexCoords);

 if(texColor.a < 0.1)
 discard;

 color = texColor;
}
```

在这儿我们检查被采样纹理颜色包含着一个低于 0.1 这个阈限的 alpha 值，如果有，就丢弃这个片段。这个片段着色器能够保证我们只渲染哪些不是完全透明的片段。现在我们来看看效果：



**Important**

需要注意的是，当采样纹理边缘的时候，OpenGL 在边界值和下一个重复的纹理的值之间进行插值（因为我们把它的放置方式设置成了 `GL_REPEAT`）。这样就行了，但是由于我们使用的是透明值，纹理图片的上部获得了它的透明值是与底边的纯色值进行插值的。结果就是一个有点半透明的边，你可以从我们的纹理四边形的四周看到。为了防止它的出现，当你使用 `alpha` 纹理的时候要把纹理环绕方式设置为 `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[在这里得到源码](#)。

## 混合

上述丢弃片段的方式，不能使我们获得渲染半透明图像，我们要么渲染出像素，要么完全地丢弃它。为了渲染出不同的透明度级别，我们需要开启**混合**(Blending)。像大多数 OpenGL 的功能一样，我们可以开启 `GL_BLEND` 来启用混合功能：

```
 glEnable(GL_BLEND);
```

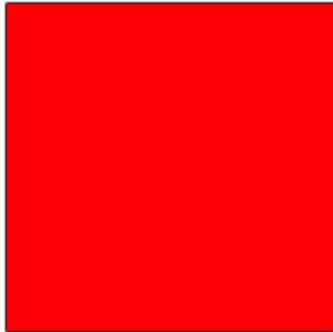
开启混合后，我们还需要告诉 OpenGL 它该如何混合。

OpenGL 以下面的方程进行混合：

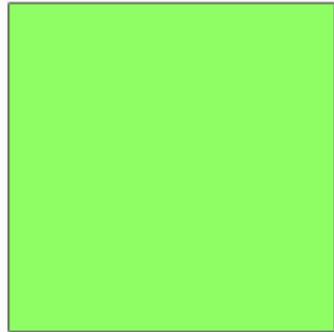
$$C_{\text{result}} = C_{\text{source}} * F_{\text{source}} + C_{\text{destination}} * F_{\text{destination}}$$

- **$C_{\text{source}}$** : 源颜色向量。这是来自纹理的本来的颜色向量。
- **$C_{\text{destination}}$** : 目标颜色向量。这是储存在颜色缓冲中当前位置的颜色向量。
- **$F_{\text{source}}$** : 源因子。设置了对源颜色的 `alpha` 值影响。
- **$F_{\text{destination}}$** : 目标因子。设置了对目标颜色的 `alpha` 影响。

片段着色器运行完成并且所有的测试都通过以后，混合方程才能自由执行片段的颜色输出，当前它在颜色缓冲中（前面片段的颜色在当前片段之前储存）。源和目标颜色会自动被 OpenGL 设置，而源和目标因子可以让我们自由设置。我们来看一个简单的例子：



(1.0, 0.0, 0.0, 1.0)



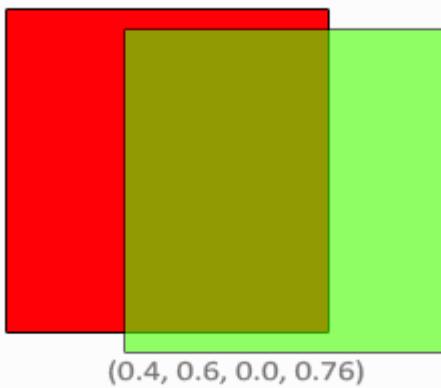
(0.0, 1.0, 0.0, 0.6)

我们有两个方块，我们希望在红色方块上绘制绿色方块。红色方块会成为源颜色（它会先进入颜色缓冲），我们将把红色方块上的颜色设为 0.6。

那么问题来了：我们怎样来设置因子呢？我们起码要把绿色方块乘以它的 alpha 值，所以我们打算把  $F_{source}$  设置为源颜色向量的 alpha 值：0.6。接着，让目标方块的浓度等于剩下的 alpha 值。如果最终的颜色中绿色方块的浓度为 60%，我们就把红色的浓度设为 40% ( $1.0 - 0.6$ )。所以我们把  $F_{destination}$  设置为 1 减去源颜色向量的 alpha 值。方程将变成：

$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} *$$

最终方块结合部分包含了 60% 的绿色和 40% 的红色，得到一种脏兮兮的颜色：



最后的颜色被储存到颜色缓冲中，取代先前的颜色。

这个方案不错，但我们怎样告诉 OpenGL 来使用这样的因子呢？恰好有一个叫做 `glBlendFunc` 的函数。

`void glBlendFunc(GLenum sfactor, GLenum dfactor)` 接收两个参数，来设置源(`source`)和目标(`destination`)因子。OpenGL 为我们定义了很多选项，我们把最常用的列在下面。注意，颜色常数向量  $C^-$  `constant` 可以用 `glBlendColor` 函数分开来设置。

Option	Value
<code>GL_ZERO</code>	0
<code>GL_ONE</code>	1
<code>GL_SRC_COLOR</code>	颜色 $C^-$ source.
<code>GL_ONE_MINUS_SRC_COLOR</code>	$1 - C^-$ source.
<code>GL_DST_COLOR</code>	$C^-$ destination
<code>GL_ONE_MINUS_DST_COLOR</code>	$1 - C^-$ destination.
<code>GL_SRC_ALPHA</code>	$C^-$ source 的 alpha 值
<code>GL_ONE_MINUS_SRC_ALPHA</code>	$1 - C^-$ source 的 alpha 值
<code>GL_DST_ALPHA</code>	$C^-$ destination 的 alpha 值
<code>GL_ONE_MINUS_DST_ALPHA</code>	$1 - C^-$ destination 的 alpha 值

Option	Value
GL_CONSTANT_COLOR	$C^-$ constant.
GL_ONE_MINUS_CONSTANT_COLOR	$1 - C^-$ constant
GL_CONSTANT_ALPHA	$C^-$ constant 的 alpha 值
GL_ONE_MINUS_CONSTANT_ALPHA	$1 - C^-$ constant 的 alpha 值

为从两个方块获得混合结果，我们打算把源颜色的 **alpha** 给源因子， $1-\text{alpha}$  给目标因子。调整到 `glBlendFunc` 之后就像这样：

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

也可以为 **RGB** 和 **alpha** 通道各自设置不同的选项，使用 `glBlendFuncSeparate`：

```
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE,
```

```
GL_ZERO);
```

这个方程就像我们之前设置的那样，设置了 **RGB** 元素，但是只让最终的 **alpha** 元素被源 **alpha** 值影响到。

**OpenGL** 给了我们更多的自由，我们可以改变方程源和目标部分的操作符。现在，源和目标元素已经相加了。如果我们愿意的话，我们还可以把它们相减。

`void glBlendEquation(GLenum mode)` 允许我们设置这个操作，有 3 种可行的选项：

- **GL\_FUNC\_ADD**: 默认的，彼此元素相加： $C^- \text{result} = \text{Src} + \text{Dst}$ .
- **GL\_FUNC\_SUBTRACT**: 彼此元素相减： $C^- \text{result} = \text{Src} - \text{Dst}$ .
- **GL\_FUNC\_REVERSE\_SUBTRACT**: 彼此元素相减，但顺序相反： $C^- \text{result} = \text{Dst} - \text{Src}$ .

通常我们可以简单地省略 `glBlendEquation` 因为 **GL\_FUNC\_ADD** 在大多数时候就是我们想要的，但是如果你真想尝试努力打破主流常规，其他的方程或许符合你的要求。

## 渲染半透明纹理

现在我们知道 OpenGL 如何处理混合，是时候把我们的知识运用起来了，我们来添加几个半透明窗子。我们会使用本教程开始时用的那个场景，但是不再渲染草纹理，取而代之的是来自教程开始处半透明窗子纹理。

首先，初始化时我们需要开启混合，设置合适和混合方程：

```
glEnable(GL_BLEND);

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

由于我们开启了混合，就不需要丢弃片段了，所以我们把片段着色器设置为原来的那个版本：

```
#version 330 core

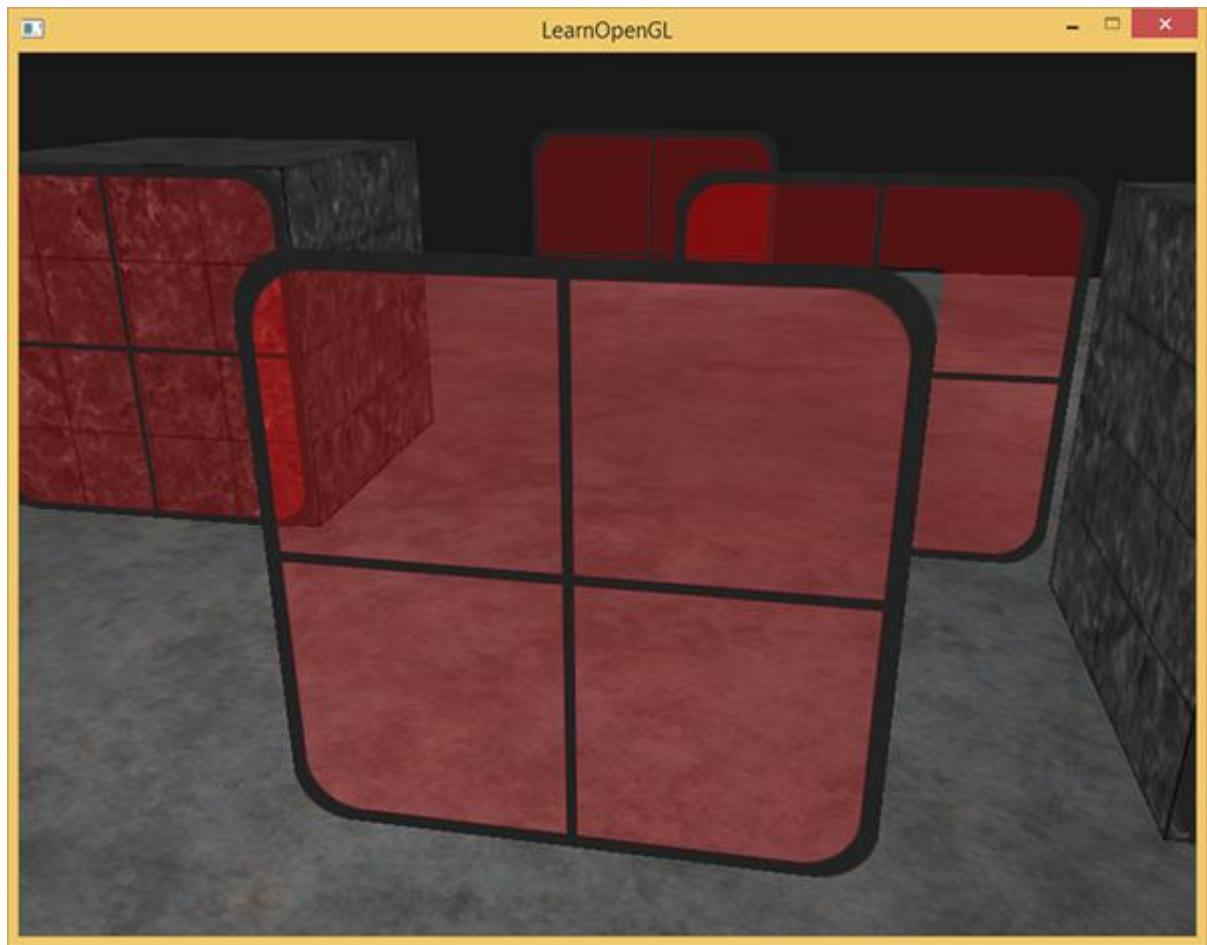
in vec2 TexCoords;

out vec4 color;

uniform sampler2D texture1;

void main()
{
 color = texture(texture1, TexCoords);
}
```

这一次（无论 OpenGL 什么时候去渲染一个片段），它都根据 `alpha` 值，把当前片段的颜色和颜色缓冲中的颜色进行混合。因为窗子的玻璃部分的纹理是半透明的，我们应该可以透过玻璃看到整个场景。



如果你仔细看看，就会注意到有些不对劲。前面的窗子透明部分阻塞了后面的。为什么会这样？

原因是深度测试在与混合的一同工作时出现了点状况。当写入深度缓冲的时候，深度测试不关心片段是否有透明度，所以透明部分被写入深度缓冲，就和其他值没什么区别。结果是整个四边形的窗子被检查时都忽视了透明度。即便透明部分应该显示出后面的窗子，深度缓冲还是丢弃了它们。

所以我们不能简简单单地去渲染窗子，我们期待着深度缓冲为我们解决这所有问题；这也正是混合之处代码不怎么好看的原因。为保证前面窗子显示了它后面的窗子，我们必须首先绘制后面的窗子。这意味着我们必须手工调整窗子的顺序，从远到近地逐个渲染。

### Important

对于全透明物体，比如草叶，我们选择简单的丢弃透明像素而不是混合，这样就减少了令我们头疼的问题（没有深度测试问题）。

## 别打乱顺序

要让混合在多物体上有效，我们必须先绘制最远的物体，最后绘制最近的物体。普通的无混合物体仍然可以使用深度缓冲正常绘制，所以不必给它们排序。我们一定要保证它们在透明物体前绘制好。当无透明度物体和透明物体一起绘制的时候，通常要遵循以下原则：

先绘制所有不透明物体。为所有透明物体排序。按顺序绘制透明物体。一种排序透明物体的方式是，获取一个物体到观察者透视图的距离。这可以通过获取摄像机的位置向量和物体的位置向量来得到。接着我们就可以把它和相应的位置向量一起储存到一个 **map** 数据结构（**STL** 库）中。**map** 会自动基于它的键排序它的值，所以当我们把它们的距离作为键添加到所有位置中后，它们就自动按照距离值排序了：

```
std::map<float, glm::vec3> sorted;

for (GLuint i = 0; i < windows.size(); i++) // windows contains all

window positions

{

 GLfloat distance = glm::length(camera.Position - windows[i]);

 sorted[distance] = windows[i];

}
```

最后产生了一个容器对象，基于它们距离从低到高储存了每个窗子的位置。

随后当渲染的时候，我们逆序获取到每个 **map** 的值（从远到近），然后以正确的绘制相应的窗子：

```
for(std::map<float,glm::vec3>::reverse_iterator it = sorted.rbegin();

it != sorted.rend(); ++it)

{

 model = glm::mat4();
```

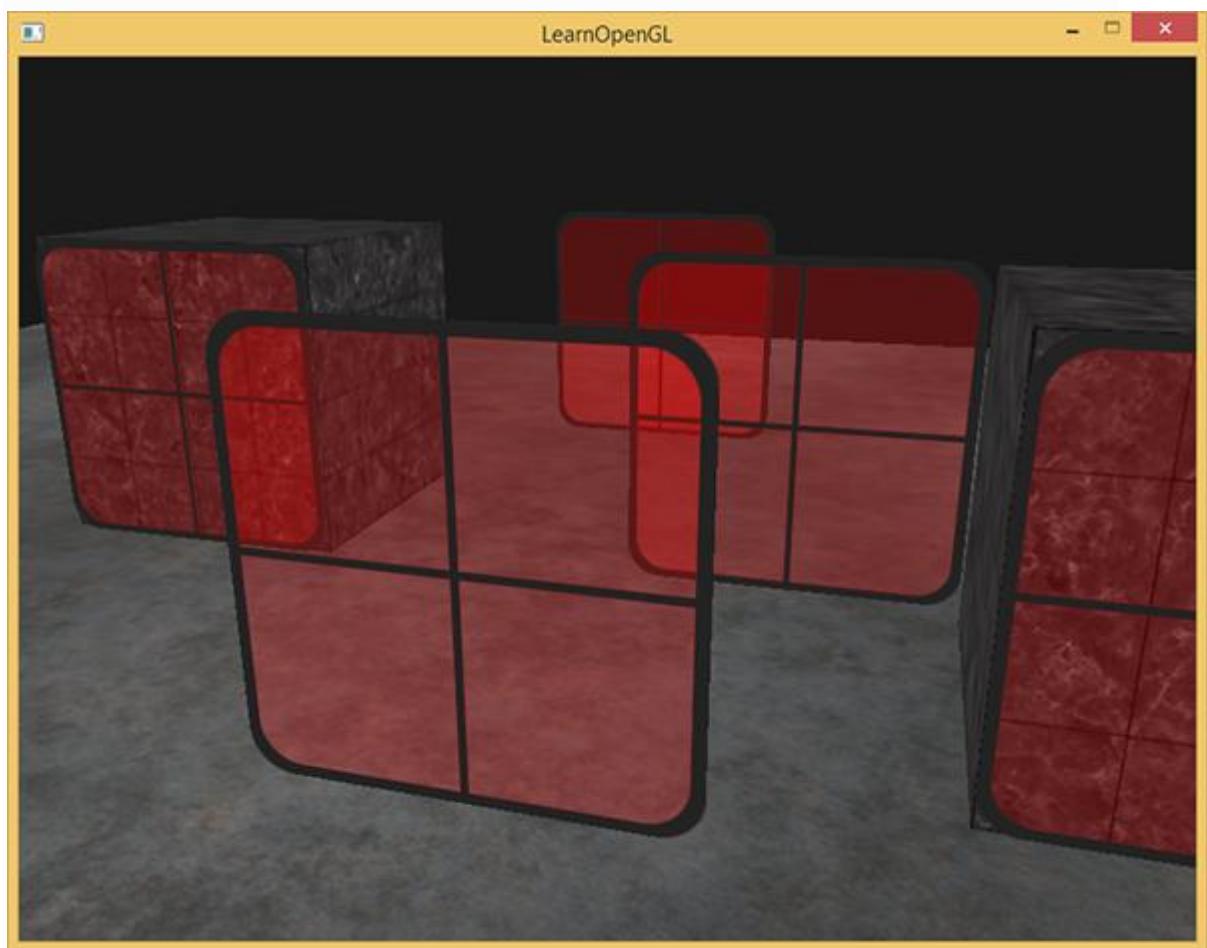
```
model = glm::translate(model, it->second);

glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model));

glDrawArrays(GL_TRIANGLES, 0, 6);

}
```

我们从 `map` 得来一个逆序的迭代器，迭代出每个逆序的条目，然后把每个窗子的四边形平移到相应的位置。这个相对简单的方法对透明物体进行了排序，修正了前面的问题，现在场景看起来像这样：



你可以从[这里](#)得到完整的带有排序的源码。

虽然这个按照它们的距离对物体进行排序的方法在这个特定的场景中能够良好工作，但它不能进行旋转、缩放或者进行其他的变换，奇怪形状的物体需要一种不同的方式，而不能简单的使用位置向量。

在场景中排序物体是个有难度的技术，它很大程度上取决于你场景的类型，更不必说会耗费额外的处理能力了。完美地渲染带有透明和不透明的物体的场景并不那么容易。有更高级的技术例如次序无关透明度（**order independent transparency**），但是这超出了本教程的范围。现在你不得不采用普通的混合你的物体，但是如果你小心谨慎，并知道这个局限，你仍可以得到颇为合适的混合实现。

## 面剔除（Face culling）

原文	<a href="#">Face culling</a>
作者	JoeyDeVries
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

尝试在头脑中想象一下有一个 3D 立方体，你从任何一个方向去看它，最多可以同时看到多少个面。如果你的想象力不是过于丰富，你最终最多能数出来的面是 3 个。你可以从一个立方体的任意位置和方向上去看它，但是你永远不能看到多于 3 个面。所以我们为何还要去绘制那三个不会显示出来的 3 个面呢。如果我们可以以某种方式丢弃它们，我们会提高片段着色器超过 50% 的性能！

### Important

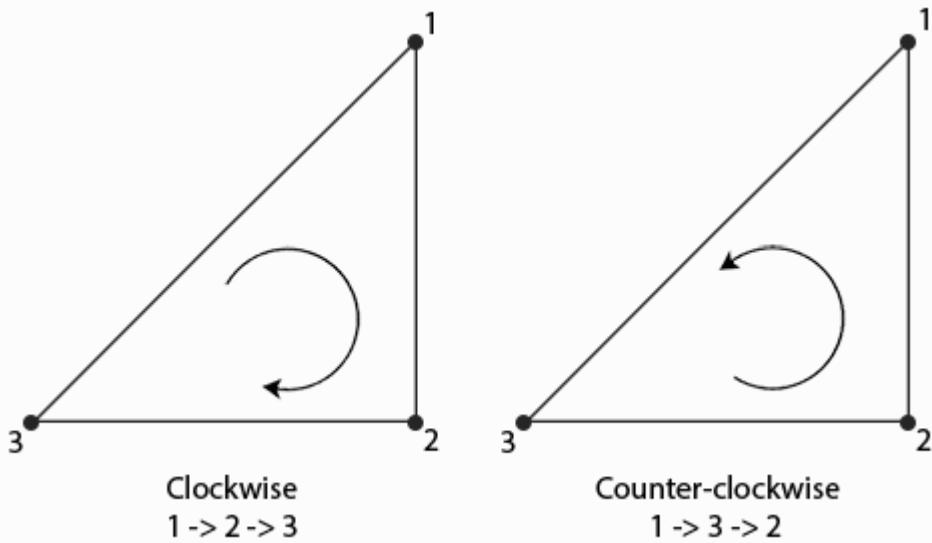
我们所说的是超过 50% 而不是 50%，因为从一个角度只有 2 个或 1 个面能够被看到。这种情况下我们就能够提高 50% 以上性能了。

这的确是个好主意，但是有个问题需要解决：我们如何知道某个面在观察者的视野中不会出现呢？如果我们去想象任何封闭的几何平面，它们都有两面，一面面向用户，另一面背对用户。假如我们只渲染面向观察者的面会怎样？

这正是面剔除(Face culling)所要做的。OpenGL 允许检查所有正面朝向（Front facing）观察者的面，并渲染它们，而丢弃所有背面朝向（Back facing）的面，这样就节约了我们很多片段着色器的命令（它们很昂贵！）。我们必须告诉 OpenGL 我们使用的哪个面是正面，哪个面是反面。OpenGL 使用一种聪明的手段解决这个问题——分析顶点数据的连接顺序（Winding order）。

## 顶点连接顺序（Winding order）

当我们定义一系列的三角顶点时，我们会把它们定义为一个特定的连接顺序，它们可能是顺时针的或逆时针的。每个三角形由 3 个顶点组成，我们从三角形的中间去看，从而把这三个顶点指定一个连接顺序。



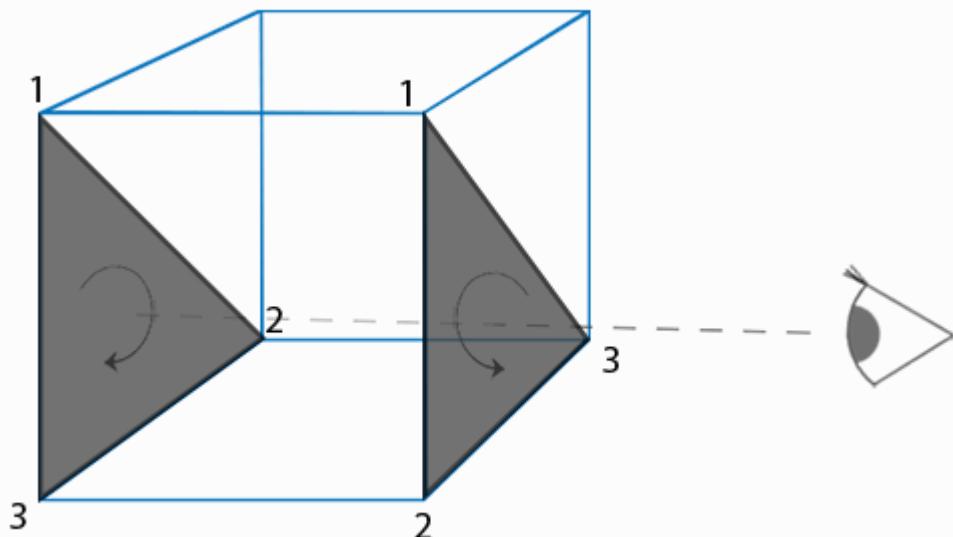
正如你所看到的那样，我们先定义了顶点 1，接着我们定义顶点 2 或 3，这个不同的选择决定了这个三角形的连接顺序。下面的代码展示出这点：

```
GLfloat vertices[] = {
 // 顺时针
 vertices[0], // vertex 1
 vertices[1], // vertex 2
 vertices[2], // vertex 3
 // 逆时针
 vertices[0], // vertex 1
 vertices[2], // vertex 3
 vertices[1] // vertex 2
};
```

每三个顶点都形成了一个包含着连接顺序的基本三角形。OpenGL 使用这个信息在渲染你的基本图形的时候决定这个三角形是三角形的正面还是三角形的背面。默认情况下，逆时针的顶点连接顺序被定义为三角形的正面。

当定义你的顶点顺序时，你如果定义能够看到的一个三角形，那它一定是正面朝向的，所以你定义的三角形应该是逆时针的，就像你直接面向这个三角形。把所有的顶点指定成这样是件炫酷的事，实际的顶点连接顺序是在光栅化阶段（Rasterization stage）计算的，所以当顶点着色器已经运行后。顶点就能够在观察者的观察点被看到。

我们指定了它们以后，观察者面对的所有三角形的顶点的连接顺序都是正确的，但是现在渲染的立方体另一面的三角形的顶点的连接顺序被反转。最终，我们所面对的三角形被视为正面朝向的三角形，后部的三角形被视为背面朝向的三角形。下图展示了这个效果：



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

在教程的开头，我们说过 OpenGL 可以丢弃背面朝向的三角形。现在我们知道了如何设置顶点的连接顺序，我们可以开始使用 OpenGL 默认关闭的面剔除选项了。

记住我们上一节所使用的立方体的定点数据不是以逆时针顺序定义的。所以我更新了顶点数据，好去反应为一个逆时针链接顺序，你可以[从这里复制它](#)。把所有三角的顶点都定义为逆时针是一个很好的习惯。

开启 OpenGL 的 `GL_CULL_FACE` 选项就能开启面剔除功能：

```
glEnable(GL_CULL_FACE);
```

从这儿以后，所有的不是正面朝向的面都会被丢弃（尝试飞入立方体看看，里面什么面都看不到了）。目前，在渲染片段上我们节约了超过 50% 的性能，但记住这只对像立方体这样的封闭形状有效。当我们绘制上个教程中那个草的时候，我们必须关闭面剔除，这是因为它的前、后面都必须是可见的。

OpenGL 允许我们改变剔除面的类型。要是我们剔除正面而不是背面会怎样？我们可以调用 `glCullFace` 来做这件事：

```
glCullFace(GL_BACK);
```

`glCullFace` 函数有三个可用的选项：

- `GL_BACK`: 只剔除背面。
- `GL_FRONT`: 只剔除正面。
- `GL_FRONT_AND_BACK`: 剔除背面和正面。

`glCullFace` 的初始值是 `GL_BACK`。另外，我们还可以告诉 OpenGL 使用顺时针而不是逆时针来表示正面，这通过 `glFrontFace` 来设置：

```
glFrontFace(GL_CCW);
```

默认值是 `GL_CCW`，它代表逆时针，`GL_CW` 代表顺时针顺序。

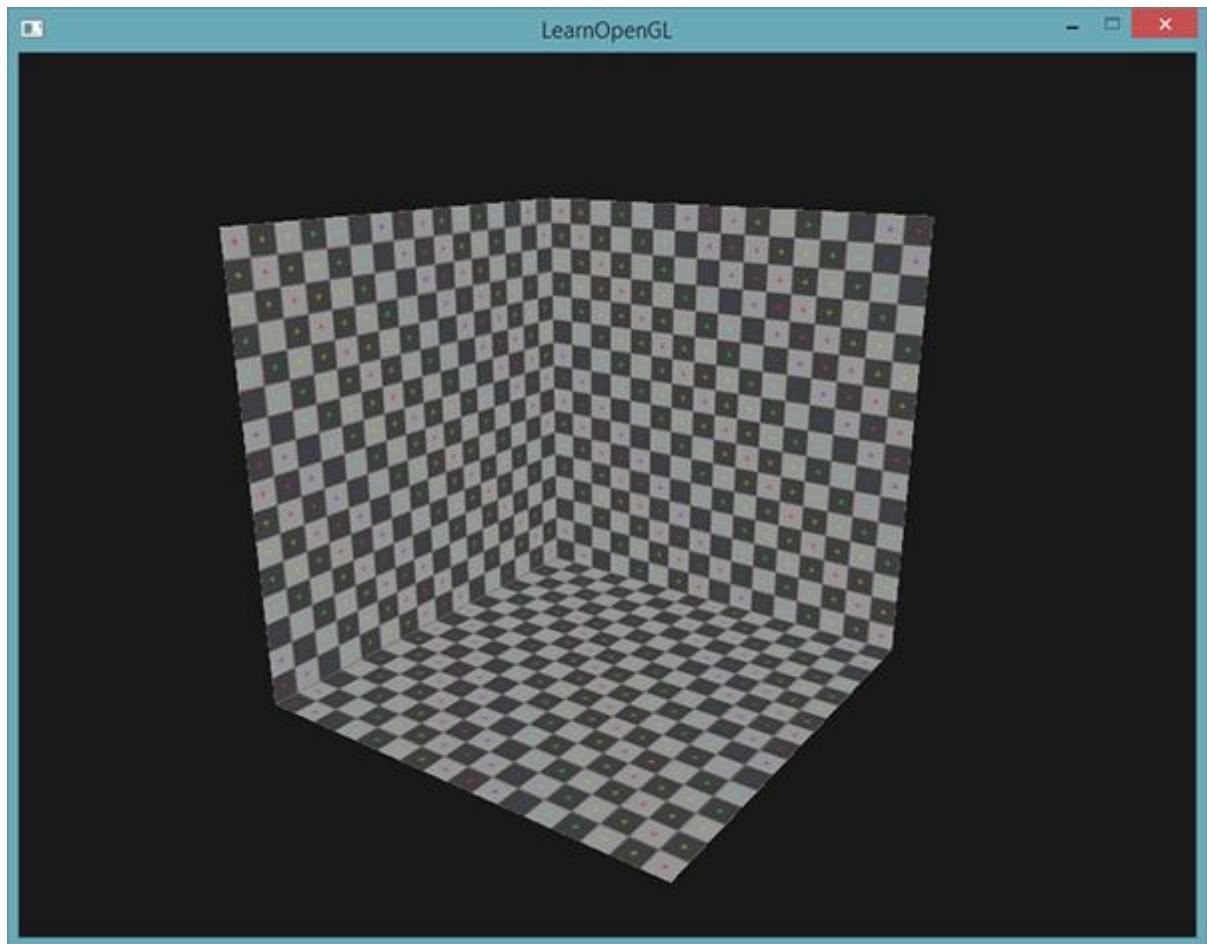
我们可以做个小实验，告诉 OpenGL 现在顺时针代表正面：

```
glEnable(GL_CULL_FACE);
```

```
glCullFace(GL_BACK);
```

```
glFrontFace(GL_CW);
```

最后的结果只有背面被渲染了：



要注意，你可以使用默认逆时针顺序剔除正面，来创建相同的效果：

```
glEnable(GL_CULL_FACE);
```

```
glCullFace(GL_FRONT);
```

正如你所看到的那样，面剔除是 OpenGL 提高效率的一个强大工具，它使应用节省运算。你必须跟踪下来哪个物体可以使用面剔除，哪些不能。

## 练习

你可以自己重新定义一个顺时针的顶点顺序，然后用顺时针作为正面把它渲染出来吗：[解决方案](#)。

## 帧缓冲（Framebuffer）

原文	<a href="#">Framebuffers</a>
作者	JoeyDeVries
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

到目前为止，我们使用了几种不同类型的屏幕缓冲：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，以及允许我们基于一些条件丢弃指定片段的模板缓冲。把这几种缓冲结合起来叫做帧缓冲(Framebuffer)，它被储存于内存中。  
OpenGL 给了我们自己定义帧缓冲的自由，我们可以选择性的定义自己的颜色缓冲、深度和模板缓冲。

我们目前所做的渲染操作都是是在默认的帧缓冲之上进行的。当你创建了你的窗口的时候默认帧缓冲就被创建和配置好了（GLFW 为我们做了这件事）。通过创建我们自己的帧缓冲我们能够获得一种额外的渲染方式。

你也许不能立刻理解应用程序的帧缓冲的含义，通过帧缓冲可以将你的场景渲染到一个不同的帧缓冲中，可以使我们能够在场景中创建镜子这样的效果，或者做出一些炫酷的特效。首先我们会讨论它们是如何工作的，然后我们将利用帧缓冲来实现一些炫酷的效果。

### 创建一个帧缓冲

就像 OpenGL 中其他对象一样，我们可以使用一个叫做 `glGenFramebuffers` 的函数来创建一个帧缓冲对象（简称 FBO）：

```
GLuint fbo;

glGenFramebuffers(1, &fbo);
```

这种对象的创建和使用的方式我们已经见过不少了，因此它们的使用方式也和之前我们见过的其他对象的使用方式相似。首先我们要创建一个帧缓冲对象，把它

绑定到当前帧缓冲，做一些操作，然后解绑帧缓冲。我们使用 `glBindFramebuffer` 来绑定帧缓冲：

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

绑定到 `GL_FRAMEBUFFER` 目标后，接下来所有的读、写帧缓冲的操作都会影响到当前绑定的帧缓冲。也可以把帧缓冲分开绑定到读或写目标上，分别使用 `GL_READ_FRAMEBUFFER` 或 `GL_DRAW_FRAMEBUFFER` 来做这件事。如果绑定到了 `GL_READ_FRAMEBUFFER`，就能执行所有读取操作，像 `glReadPixels` 这样的函数使用了；绑定到 `GL_DRAW_FRAMEBUFFER` 上，就允许进行渲染、清空和其他的写入操作。大多数时候你不必分开用，通常把两个都绑定到 `GL_FRAMEBUFFER` 上就行。

很遗憾，现在我们还不能使用自己的帧缓冲，因为还没做完呢。建构一个完整的帧缓冲必须满足以下条件：

- 我们必须往里面加入至少一个附件（颜色、深度、模板缓冲）。
- 其中至少有一个是颜色附件。
- 所有的附件都应该是已经完全做好的（已经存储在内存之中）。
- 每个缓冲都应该有同样数目的样本。

如果你不知道什么是样本也不用担心，我们会在后面的教程中讲到。

从上面的需求中你可以看到，我们需要为帧缓冲创建一些附件，还需要把这些附件附加到帧缓冲上。当我们做完所有上面提到的条件的时候我们就可以用 `glCheckFramebufferStatus` 带上 `GL_FRAMEBUFFER` 这个参数来检查是否真的成功做到了。然后检查当前绑定的帧缓冲，返回了这些规范中的哪个值。如果返回的是 `GL_FRAMEBUFFER_COMPLETE` 就对了：

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) ==
 GL_FRAMEBUFFER_COMPLETE)
 // Execute victory dance
```

后续所有渲染操作将渲染到当前绑定的帧缓冲的附加缓冲中，由于我们的帧缓冲不是默认的帧缓冲，渲染命令对窗口的视频输出不会产生任何影响。出于这个原因，它被称为离屏渲染（off-screen rendering），就是渲染到一个另外的缓冲中。为了让所有的渲染操作对主窗口产生影响我们必须通过绑定为 0 来使默认帧缓冲被激活：

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

当我们做完所有帧缓冲操作，不要忘记删除帧缓冲对象：

```
glDeleteFramebuffers(1, &fbo);
```

现在在执行完成检测前，我们需要把一个或更多的附件附加到帧缓冲上。一个附件就是一个内存地址，这个内存地址里面包含一个为帧缓冲准备的缓冲，它可能是个图像。当创建一个附件的时候我们有两种方式可以采用：纹理或渲染缓冲（renderbuffer）对象。

## 纹理附件

当把一个纹理附加到帧缓冲上的时候，所有渲染命令会写入到纹理上，就像它是一个普通的颜色/深度或者模板缓冲一样。使用纹理的好处是，所有渲染操作的结果都会被储存为一个纹理图像，这样我们就可以简单的在着色器中使用了。

创建一个帧缓冲的纹理和创建普通纹理差不多：

```
GLuint 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);
```

这里主要的区别是我们把纹理的维度设置为屏幕大小（尽管不是必须的），我们还传递 `NULL` 作为纹理的 `data` 参数。对于这个纹理，我们只分配内存，而不去填充它。纹理填充会在渲染到帧缓冲的时候去做。同样，要注意，我们不用关心环绕方式或者 Mipmap，因为在大多数时候都不会需要它们的。

如果你打算把整个屏幕渲染到一个或大或小的纹理上，你需要用新的纹理的尺寸作为参数再次调用 `glViewport`（要在渲染到你的帧缓冲之前做好），否则只有一小部分纹理或屏幕能够绘制到纹理上。

现在我们已经创建了一个纹理，最后一件要做的事情是把它附加到帧缓冲上：

```
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D, texture, 0);
```

`glFramebufferTexture2D` 函数需要传入下列参数：

- **target:** 我们所创建的帧缓冲类型的目标（绘制、读取或两者都有）。
- **attachment:** 我们所附加的附件的类型。现在我们附加的是一个颜色附件。需要注意，最后的那个 0 是暗示我们可以附加 1 个以上颜色的附件。我们会在后面的教程中谈到。
- **textarget:** 你希望附加的纹理类型。
- **texture:** 附加的实际纹理。
- **level:** Mipmap level。我们设置为 0。

除颜色附件以外，我们还可以附加一个深度和一个模板纹理到帧缓冲对象上。为了附加一个深度缓冲，我们可以知道那个 `GL_DEPTH_ATTACHMENT` 作为附件类型。记住，这时纹理格式和内部格式类型（internalformat）就成了 `GL_DEPTH_COMPONENT` 去反应深度缓冲的存储格式。附加一个模板缓冲，你要使用 `GL_STENCIL_ATTACHMENT` 作为第二个参数，把纹理格式指定为 `GL_STENCIL_INDEX`。

也可以同时附加一个深度缓冲和一个模板缓冲为一个单独的纹理。这样纹理的每 32 位数值就包含了 24 位的深度信息和 8 位的模板信息。为了把一个深度和模板缓冲附加到一个单独纹理上，我们使用 `GL_DEPTH_STENCIL_ATTACHMENT` 类型配置纹理格式以包含深度值和模板值的结合物。下面是一个附加了深度和模板缓冲为单一纹理的例子：

```
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（Renderbuffer object attachments）

在介绍了帧缓冲的可行附件类型——纹理后，OpenGL 引进了渲染缓冲对象（**Renderbuffer objects**），所以在过去那些美好时光里纹理是附件的唯一可用的类型。和纹理图像一样，渲染缓冲对象也是一个缓冲，它可以是一堆字节、整数、像素或者其他东西。渲染缓冲对象的一大优点是，它以 OpenGL 原生渲染格式储存它的数据，因此在离屏渲染到帧缓冲的时候，这些数据就相当于被优化过的了。

渲染缓冲对象将所有渲染数据直接储存到它们的缓冲里，而不会进行针对特定纹理格式的任何转换，这样它们就成了一种快速可写的存储介质了。然而，渲染缓冲对象通常是只写的，不能修改它们（就像获取纹理，不能写入纹理一样）。可以用 `glReadPixels` 函数去读取，函数返回一个当前绑定的帧缓冲的特定像素区域，而不是直接返回附件本身。

因为它们的数据已经是原生格式了，在写入或把它们的数据简单地到其他缓冲的时候非常快。当使用渲染缓冲对象时，像切换缓冲这种操作变得异常高速。我们在每个渲染迭代末尾使用的那个 `glfwSwapBuffers` 函数，同样以渲染缓冲对象实现：我们简单地写入到一个渲染缓冲图像，最后交换到另一个里。渲染缓冲对象对于这种操作来说很完美。

创建一个渲染缓冲对象和创建帧缓冲代码差不多：

```
GLuint rbo;

 glGenRenderbuffers(1, &rbo);
```

相似地，我们打算把渲染缓冲对象绑定，这样所有后续渲染缓冲操作都会影响到当前的渲染缓冲对象：

```
 glBindRenderbuffer(GL_RENDERBUFFER, rbo);
```

由于渲染缓冲对象通常是只写的，它们经常作为深度和模板附件来使用，由于大多数时候，我们不需要从深度和模板缓冲中读取数据，但仍关心深度和模板测试。我们就需要有深度和模板值提供给测试，但不需要对这些值进行采样（**sample**），

所以深度缓冲对象是完全符合的。当我们不去从这些缓冲中采样的时候，渲染缓冲对象通常很合适，因为它们等于是被优化过的。

调用 `glRenderbufferStorage` 函数可以创建一个深度和模板渲染缓冲对象：

```
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800,
600);
```

创建一个渲染缓冲对象与创建纹理对象相似，不同之处在于这个对象是专门被设计用于图像的，而不是通用目的的数据缓冲，比如纹理。这里我们选择 `GL_DEPTH24_STENCIL8` 作为内部格式，它同时代表 24 位的深度和 8 位的模板缓冲。

最后一件还要做的事情是把帧缓冲对象附加上：

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

在帧缓冲项目中，渲染缓冲对象可以提供一些优化，但更重要的是知道何时使用渲染缓冲对象，何时使用纹理。通常的规则是，如果你永远都不需要从特定的缓冲中进行采样，渲染缓冲对象对特定缓冲是更明智的选择。如果哪天需要从比如颜色或深度值这样的特定缓冲采样数据的话，你最好还是使用纹理附件。从执行效率角度考虑，它不会对效率有太大影响。

## 渲染到纹理

现在我们知道了（一些）帧缓冲如何工作的，是时候把它们用起来了。我们会把场景渲染到一个颜色纹理上，这个纹理附加到一个我们创建的帧缓冲上，然后把纹理绘制到一个简单的四边形上，这个四边形铺满整个屏幕。输出的图像看似和没用帧缓冲一样，但是这次，它其实是直接打印到了一个单独的四边形上面。为什么这很有用呢？下一部分我们会看到原因。

第一件要做的事情是创建一个帧缓冲对象，并绑定它，这比较明了：

```
GLuint framebuffer;
glGenFramebuffers(1, &framebuffer);
```

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

下一步我们创建一个纹理图像，这是我们将要附加到帧缓冲的颜色附件。我们把纹理的尺寸设置为窗口的宽度和高度，并保持数据未初始化：

```
// Generate texture
```

```
GLuint 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);
```

```
// Attach it to currently bound framebuffer object
```

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
```

```
GL_TEXTURE_2D, texColorBuffer, 0);
```

我们同样打算要让 OpenGL 确定可以进行深度测试（模板测试，如果你用的话）所以我们必须还要确保向帧缓冲中添加一个深度（和模板）附件。由于我们只采样颜色缓冲，并不采样其他缓冲，我们可以创建一个渲染缓冲对象来达到这个目的。记住，当你不打算从指定缓冲采样的时候，它们是一个不错的选择。

创建一个渲染缓冲对象不太难。唯一一件要记住的事情是，我们正在创建的是一个渲染缓冲对象的深度和模板附件。我们把它的内部给事设置为

[GL\\_DEPTH24\\_STENCIL8](#)，对于我们的目的来说这个精确度已经足够了。

```
GLuint rbo;
```

```
 glGenRenderbuffers(1, &rbo);
```

```
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800,
600);
glBindRenderbuffer(GL_RENDERBUFFER, 0);
```

我们为渲染缓冲对象分配了足够的内存空间以后，我们可以解绑渲染缓冲。

接着，在做好帧缓冲之前，还有最后一步，我们把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);
```

然后我们要检查帧缓冲是否真的做好了，如果没有，我们就打印一个错误消息。

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
GL_FRAMEBUFFER_COMPLETE)
cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

还要保证解绑帧缓冲，这样我们才不会意外渲染到错误的帧缓冲上。

现在帧缓冲做好了，我们要做的全部就是渲染到帧缓冲上，而不是绑定到帧缓冲对象的默认缓冲。余下所有命令会影响到当前绑定的帧缓冲上。所有深度和模板操作同样会从当前绑定的帧缓冲的深度和模板附件中读取，当然，得是在它们可用的情况下。如果你遗漏了比如深度缓冲，所有深度测试就不会工作，因为当前绑定的帧缓冲里没有深度缓冲。

所以，为把场景绘制到一个单独的纹理，我们必须以下面步骤来做：

1. 使用新的绑定为激活帧缓冲的帧缓冲，像往常那样渲染场景。
2. 绑定到默认帧缓冲。
3. 绘制一个四边形，让它平铺到整个屏幕上，用新的帧缓冲的颜色缓冲作为他的纹理。

我们使用在深度测试教程中同一个场景进行绘制，但是这次使用老气横秋的箱子纹理。

为了绘制四边形我们将会创建新的着色器。我们不打算引入任何花哨的变换矩阵，因为我们只提供已经是标准化设备坐标的顶点坐标，所以我们可以直接把它们作为顶点着色器的输出。顶点着色器看起来像这样：

```
#version 330 core

layout (location = 0) in vec2 position;

layout (location = 1) in vec2 texCoords;

out vec2 TexCoords;

void main()
{
 gl_Position = vec4(position.x, position.y, 0.0f, 1.0f);
 TexCoords = texCoords;
}
```

没有花哨的地方。片段着色器更简洁，因为我们做的唯一一件事是从纹理采样：

```
#version 330 core

in vec2 TexCoords;

out vec4 color;

uniform sampler2D screenTexture;

void main()
{
}
```

```
 color = texture(screenTexture, TexCoords);
```

```
}
```

接着需要你为屏幕上的四边形创建和配置一个 VAO。渲染迭代中帧缓冲处理会有下面的结构：

```
// First pass
```

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

```
glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // We're not
```

```
using stencil buffer now
```

```
 glEnable(GL_DEPTH_TEST);
```

```
DrawScene();
```

```
// Second pass
```

```
glBindFramebuffer(GL_FRAMEBUFFER, 0); // back to default
```

```
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
```

```
glClear(GL_COLOR_BUFFER_BIT);
```

```
screenShader.Use();
```

```
 glBindVertexArray(quadVAO);
```

```
 glDisable(GL_DEPTH_TEST);
```

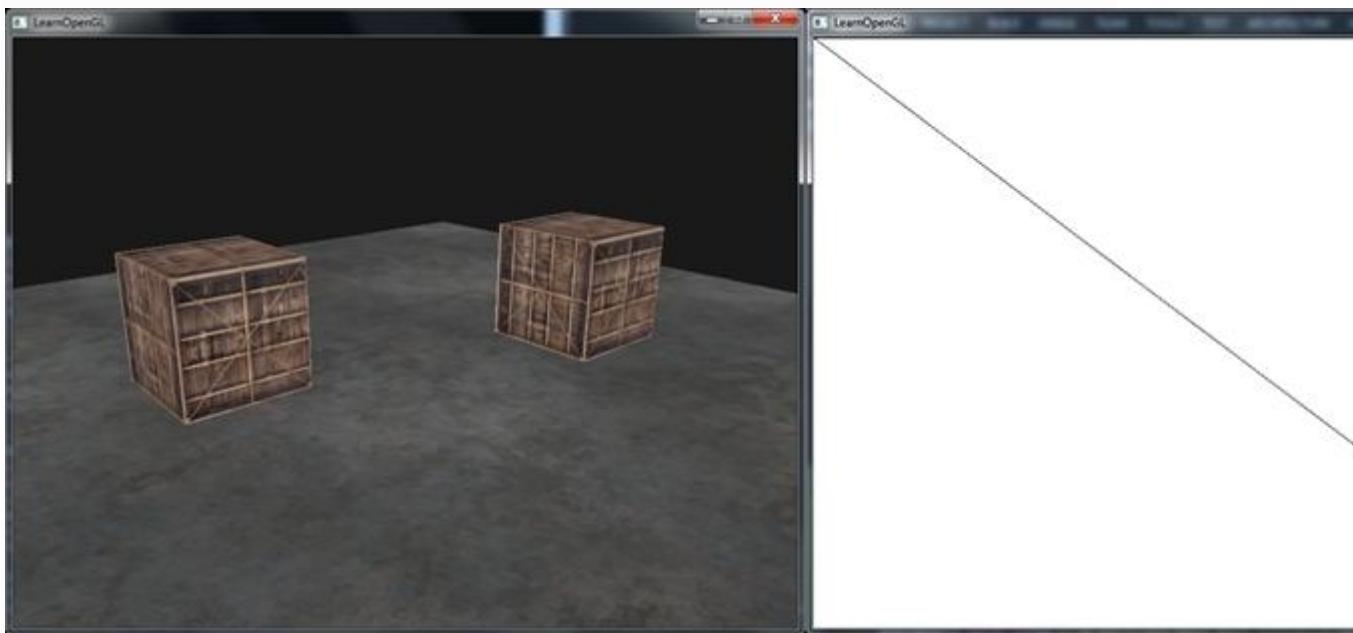
```
 glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
```

```
 glDrawArrays(GL_TRIANGLES, 0, 6);
```

```
 glBindVertexArray(0);
```

只有很少的事情要说明。第一，由于我们用的每个帧缓冲都有自己的一系列缓冲，我们打算使用 `glClear` 设置的合适的位（**bits**）来清空这些缓冲。第二，当渲染四边形的时候，我们关闭深度测试，因为我们不关系深度测试，我们绘制的是一个简单的四边形；当我们绘制普通场景时我们必须再次开启深度测试。

这里的确有很多地方会做错，所以如果你没有获得任何输出，尝试排查任何可能出现错误的地方，再次阅读教程中相关章节。如果每件事都做对了就一定能成功，你将会得到这样的输出：



左侧展示了和深度测试教程中一样的输出结果，但是这次却是渲染到一个简单的四边形上的。如果我们以线框方式显示的话，那么显然，我们只是绘制了一个默认帧缓冲中单调的四边形。

你可以[从这里得到应用的源码](#)。

然而这有什么好处呢？好处就是我们现在可以自由的获取已经渲染场景中的任何像素，然后把它当作一个纹理图像了，我们可以在片段着色器中创建一些有意思的效果。所有这些有意思的效果统称为后处理特效。

## 后处理

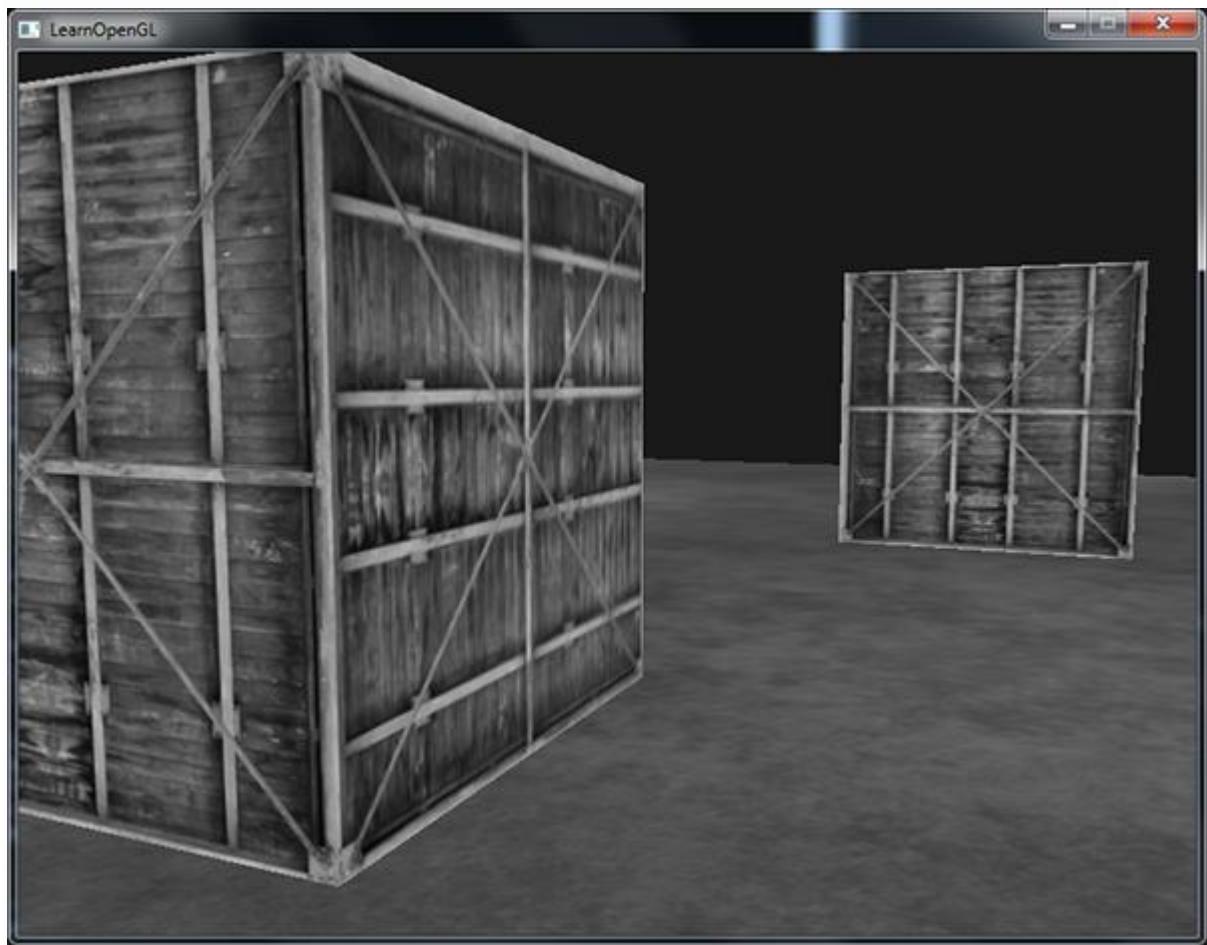
现在，整个场景渲染到了一个单独的纹理上，我们可以创建一些有趣的效果，只要简单操纵纹理数据就能做到。这部分，我们会向你展示一些流行的后处理特效，以及怎样添加一些创造性去创建出你自己的特效。

### 反相

我们已经取得了渲染输出的每个颜色，所以在片段着色器里返回这些颜色的反色并不难。我们得到屏幕纹理的颜色，然后用 1.0 减去它：

```
void main()
{
 color = vec4(vec3(1.0 - texture(screenTexture, TexCoords)), 1.0);
}
```

虽然反相是一种相对简单的后处理特效，但是已经很有趣了：



整个场景现在的颜色都反转了，只需在着色器中写一行代码就能做到，酷吧？

## 灰度

另一个有意思的效果是移除所有除了黑白灰以外的颜色作用，是整个图像成为黑白的。实现它的简单的方式是获得所有颜色元素，然后将它们平均化：

```
void main()
{
 color = texture(screenTexture, TexCoords);

 float average = (color.r + color.g + color.b) / 3.0;

 color = vec4(average, average, average, 1.0);

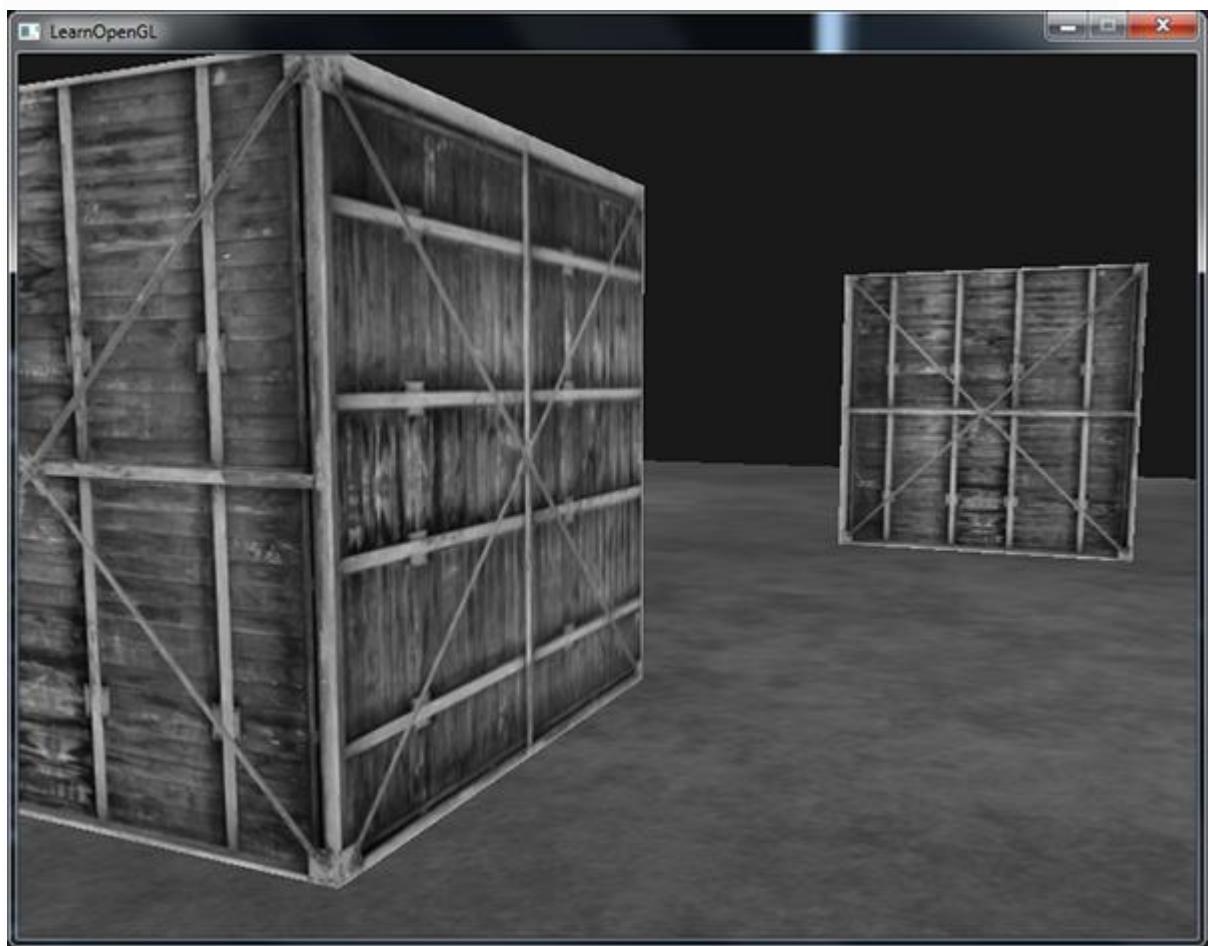
}
```

这已经创造出很赞的效果了，但是人眼趋向于对绿色更敏感，对蓝色感知比较弱，所以为了获得更精确的符合人体物理的结果，我们需要使用加权通道：

```
void main()
{
 color = texture(screenTexture, TexCoords);

 float average = 0.2126 * color.r + 0.7152 * color.g + 0.0722 *
color.b;

 color = vec4(average, average, average, 1.0);
}
```



## Kernel effects

在单独纹理图像上进行后处理的另一个好处是我们可以从纹理的其他部分进行采样。比如我们可以从当前纹理值的周围采样多个纹理值。创造性地把它们结合起来就能创造出有趣的效果了。

`kernel` 是一个长得有点像一个小矩阵的数值数组，它中间的值中心可以映射到一个像素上，这个像素和这个像素周围的值再乘以 `kernel`，最后再把结果相加就能得到一个值。所以，我们基本上就是给当前纹理坐标加上一个它四周的偏移量，然后基于 `kernel` 把它们结合起来。下面是一个 `kernel` 的例子：

这个 `kernel` 表示一个像素周围八个像素乘以 2，它自己乘以-15。这个例子基本上就是把周围像素乘上 2，中间像素去乘以一个比较大的负数来进行平衡。

### Important

你在网上能找到的 `kernel` 的例子大多数都是所有值加起来等于 1，如果加起来不等于 1 就意味着这个纹理值比原来更大或者更小了。

`kernel` 对于后处理来说非常管用，因为用起来简单。网上能找到很多实例，为了能用上 `kernel` 我们还得改改片段着色器。这里假设每个 `kernel` 都是 3x3（实际上大多数都是 3x3）：

```
const float offset = 1.0 / 300;
```

```
void main()
```

```
{
```

```
 vec2 offsets[9] = vec2[](
```

```
 vec2(-offset, offset), // top-left
```

```
 vec2(0.0f, offset), // top-center
```

```
 vec2(offset, offset), // top-right
```

```
 vec2(-offset, 0.0f), // center-left
 vec2(0.0f, 0.0f), // center-center
 vec2(offset, 0.0f), // center-right
 vec2(-offset, -offset), // bottom-left
 vec2(0.0f, -offset), // bottom-center
 vec2(offset, -offset) // bottom-right
);
```

```
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]));
}

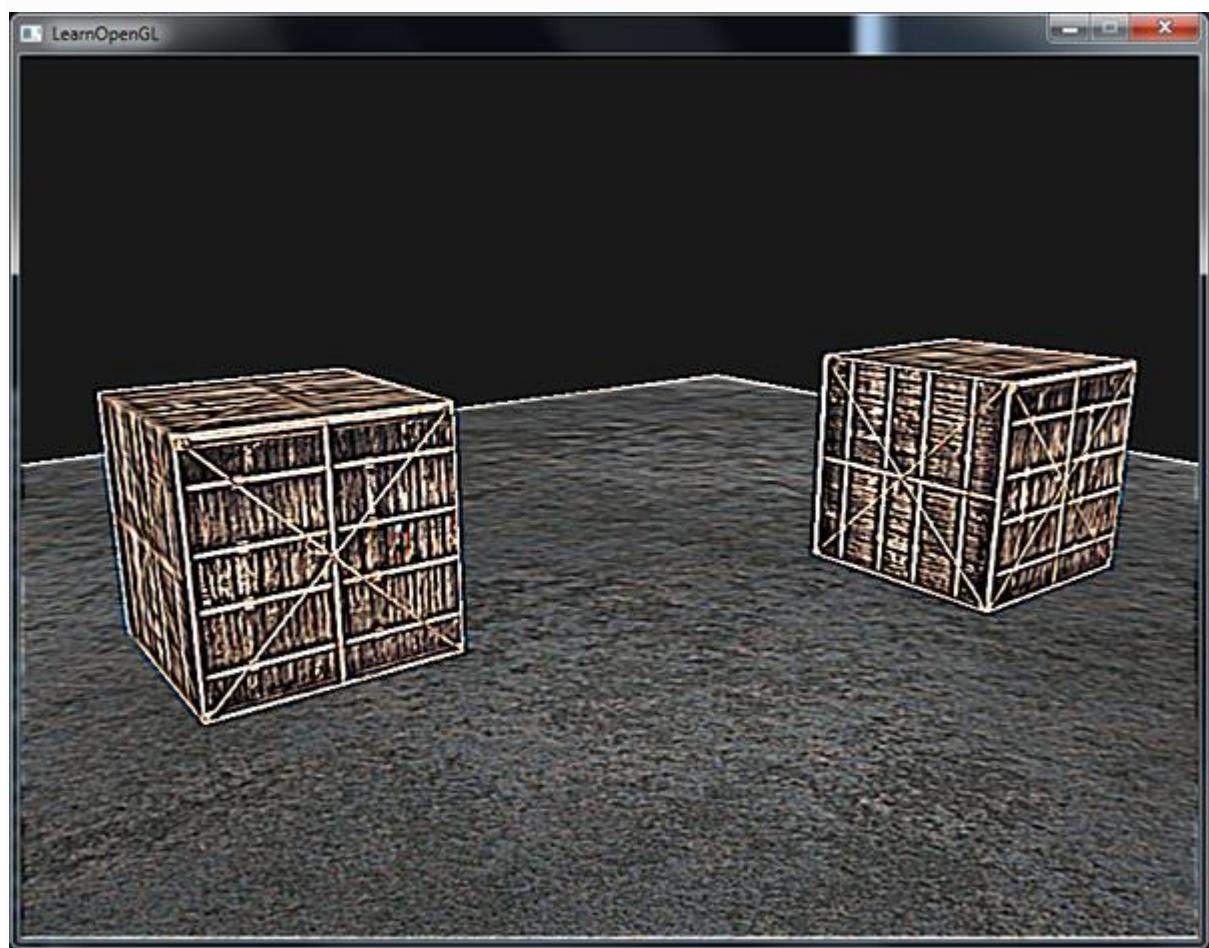
vec3 col;

for(int i = 0; i < 9; i++)
 col += sampleTex[i] * kernel[i];
```

```
color = vec4(col, 1.0);
}
```

在片段着色器中我们先为每个四周的纹理坐标创建一个 9 个 `vec2` 偏移量的数组。偏移量是一个简单的常数，你可以设置为自己喜欢的。接着我们定义 `kernel`，这里应该是一个锐化 `kernel`，它通过一种有趣的方式从所有周边的像素采样，对每个颜色值进行锐化。最后，在采样的时候我们把每个偏移量加到当前纹理坐标上，然后用加在一起的 `kernel` 的值乘以这些纹理值。

这个锐化的 `kernel` 看起来像这样：



这里创建的有趣的效果就好像你的玩家吞了某种麻醉剂产生的幻觉一样。

## Blur

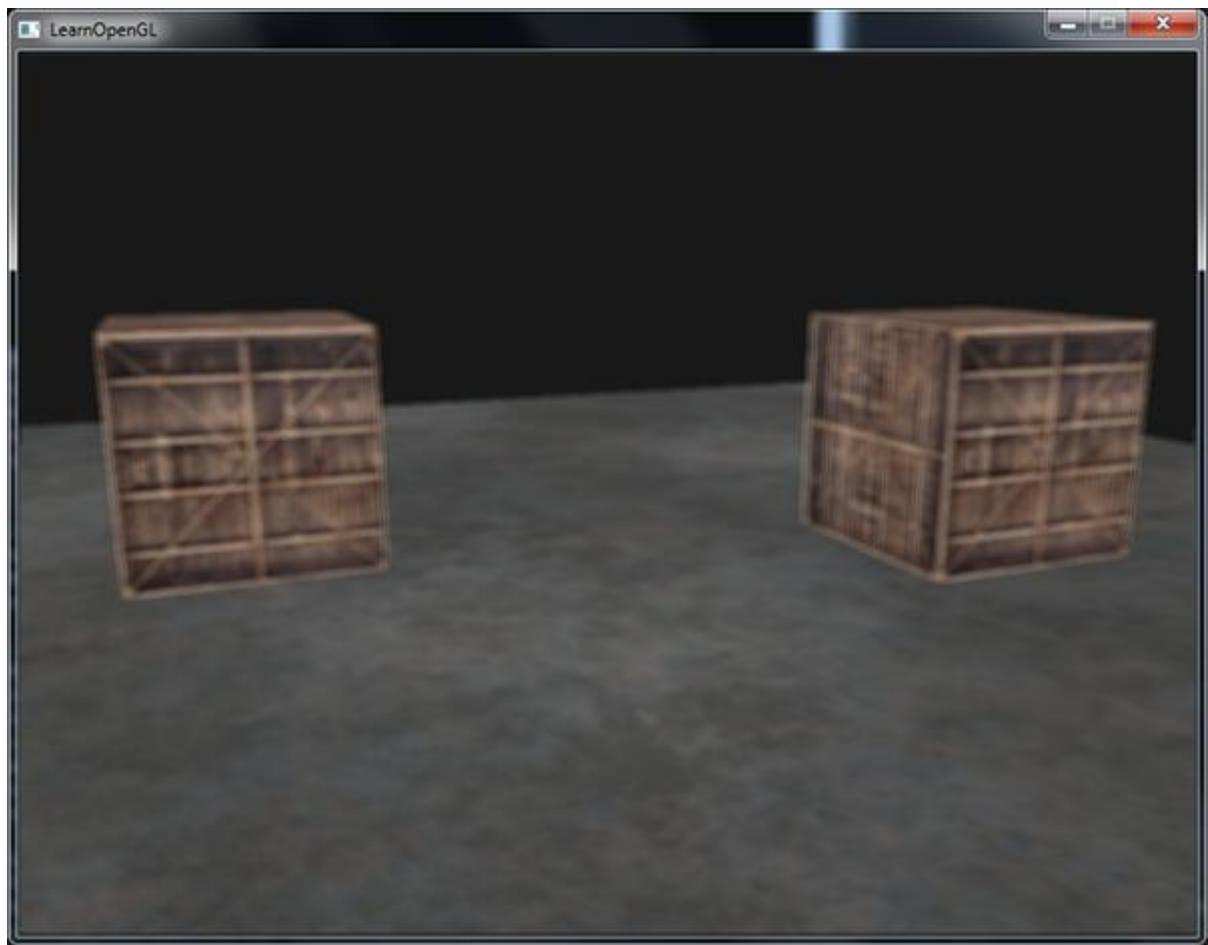
创建模糊效果的 `kernel` 定义如下：

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} / 16$$

由于所有数值加起来的总和为 16,简单返回结合起来的采样颜色是非常亮的,所以我们必须将 kernel 的每个值除以 16.最终的 kernel 数组会是这样的:

```
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
);
```

通过在像素着色器中改变 kernel 的 float 数组,我们就完全改变了之后的后处理效果.现在看起来会像是这样:



这样的模糊效果具有创建许多有趣效果的潜力.例如,我们可以随着时间的变化改变模糊量,创建出类似于某人喝醉酒的效果,或者,当我们的主角摘掉眼镜的时候增加模糊.模糊也能为我们在后面的教程中提供都颜色值进行平滑处理的能力.

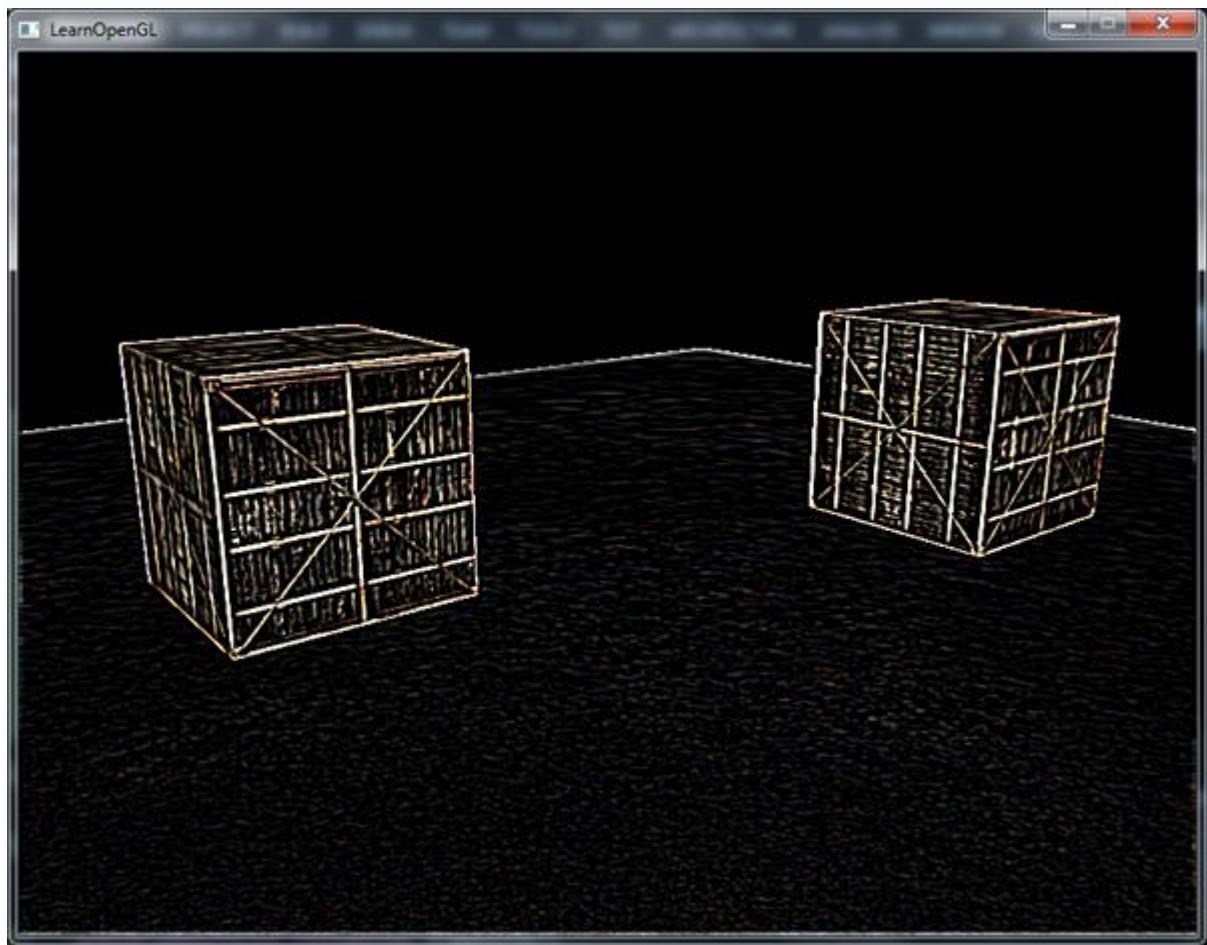
你可以看到我们一旦拥有了这个 `kernel` 的实现以后,创建一个后处理特效就不再是一件难事.最后,我们再来讨论一个流行的特效,以结束本节内容.

## 边检测

下面的边检测 `kernel` 与锐化 `kernel` 类似:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

这个 **kernel** 将所有的边提高亮度, 而对其他部分进行暗化处理, 当我们值关心一副图像的边缘的时候, 它非常有用.



在一些像 Photoshop 这样的软件中使用这些 kernel 作为图像操作工具/过滤器一点都不奇怪.因为掀开可以具有很强的平行处理能力，我们以实时进行针对每个像素的图像操作便相对容易，图像编辑工具因而更经常使用显卡来进行图像处理。

## 练习

- 你可以使用帧缓冲来创建一个后视镜吗?做到它,你必须绘制场景两次:一次正常绘制,另一次摄像机旋转 180 度后绘制.尝试在你的显示器顶端创建一个小四边形,在上面应用后视镜的镜面纹理:[解决方案](#),[视觉效果](#)
- 自己随意调整一下 kernel 值,创建出你自己后处理特效.尝试在网上搜索其他有趣的 kernel.

## 立方体贴图(Cubemap)

原文

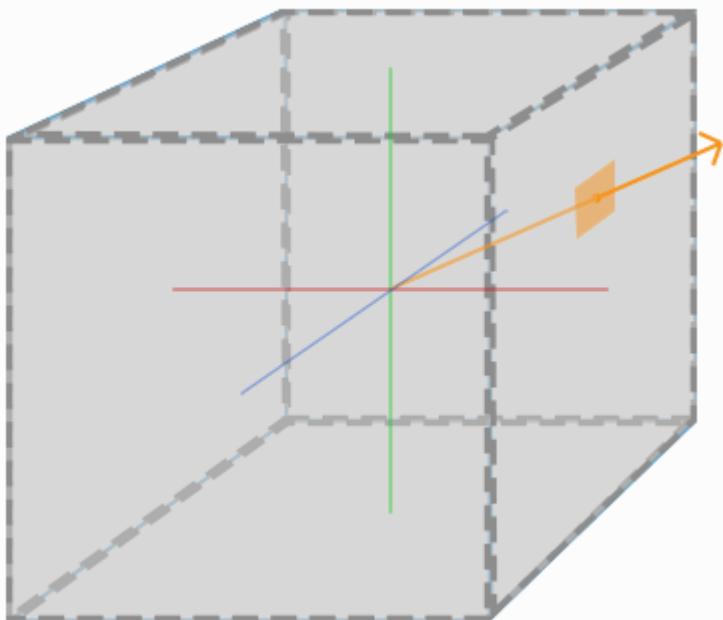
[Cubemaps](#)

原文	<a href="#">Cubemaps</a>
作者	<a href="#">JoeyDeVries</a>
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

我们之前一直使用的是 2D 纹理，还有更多的纹理类型我们没有探索过，本教程中我们讨论的纹理类型是将多个纹理组合起来映射到一个单一纹理，它就是 cubemap。

基本上说 cubemap 它包含 6 个 2D 纹理，这每个 2D 纹理是一个立方体（cube）的一个面，也就是说它是一个有贴图的立方体。你可能会奇怪这样的立方体有什么用？为什么费事地把 6 个独立纹理结合为一个单独的纹理，只使用 6 个各自独立的不行吗？这是因为 cubemap 有自己特有的属性，可以使用方向向量对它们索引和采样。想象一下，我们有一个  $1 \times 1 \times 1$  的单位立方体，有个以原点为起点的方向向量在它的中心。

从 cubemap 上使用橘黄色向量采样一个纹理值看起来和下图有点像：



**Important**

方向向量的大小无关紧要。一旦提供了方向，OpenGL 就会获取方向向量触碰到立方体表面上的相应的纹理像素（texel），这样就返回了正确的纹理采样值。

方向向量触碰到立方体表面的一点也就是 **cubemap** 的纹理位置，这意味着只要立方体的中心位于原点上，我们就可以使用立方体的位置向量来对 **cubemap** 进行采样。然后我们就可以获取所有顶点的纹理坐标，就和立方体上的顶点位置一样。所获得的结果是一个纹理坐标，通过这个纹理坐标就能获取到 **cubemap** 上正确的纹理。

## 创建一个 Cubemap

Cubemap 和其他纹理一样，所以要创建一个 **cubemap**，在进行任何纹理操作之前，需要生成一个纹理，激活相应纹理单元然后绑定到合适的纹理目标上。这次要绑定到 **GL\_TEXTURE\_CUBE\_MAP** 纹理类型：

```
GLuint textureID;

 glGenTextures(1, &textureID);

 glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

由于 **cubemap** 包含 6 个纹理，立方体的每个面一个纹理，我们必须调用 **glTexImage2D** 函数 6 次，函数的参数和前面教程讲的相似。然而这次我们必须把纹理目标（target）参数设置为 **cubemap** 特定的面，这是告诉 OpenGL 我们创建的纹理是对应立方体哪个面的。因此我们便需要为 **cubemap** 的每个面调用一次 **glTexImage2D**。

由于 **cubemap** 有 6 个面，OpenGL 就提供了 6 个不同的纹理目标，来应对 **cubemap** 的各个面。

纹理目标（Texture target）	方位
GL_TEXTURE_CUBE_MAP_POSITIVE_X	右
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	左
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	上
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	下

纹理目标 (Texture target)	方位
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	后
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	前

和很多 OpenGL 其他枚举一样，对应的 int 值都是连续增加的，所以我们有一个纹理位置的数组或 vector，就能以 `GL_TEXTURE_CUBE_MAP_POSITIVE_X` 为起始来对它们进行遍历，每次迭代枚举值加 1，这样循环所有的纹理目标效率较高：

```

int width,height;

unsigned char* image;

for(GLuint i = 0; i < textures_faces.size(); i++)
{
 image = SOIL_load_image(textures_faces[i], &width, &height, 0,
 SOIL_LOAD_RGB);

 glTexImage2D(
 GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image
);
}

```

这儿我们有个 vector 叫 `textures_faces`，它包含 cubemap 所各个纹理的文件路径，并且以上表所列的顺序排列。它将为每个当前绑定的 cubemap 的每个面生成一个纹理。

由于 cubemap 和其他纹理没什么不同，我们也要定义它的环绕方式和过滤方式：

```

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);
```

别被 `GL_TEXTURE_WRAP_R` 吓到，它只是简单的设置了纹理的 R 坐标，R 坐标对应于纹理的第三个维度（就像位置的 z 一样）。我们把放置方式设置为 `GL_CLAMP_TO_EDGE`，由于纹理坐标在两个面之间，所以可能并不能触及哪个面（由于硬件限制），因此使用 `GL_CLAMP_TO_EDGE` 后 OpenGL 会返回它们的边界的值，尽管我们可能在两个两个面中间进行的采样。

在绘制物体之前，将使用 `cubemap`，而在渲染前我们要激活相应的纹理单元并绑定到 `cubemap` 上，这和普通的 2D 纹理没什么区别。

在片段着色器中，我们也必须使用一个不同的采样器——**samplerCube**，用它来从 `texture` 函数中采样，但是这次使用的是一个 `vec3` 方向向量，取代 `vec2`。下面是一个片段着色器使用了 `cubemap` 的例子：

```
in vec3 textureDir; // 用一个三维方向向量来表示Cubemap 纹理的坐标

uniform samplerCube cubemap; // Cubemap 纹理采样器

void main()
{
 color = texture(cubemap, textureDir);
```

}

看起来不错，但是何必这么做呢？因为恰巧使用 **cubemap** 可以简单的实现很多有意思的技术。其中之一便是著名的**天空盒(Skybox)**。

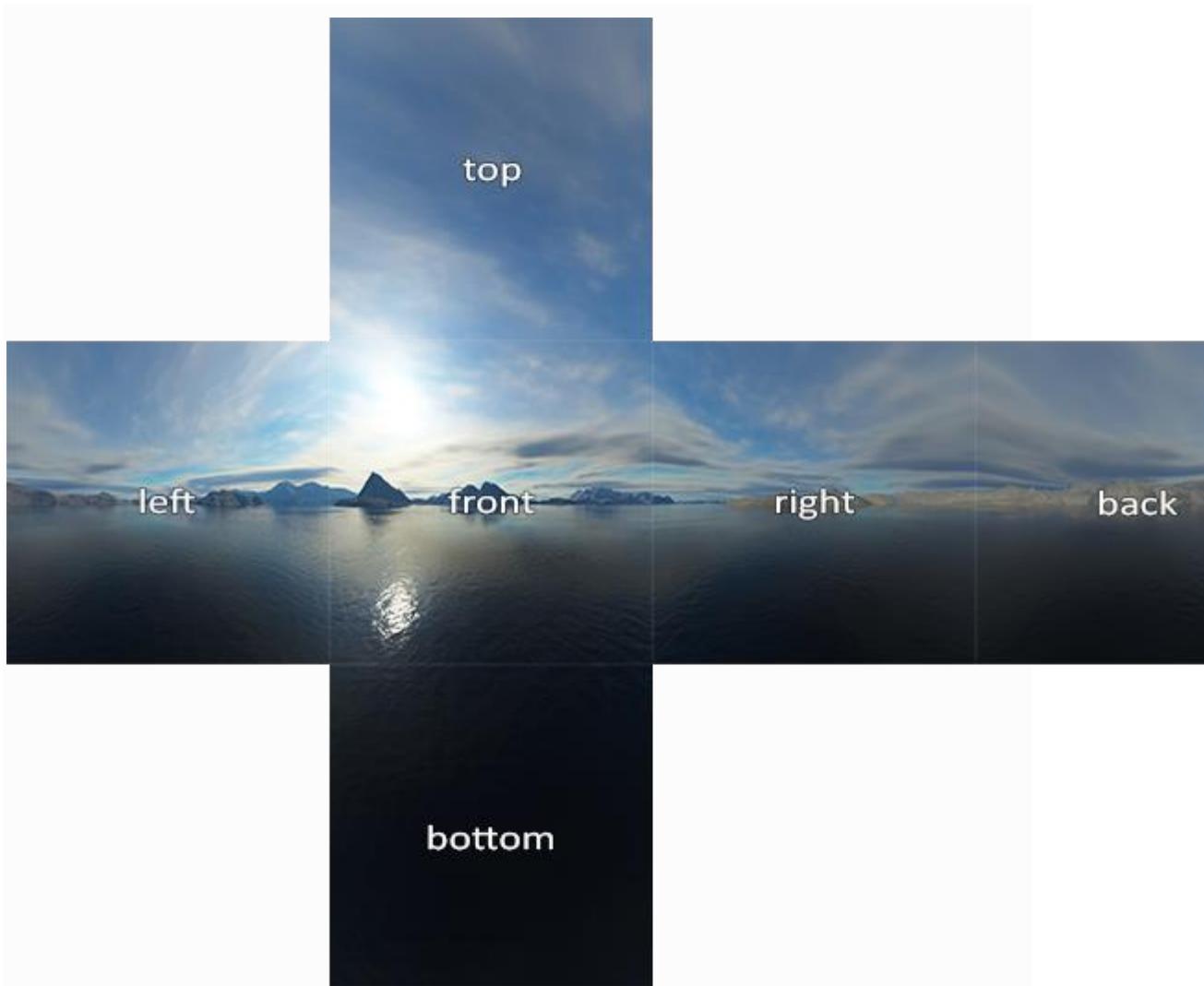
## 天空盒(Skybox)

天空盒是一个包裹整个场景的立方体，它由 6 个图像构成一个环绕的环境，给玩家一种他所在的场景比实际的要大得多的幻觉。比如有些在视频游戏中使用的天空盒的图像是群山、白云或者满天繁星。比如下面的夜空繁星的图像就来自《上古卷轴》：



你现在可能已经猜到 **cubemap** 完全满足天空盒的要求：我们有一个立方体，它有 6 个面，每个面需要一个贴图。上图中使用了几个夜空的图片给予玩家一种置身广袤宇宙的感觉，可实际上，他还是在一个小盒子之中。

网上有很多这样的天空盒的资源。[这个网站](#)就提供了很多。这些天空盒图像通常有下面的样式：



如果你把这 6 个面折叠到一个立方体中，你机会获得模拟了一个巨大的风景的立方体。有些资源所提供的天空盒比如这个例子 6 个图是连在一起的，你必须手工它们切割出来，不过大多数情况它们都是 6 个单独的纹理图像。

这个细致（高精度）的天空盒就是我们将在场景中使用的那个，你可以[在这里下载](#)。

## 加载一个天空盒

由于天空盒实际上就是一个 `cubemap`，加载天空盒和之前我们加载 `cubemap` 的没什么大的不同。为了加载天空盒我们将使用下面的函数，它接收一个包含 6 个纹理文件路径的 `vector`：

```
GLuint LoadCubemap(vector<const GLchar*> faces)
```

```
{
 GLuint textureID;

 glGenTextures(1, &textureID);

 glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

 for(GLuint i = 0; i < faces.size(); i++)
 {
 image = SOIL_load_image(faces[i], &width, &height, 0,
 SOIL_LOAD_RGB);

 glTexImage2D(
 GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
 GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image
);
 }

 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);

glBindTexture(GL_TEXTURE_CUBE_MAP, 0);

return textureID;
}
```

这个函数没什么特别之处。这就是我们前面已经见过的 cubemap 代码，只不过放进了一个可管理的函数中。

然后，在我们调用这个函数之前，我们将把合适的纹理路径加载到一个 vector 之中，顺序还是按照 cubemap 枚举的特定顺序：

```
vector<const GLchar*> faces;

faces.push_back("right.jpg");

faces.push_back("left.jpg");

faces.push_back("top.jpg");

faces.push_back("bottom.jpg");

faces.push_back("back.jpg");

faces.push_back("front.jpg");

GLuint cubemapTexture = loadCubemap(faces);
```

现在我们已经用 `cubemapTexture` 作为 id 把天空盒加载为 cubemap。我们现在可以把它绑定到一个立方体来替换不完美的 `clear color`，在前面的所有教程中这个东西做背景已经很久了。

## 天空盒的显示

因为天空盒绘制在了一个立方体上，我们还需要另一个 VAO、VBO 以及一组全新的顶点，和任何其他物体一样。你可以[从这里获得顶点数据](#)。

`cubemap` 用于给 3D 立方体帖上纹理，可以用立方体的位置作为纹理坐标进行采样。当一个立方体的中心位于原点(0, 0, 0)的时候，它的每一个位置向量也就是以原点为起点的方向向量。这个方向向量就是我们要得到的立方体某个位置的相应纹理值。出于这个理由，我们只需要提供位置向量，而无需纹理坐标。为了渲染天空盒，我们需要一组新着色器，它们不会太复杂。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 position;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

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

注意，顶点着色器有意思的地方在于我们把输入的位置向量作为输出给片段着色器的纹理坐标。片段着色器就会把它们作为输入去采样 `samplerCube`：

```
#version 330 core

in vec3 TexCoords;

out vec4 color;
```

```
uniform samplerCube skybox;
```

```
void main()
{
 color = texture(skybox, TexCoords);
}
```

片段着色器比较明了，我们把顶点属性中的位置向量作为纹理的方向向量，使用它们从 **cubemap** 采样纹理值。渲染天空盒现在很简单，我们有了一个 **cubemap** 纹理，我们简单绑定 **cubemap** 纹理，天空盒就自动地用天空盒的 **cubemap** 填充了。为了绘制天空盒，我们将把它作为场景中第一个绘制的物体并且关闭深度写入。这样天空盒才能成为所有其他物体的背景来绘制出来。

```
glDepthMask(GL_FALSE);

skyboxShader.Use();

// ... Set view and projection matrix

glBindVertexArray(skyboxVAO);

glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);

glDrawArrays(GL_TRIANGLES, 0, 36);

glBindVertexArray(0);

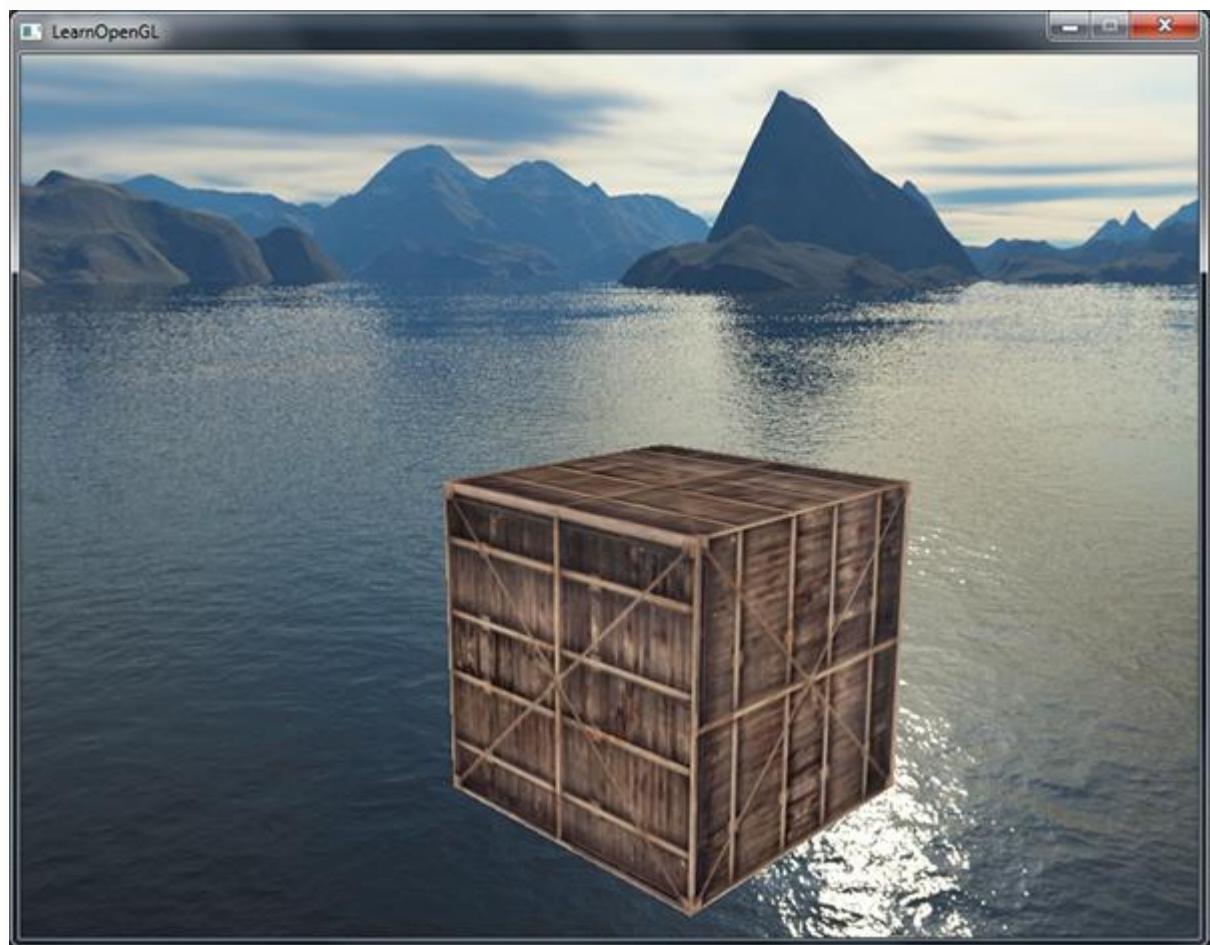
glDepthMask(GL_TRUE);

// ... Draw rest of the scene
```

如果你运行程序就会陷入困境，我们希望天空盒以玩家为中心，这样无论玩家移动了多远，天空盒都不会变近，这样就产生一种四周的环境真的非常大的印象。当前的视图矩阵对所有天空盒的位置进行了转转缩放和平移变换，所以玩家移动，`cubemap` 也会跟着移动！我们打算移除视图矩阵的平移部分，这样移动就影响不到天空盒的位置向量了。在基础光照教程里我们提到过我们可以只用  $4 \times 4$  矩阵的  $3 \times 3$  部分去除平移。我们可以简单地将矩阵转为  $3 \times 3$  矩阵再转回来，就能达到目标

```
glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

这会移除所有平移，但保留所有旋转，因此用户仍然能够向四面八方看。由于有了天空盒，场景即可变得巨大了。如果你添加些物体然后自由在其中游荡一会儿你会发现场景的真实度有了极大提升。最后的效果看起来像这样：



[这里有全部源码](#)，你可以对比一下你写的。

尝试用不同的天空盒实验，看看它们对场景有多大影响。

## 优化

现在我们在渲染场景中的其他物体之前渲染了天空盒。这么做没错，但是不怎么高效。如果我们先渲染了天空盒，那么我们就是在为每一个屏幕上的像素运行片段着色器，即使天空盒只有部分在显示着；`fragment` 可以使用前置深度测试（`early depth testing`）简单地被丢弃，这样就节省了我们宝贵的带宽。

所以最后渲染天空盒就能够给我们带来轻微的性能提升。采用这种方式，深度缓冲被全部物体的深度值完全填充，所以我们只需要渲染通过前置深度测试的那部分天空的片段就行了，而且能显著减少片段着色器的调用。问题是天空盒是个  $1 \times 1 \times 1$  的立方体，极有可能会渲染失败，因为极有可能通不过深度测试。简单地不用深度测试渲染它也不是解决方案，这是因为天空盒会在之后覆盖所有的场景中其他物体。我们需要耍个花招让深度缓冲相信天空盒的深度缓冲有着最大深度值  $1.0$ ，如此只要有个物体存在深度测试就会失败，看似物体就在它前面了。

在坐标系教程中我们说过，透视除法（`perspective division`）是在顶点着色器运行之后执行的，把 `gl_Position` 的  $xyz$  坐标除以  $w$  元素。我们从深度测试教程了解到除法结果的  $z$  元素等于顶点的深度值。利用这个信息，我们可以把输出位置的  $z$  元素设置为它的  $w$  元素，这样就会导致  $z$  元素等于  $1.0$  了，因为，当透视除法应用后，它的  $z$  元素转换为  $w/w = 1.0$ ：

```
void main()
{
 vec4 pos = projection * view * vec4(position, 1.0);
 gl_Position = pos.xyww;
 TexCoords = position;
}
```

最终，标准化设备坐标就总会有个与  $1.0$  相等的  $z$  值了， $1.0$  就是深度值的最大值。只有在没有任何物体可见的情况下天空盒才会被渲染（只有通过深度测试才渲染，否则假如有任何物体存在，就不会被渲染，只去渲染物体）。

我们必须改变一下深度方程，把它设置为 `GL_LESS`，原来默认的是 `GL_EQUAL`。深度缓冲会为天空盒用 **1.0** 这个值填充深度缓冲，所以我们需要保证天空盒是使用小于等于深度缓冲来通过深度测试的，而不是小于。

你可以在这里找到优化过的版本的[源码](#)。

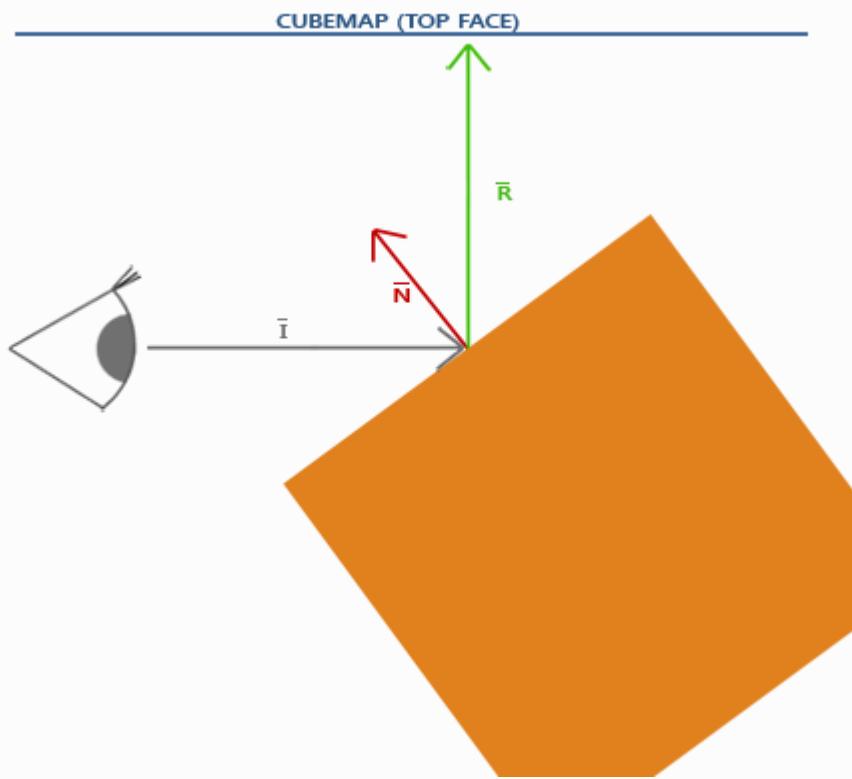
## 环境映射

我们现在有了一个把整个环境映射到为一个单独纹理的对象，我们利用这个信息能做的不仅是天空盒。使用带有场景环境的 `cubemap`，我们还可以让物体有一个反射或折射属性。像这样使用了环境 `cubemap` 的技术叫做**环境贴图技术**，其中最重要的两个是**反射(reflection)**和**折射(refraction)**。

### 反射(reflection)

凡是是一个物体（或物体的某部分）反射他周围的环境的属性，比如物体的颜色多少有些等于它周围的环境，这要基于观察者的角度。例如一个镜子是一个反射物体：它会基于观察者的角度泛着它周围的环境。

反射的基本思路不难。下图展示了我们如何计算反射向量，然后使用这个向量去从一个 `cubemap` 中采样：



我们基于观察方向向量  $\bar{I}$  和物体的法线向量  $\bar{N}$  计算出反射向量  $\bar{R}$ 。我们可以使用 GLSL 的内建函数 `reflect` 来计算这个反射向量。最后向量  $\bar{R}$  作为一个方向向量对 `cubemap` 进行索引/采样，返回一个环境的颜色值。最后的效果看起来就像物体反射了天空盒。

因为我们在场景中已经设置了一个天空盒，创建反射就不难了。我们改变一下箱子使用的那个片段着色器，给箱子一个反射属性：

```
#version 330 core

in vec3 Normal;
in vec3 Position;
out vec4 color;

uniform vec3 cameraPos;
uniform samplerCube skybox;
```

```
void main()
{
 vec3 I = normalize(Position - cameraPos);

 vec3 R = reflect(I, normalize(Normal));

 color = texture(skybox, R);
}
```

我们先来计算观察/摄像机方向向量 **I**，然后使用它来计算反射向量 **R**，接着我们用 **R** 从天空盒 cubemap 采样。要注意的是，我们有了片段的插值 **Normal** 和 **Position** 变量，所以我们需要修正顶点着色器适应它。

```
#version 330 core

layout (location = 0) in vec3 position;

layout (location = 1) in vec3 normal;

out vec3 Normal;
out vec3 Position;

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

void main()
{
 gl_Position = projection * view * model * vec4(position, 1.0f);

 Normal = mat3(transpose(inverse(model))) * normal;
}
```

```
Position = vec3(model * vec4(position, 1.0f));
```

```
}
```

我们用了法线向量，所以我们打算使用一个**法线矩阵(normal matrix)**变换它们。  
`Position`输出的向量是一个世界空间位置向量。顶点着色器输出的 `Position` 用来在片段着色器计算观察方向向量。

因为我们使用法线，你还得更新顶点数据，更新属性指针。还要确保设置 `cameraPos` 的 uniform。

然后在渲染箱子前我们还得绑定 cubemap 纹理：

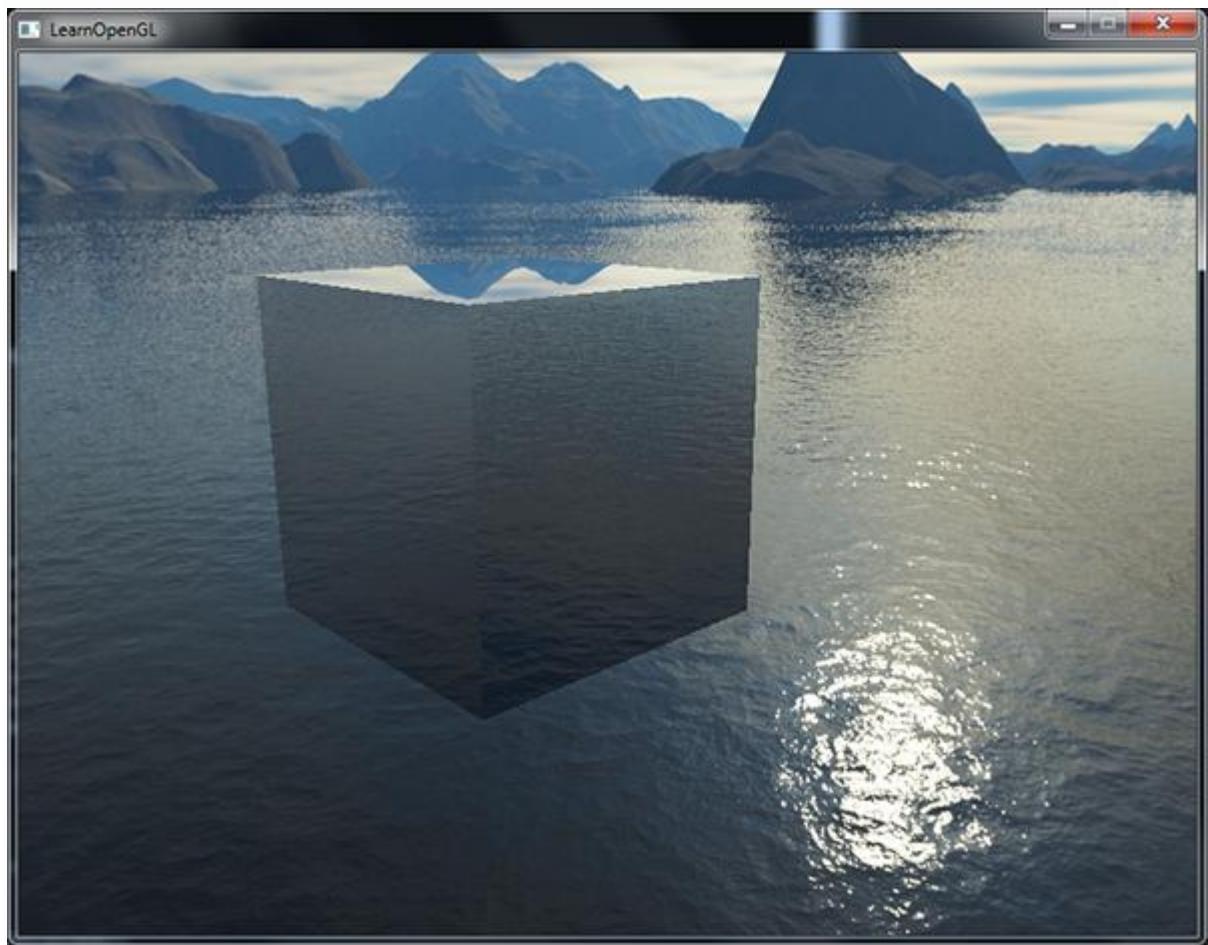
```
glBindVertexArray(cubeVAO);
```

```
glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);
```

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

```
glBindVertexArray(0);
```

编译运行你的代码，你等得到一个镜子一样的箱子。箱子完美地反射了周围的天空盒：



你可以从[这里](#)找到全部源代码。

当反射应用于整个物体之上的时候，物体看上去就像有一个像钢和铬这种高反射材质。如果我们加载[模型教程](#)中的纳米铠甲模型，我们就会获得一个铬金属制铠甲：

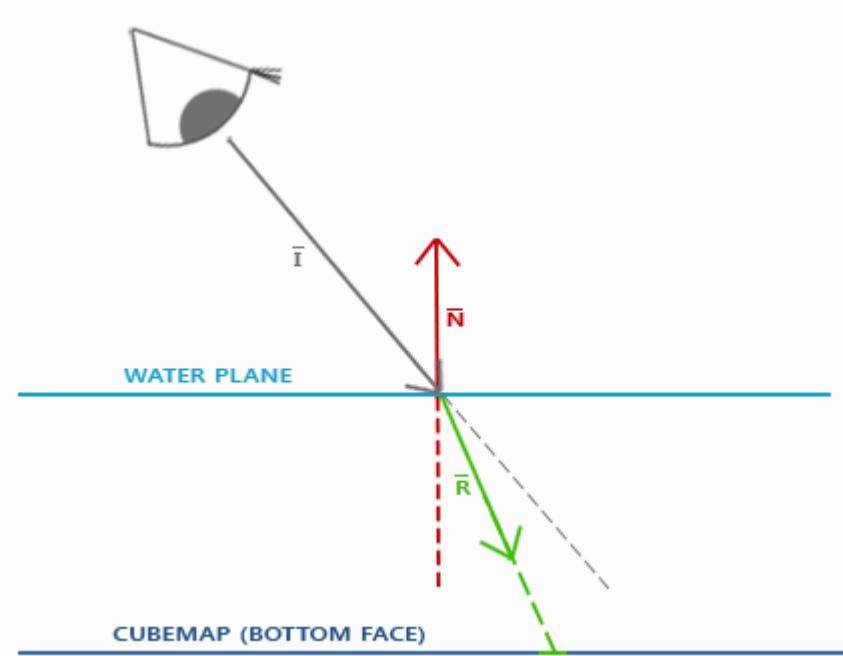


看起来挺惊艳，但是现实中大多数模型都不是完全反射的。我们可以引进反射贴图（**reflection map**）来使模型有另一层细节。和 **diffuse**、**specular** 贴图一样，我们可以从反射贴图上采样来决定 **fragment** 的反射率。使用反射贴图我们还可以决定模型的哪个部分有反射能力，以及强度是多少。本节的练习中，要由你来在我们早期创建的模型加载器引入反射贴图，这回极大的提升纳米服模型的细节。

### 折射(refraction)

环境映射的另一个形式叫做折射，它和反射差不多。折射是光线通过特定材质对光线方向的改变。我们通常看到像水一样的表面，光线并不是直接通过的，而是让光线弯曲了一点。它看起来像你把半只手伸进水里的效果。

折射遵守斯涅尔定律，使用环境贴图看起来就像这样：



我们有个观察向量  $\bar{I}$ ，一个法线向量  $\bar{N}$ ，这次折射向量是  $\bar{R}$ 。就像你所看到的那样，观察向量的方向有轻微弯曲。弯曲的向量  $\bar{R}$  随后用来从 cubemap 上采样。

折射可以通过 GLSL 的内建函数 `refract` 来实现，除此之外还需要一个法线向量，一个观察方向和一个两种材质之间的折射指数。

折射指数决定了一个材质上光线扭曲的数量，每个材质都有自己的折射指数。下表是常见的折射指数：

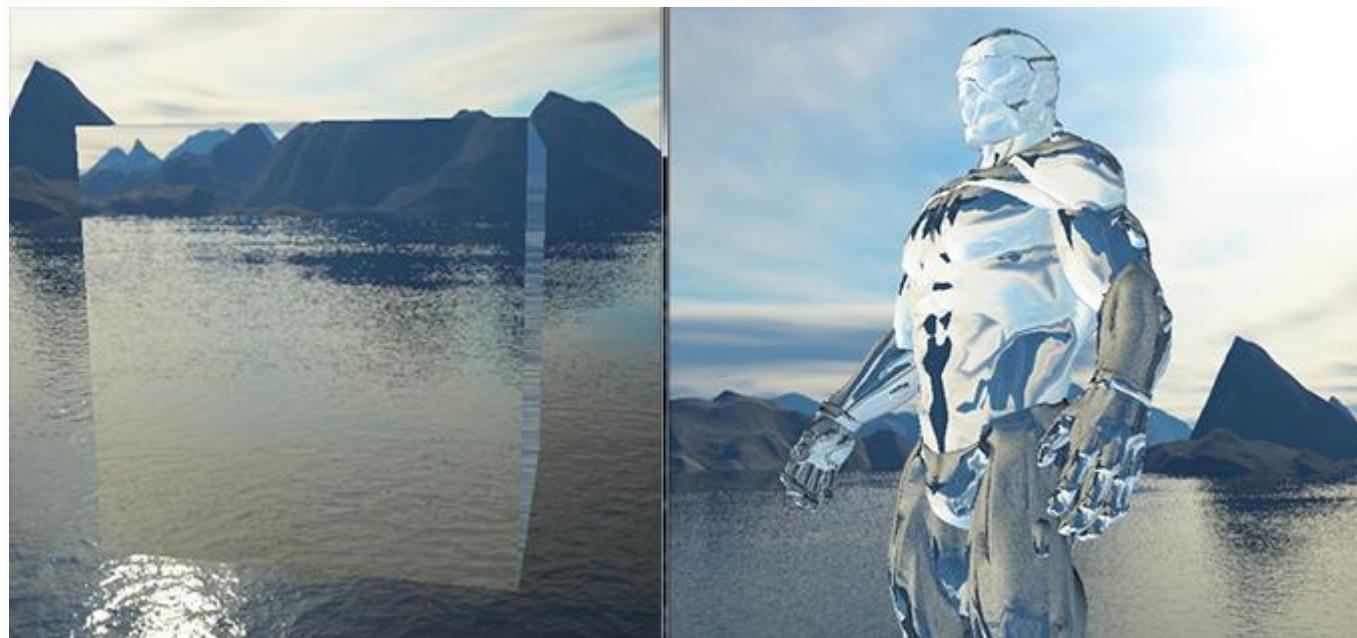
材质	折射指数
空气	1.00
水	1.33
冰	1.309
玻璃	1.52
宝石	2.42

我们使用这些折射指数来计算光线通过两个材质的比率。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做的）所以比率是  $1.00 / 1.52 = 0.658$ 。

我们已经绑定了 cubemap，提供了定点数据，设置了摄像机位置的 uniform。现在只需要改变片段着色器：

```
void main()
{
 float ratio = 1.00 / 1.52;
 vec3 I = normalize(Position - cameraPos);
 vec3 R = refract(I, normalize(Normal), ratio);
 color = texture(skybox, R);
}
```

通过改变折射指数你可以创建出完全不同的视觉效果。编译运行应用，结果也不是太有趣，因为我们只是用了一个普通箱子，这不能显示出折射的效果，看起来像个放大镜。使用同一个着色器，纳米服模型却可以展示出我们期待的效果：玻璃制物体。



你可以向想象一下，如果将光线、反射、折射和顶点的移动合理的结合起来就能创造出漂亮的水的图像。一定要注意，出于物理精确的考虑当光线离开物体的时

候还要再次进行折射；现在我们简单的使用了单边（一次）折射，大多数目的都可以得到满足。

## 动态环境贴图（Dynamic environment maps）

现在，我们已经使用了静态图像组合的天空盒，看起来不错，但是没有考虑到物体可能移动的实际场景。我们到现在还没注意到这点，是因为我们目前还只使用了一个物体。如果我们有个镜子一样的物体，它周围有多个物体，只有天空盒在镜子中可见，和场景中只有这一个物体一样。

使用帧缓冲可以为提到的物体的所有 6 个不同角度创建一个场景的纹理，把它们每次渲染迭代储存为一个 **cubemap**。之后我们可以使用这个（动态生成的）**cubemap** 来创建真实的反射和折射表面，这样就能包含所有其他物体了。这种方法叫做动态环境映射（dynamic environment mapping），因为我们动态地创建了一个物体的以其四周为参考的 **cubemap**，并把它用作环境贴图。

它看起效果很好，但是有一个劣势：使用环境贴图我们必须为每个物体渲染场景 6 次，这需要非常大的开销。现代应用尝试尽量使用天空盒子，凡可能预编译 **cubemap** 就创建少量动态环境贴图。动态环境映射是个非常棒的技术，要想在不降低执行效率的情况下实现它就需要很多巧妙的技巧。

## 练习

尝试在模型加载中引进反射贴图，你将再次得到很大视觉效果的提升。这其中有一点需要注意：

- Assimp 并不支持反射贴图，我们可以使用环境贴图的方式将反射贴图从 **aiTextureType\_AMBIENT** 类型中来加载反射贴图的材质。
- 我匆忙地使用反射贴图来作为镜面反射的贴图，而反射贴图并没有很好的映射在模型上:)。
- 由于加载模型已经占用了 3 个纹理单元，因此你要绑定天空盒到第 4 个纹理单元上，这样才能在同一个着色器内从天空盒纹理中取样。

You can find the solution source code here together with the updated model and mesh class. The shaders used for rendering the reflection maps can be found here: vertex shader and fragment shader.

你可以在此获取解决方案的[源代码](#)，这其中还包括升级过的[Model](#) 和 [Mesh](#) 类，还有用来绘制反射贴图的[顶点着色器](#)和[片段着色器](#)。

如果你一切都做对了，那你应该看到和下图类似的效果：



## 高级数据

原文	<a href="#">Advanced Data</a>
作者	JoeyDeVries
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

## 缓冲数据写入

我们在 OpenGL 中大量使用缓冲来储存数据已经有一会儿了。有一些有趣的方式来操纵缓冲，也有一些有趣的方式通过纹理来向着色器传递大量数据。本教程中，我们会讨论一些更加有意思的缓冲函数，以及如何使用纹理对象来储存大量数据（教程中纹理部分还没写）。

OpenGL 中缓冲只是一块儿内存区域的对象，除此没有更多点的了。当把缓冲绑定到一个特定缓冲对象的时候，我们就给缓冲赋予了一个特殊的意义。当我们绑定到 `GL_ARRAY_BUFFER` 的时候，这个缓冲就是一个顶点数组缓冲，我们也可以简单地绑定到 `GL_ELEMENT_ARRAY_BUFFER`。OpenGL 内部为每个目标（target）储存一个缓冲，并基于目标来处理不同的缓冲。

到目前为止，我们使用 `glBufferData` 函数填充缓冲对象管理的内存，这个函数分配了一块内存空间，然后把数据存入其中。如果我们向它的 `data` 这个参数传递的是 `NULL`，那么 OpenGL 只会帮我们分配内存，而不会填充它。如果我们先打算开辟一些内存，稍后回到这个缓冲一点一点的填充数据，有些时候会很有用。

我们还可以调用 `glBufferSubData` 函数填充特定区域的缓冲，而不是一次填充整个缓冲。这个函数需要一个缓冲目标（target），一个偏移量（offset），数据的大小以及数据本身作为参数。这个函数新的功能是我们可以给它一个偏移量（offset）来指定我们打算填充缓冲的位置与起始位置之间的偏移量。这样我们就可以插入/更新指定区域的缓冲内存空间了。一定要确保修改的缓冲要有足够的内存分配，所以在调用 `glBufferSubData` 之前，调用 `glBufferData` 是必须的。

```
glBufferSubData(GL_ARRAY_BUFFER, 24, sizeof(data), &data); // 范围:
[24, 24 + sizeof(data)]
```

把数据传进缓冲另一个方式是向缓冲内存请求一个指针，你自己直接把数据复制到缓冲中。调用 `glMapBuffer` 函数 OpenGL 会返回一个当前绑定缓冲的内存的地址，供我们操作：

```
float data[] = {
 0.5f, 1.0f, -0.35f
 ...
};
```

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// 获取当前绑定缓存 buffer 的内存地址
void* ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
```

```
// 向缓冲中写入数据

memcpy(ptr, data, sizeof(data));

// 完成够别忘了告诉OpenGL 我们不再需要它了

glUnmapBuffer(GL_ARRAY_BUFFER);
```

调用 `glUnmapBuffer` 函数可以告诉 OpenGL 我们已经用完指针了，OpenGL 会知道你已经做完了。通过解映射（unmapping），指针会不再可用，如果 OpenGL 可以把你的数据映射到缓冲上，就会返回 `GL_TRUE`。

把数据直接映射到缓冲上使用 `glMapBuffer` 很有用，因为不用把它储存在临时内存里。你可以从文件读取数据然后直接复制到缓冲的内存里。

## 分批处理顶点属性

使用 `glVertexAttribPointer` 函数可以指定缓冲内容的顶点数组的属性的布局（layout）。我们已经知道，通过使用顶点属性指针我们可以交叉属性，也就是说我们可以把每个顶点的位置、法线、纹理坐标放在彼此挨着的地方。现在我们了解了更多的缓冲的内容，可以采取另一种方式了。

我们可以做的是把每种类型的属性的所有向量数据批量保存在一个布局，而不是交叉布局。与交叉布局 123123123123 不同，我们采取批量方式 111122223333。

当从文件加载顶点数据时你通常获取一个位置数组，一个法线数组和一个纹理坐标数组。需要花点力气才能把它们结合为交叉数据。使用 `glBufferSubData` 可以简单的实现分批处理方式：

```
GLfloat positions[] = { ... };

GLfloat normals[] = { ... };

GLfloat 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);
```

这样我们可以把属性数组当作一个整体直接传输给缓冲，不需要再处理它们了。我们还可以把它们结合为一个更大的数组然后使用 `glBufferData` 立即直接填充它，不过对于这项任务使用 `glBufferSubData` 是更好的选择。

我们还要更新顶点属性指针来反应这些改变：

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
(GLvoid*)(sizeof(positions)));

glVertexAttribPointer(
2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat),
(GLvoid*)(sizeof(positions) + sizeof(normals)));
```

注意，`stride` 参数等于顶点属性的大小，由于同类型的属性是连续储存的，所以下一个顶点属性向量可以在它的后面 3（或 2）的元素那儿找到。

这是我们有了另一种设置和指定顶点属性的方式。使用哪个方式对 OpenGL 来说也不会有立竿见影的效果，这只不过是一种采用更加组织化的方式去设置顶点属性。选用哪种方式取决于你的偏好和应用类型。

## ## 复制缓冲

当你的缓冲被数据填充以后，你可能打算让其他缓冲能分享这些数据或者打算把缓冲的内容复制到另一个缓冲里。`glCopyBufferSubData` 函数让我们能够相对容易地把一个缓冲的数据复制到另一个缓冲里。函数的原型是：

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget,
GLintptr readoffset, GLintptr writeoffset, GLsizeiptr size);
```

`readtarget` 和 `writetarget` 参数是复制的来源和目的的缓冲目标。例如我们可以从一个 `VERTEX_ARRAY_BUFFER` 复制到一个 `VERTEX_ELEMENT_ARRAY_BUFFER`，各自指定源和目的的缓冲目标。当前绑定到这些缓冲目标上的缓冲会被影响到。

但如果我们打算读写的两个缓冲都是顶点数组缓冲(`GL_VERTEX_ARRAY_BUFFER`)怎么办？我们不能用通一个缓冲作为操作的读取和写入目标次。出于这个理由，OpenGL 给了我们另外两个缓冲目标叫做：`GL_COPY_READ_BUFFER` 和 `GL_COPY_WRITE_BUFFER`。这样我们就可以把我们选择的缓冲，用上面二者作为 `readtarget` 和 `writetarget` 的参数绑定到新的缓冲目标上了。

接着 `glCopyBufferSubData` 函数会从 `readoffset` 处读取的 `size` 大小的数据，写入到 `writetarget` 缓冲的 `writeoffset` 位置。下面是一个复制两个顶点数组缓冲的例子：

```
GLfloat 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));
```

我们也可以把 `writetarget` 缓冲绑定为新缓冲目标类型其中之一：

```
GLfloat vertexData[] = { ... };

glBindBuffer(GL_ARRAY_BUFFER, vbo1);

glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);

glCopyBufferSubData(GL_ARRAY_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0,
sizeof(vertexData));
```

有了这些额外的关于如何操纵缓冲的知识，我们已经可以以更有趣的方式来使用它们了。当你对 OpenGL 更熟悉，这些新缓冲方法就变得更有用。下个教程中我们会讨论 `uniform` 缓冲对象，彼时我们会充分利用 `glBufferSubData`。

## 高级 GLSL

原文	<a href="#">Advanced GLSL</a>
作者	JoeyDeVries
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

这章不会向你展示什么新的功能，也不会对你的场景的视觉效果有较大提升。本文多多少少地深入探讨了一些 GLSL 有趣的知识，它们可能在将来能帮助你。基本来说有些不可不知的内容和功能在你去使用 GLSL 创建 OpenGL 应用的时候能让你的生活更轻松。

我们会讨论一些内建变量、组织着色器输入和输出的新方式以及一个叫做 `uniform` 缓冲对象的非常有用的工具。

### GLSL 的内建变量

着色器是很小的，如果我们需要从当前着色器以外的别的资源里的数据，那么我们就不得不传给它。我们学过了使用顶点属性、`uniform` 和采样器可以实现这个目标。GLSL 有几个以 `gl_` 为前缀的变量，使我们有一个额外的手段来获取和写入数据。其中两个我们已经打过交道了：`gl_Position` 和 `gl_FragCoord`，前一个是顶点着色器的输出向量，后一个是片段着色器的变量。

我们会讨论几个有趣的 GLSL 内建变量，并向你解释为什么它们对我们来说很有好处。注意，我们不会讨论到 GLSL 中所有的内建变量，因此如果你想看到所有的内建变量还是最好去查看[OpenGL 的 [wiki\]](http://www.opengl.org/wiki/Built-in_Variable_(GLSL))([http://www.opengl.org/wiki/Built-in\\_Variable\\_\(GLSL\)](http://www.opengl.org/wiki/Built-in_Variable_(GLSL)))。

## 顶点着色器变量

### gl\_Position

我们已经了解 `gl_Position` 是顶点着色器裁切空间输出的位置向量。如果你想让屏幕上渲染出东西 `gl_Position` 必须使用。否则我们什么都看不到。

### gl\_PointSize

我们可以使用的另一个可用于渲染的基本图形(primitive)是 `GL_POINTS`， 使用它每个顶点作为一个基本图形， 被渲染为一个点(point)。可以使用 `glPointSize` 函数来设置这个点的大小， 但我们还可以在顶点着色器里修改点的大小。

`GLSL` 有另一个输出变量叫做 `gl_PointSize`， 他是一个 `float` 变量， 你可以以像素的方式设置点的高度和宽度。它在着色器中描述每个顶点做为点被绘制出来的大小。

在着色器中影响点的大小默认是关闭的， 但是如果你打算开启它， 你需要开启 `OpenGL` 的 `GL_PROGRAM_POINT_SIZE`:

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

把点的大小设置为裁切空间的 `z` 值， 这样点的大小就等于顶点距离观察者的距离， 这是一种影响点的大小的方式。当顶点距离观察者更远的时候， 它就会变得更大。

```
void main()
```

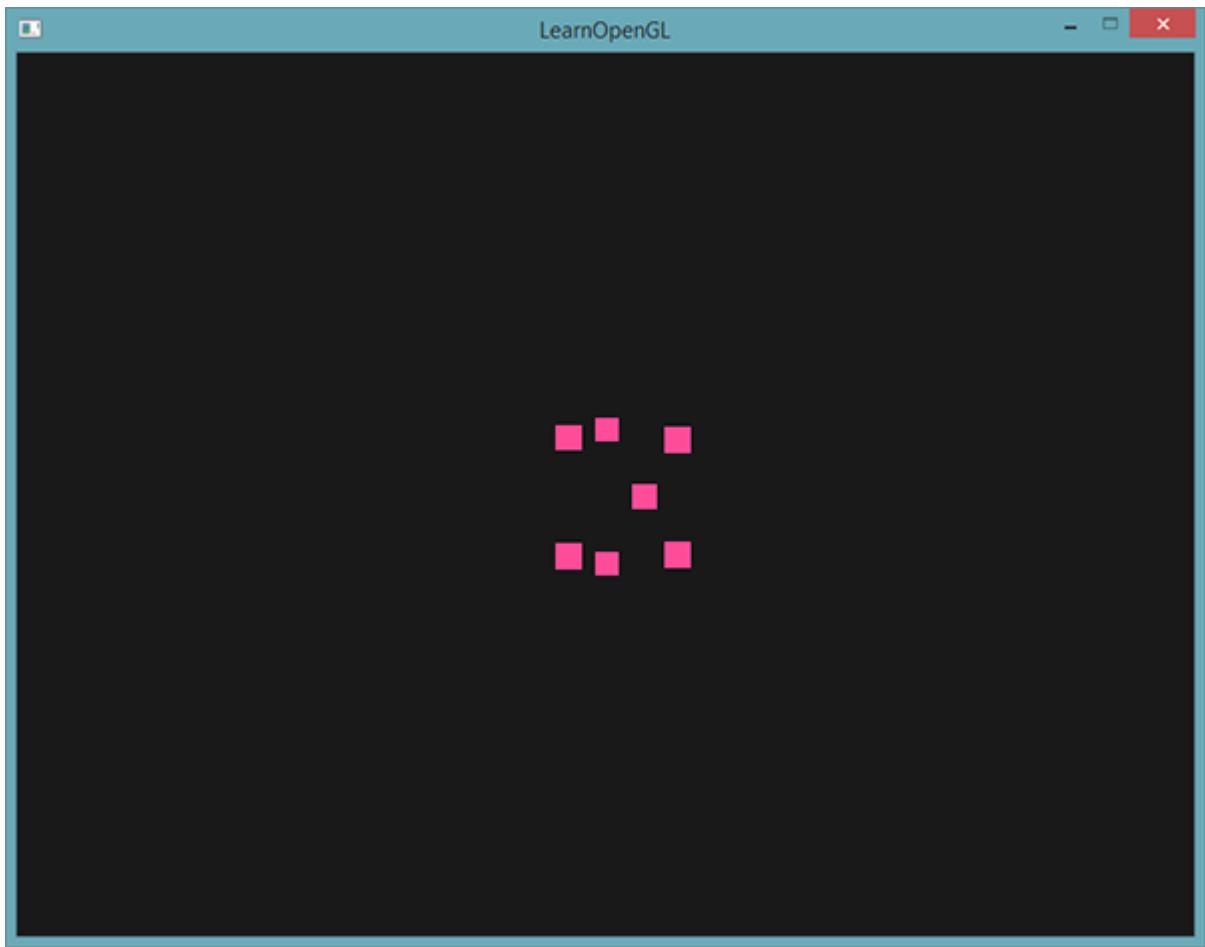
```
{
```

```
 gl_Position = projection * view * model * vec4(position, 1.0f);
```

```
 gl_PointSize = gl_Position.z;
```

```
}
```

结果是我们绘制的点距离我们越远就越大:



想象一下，每个顶点表示出来的点的大小的不同，如果用在像粒子生成之类的技术里会挺有意思的。

#### #### 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；我们可以向它们写入数据来影响结果。顶点着色器为我们提供了一个有趣的输入变量，我们只能从它那里读取，这个变量叫做 `gl_VertexID`。

`gl_VertexID` 是个整型变量，它储存着我们绘制的当前顶点的 ID。当进行索引渲染（indexed rendering，使用 `glDrawElements` 渲染）时，这个变量保存着当前绘制的顶点的索引。当用的不是索引绘制（`glDrawArrays`）时，这个变量保存的是从渲染开始起直到当前处理的这个顶点的（当前顶点）编号。

尽管目前看似没用，但是我们最好知道我们能获取这样的信息。

## 片段着色器的变量

在片段着色器中也有一些有趣的变量。GLSL 给我们提供了两个有意思的输入变量，它们是 `gl_FragCoord` 和 `gl_FrontFacing`。

### gl\_FragCoord

在讨论深度测试的时候，我们已经看过 `gl_FragCoord` 好几次了，因为 `gl_FragCoord` 向量的 z 元素和特定的 `fragment` 的深度值相等。然而，我们也可以使用这个向量的 x 和 y 元素来实现一些有趣的效果。

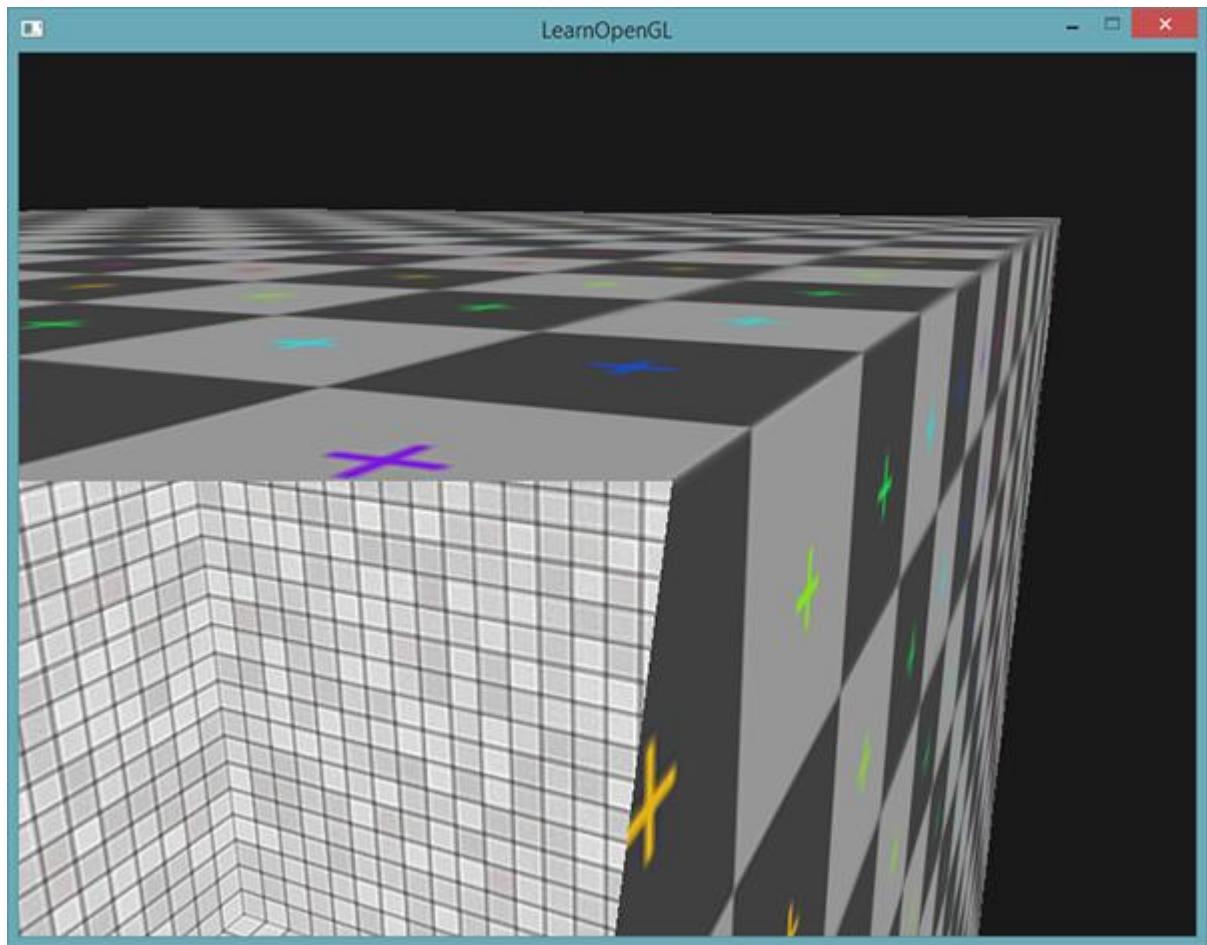
`gl_FragCoord` 的 x 和 y 元素是当前片段的窗口空间坐标（window-space coordinate）。它们的起始处是窗口的左下角。如果我们的窗口是 800×600 的，那么一个片段的窗口空间坐标 x 的范围就在 0 到 800 之间，y 在 0 到 600 之间。

我们可以使用片段着色器基于片段的窗口坐标计算出一个不同的颜色。

`gl_FragCoord` 变量的一个常用的方式是与一个不同的片段计算出来的视频输出进行对比，通常在技术演示中常见。比如我们可以把屏幕分为两个部分，窗口的左侧渲染一个输出，窗口的右边渲染另一个输出。下面是一个基于片段的窗口坐标的位置的不同输出不同的颜色的片段着色器：

```
void main()
{
 if(gl_FragCoord.x < 400)
 color = vec4(1.0f, 0.0f, 0.0f, 1.0f);
 else
 color = vec4(0.0f, 1.0f, 0.0f, 1.0f);
}
```

因为窗口的宽是 800，当一个像素的 x 坐标小于 400，那么它一定在窗口的左边，这样我们就让物体有个不同的颜色。



我们现在可以计算出两个完全不同的片段着色器结果，每个显示在窗口的一端。这对于测试不同的光照技术很有好处。

### gl\_FrontFacing

片段着色器另一个有意思的输入变量是 `gl_FrontFacing` 变量。在面剔除教程中，我们提到过 OpenGL 可以根据顶点绘制顺序弄清楚一个面是正面还是背面。如果我们不适用面剔除，那么 `gl_FrontFacing` 变量能告诉我们当前片段是某个正面的一部分还是背面的一部分。然后我们可以决定做一些事情，比如为正面计算出不同的颜色。

`gl_FrontFacing` 变量是一个布尔值，如果当前片段是正面的一部分那么就是 `true`，否则就是 `false`。这样我们可以创建一个立方体，里面和外面使用不同的纹理：

```
#version 330 core
```

```
out vec4 color;
```

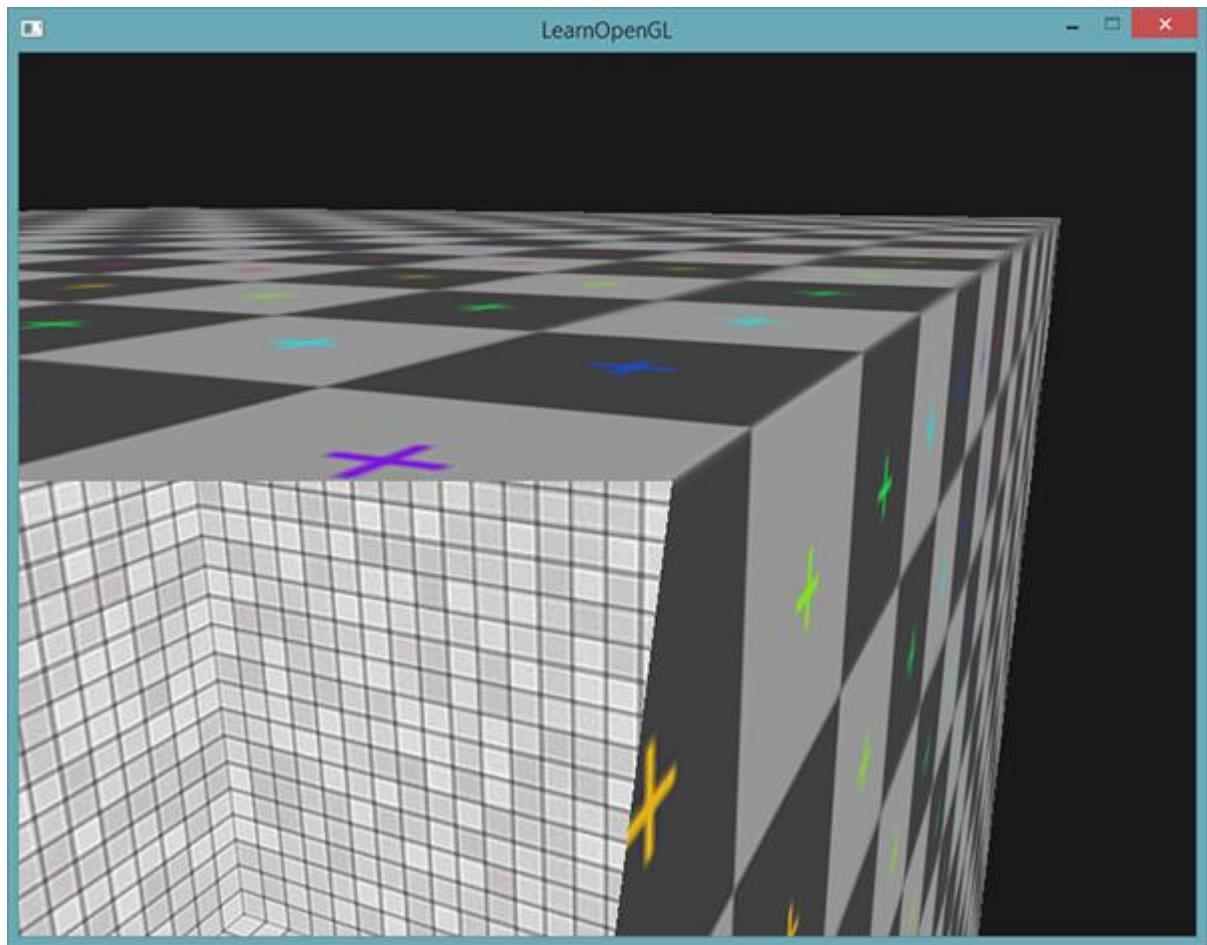
```
in vec2 TexCoords;

uniform sampler2D frontTexture;

uniform sampler2D backTexture;

void main()
{
 if(gl_FrontFacing)
 color = texture(frontTexture, TexCoords);
 else
 color = texture(backTexture, TexCoords);
}
```

如果我们从箱子的一角往里看，就能看到里面用的是另一个纹理。



注意，如果你开启了面剔除，你就看不到箱子里面有任何东西了，所以此时使用 `gl_FrontFacing` 毫无意义。

## gl\_FragDepth

输入变量 `gl_FragCoord` 让我们可以读得当前片段的窗口空间坐标和深度值，但是它是只读的。我们不能影响到这个片段的窗口屏幕坐标，但是可以设置这个像素的深度值。**GLSL** 给我们提供了一个叫做 `gl_FragDepth` 的变量，我们可以用它在着色器中遂舍之像素的深度值。

为了在着色器中设置深度值，我们简单的写一个 0.0 到 1.0 之间的 `float` 数，赋值给这个输出变量：

```
gl_FragDepth = 0.0f; //现在片段的深度值被设为0
```

如果着色器中没有像 `gl_FragDepth` 变量写入，它就会自动采用 `gl_FragCoord.z` 的值。

我们自己设置深度值有一个显著缺点，因为只要我们在片段着色器中对 `gl_FragDepth` 写入什么，OpenGL 就会关闭所有的前置深度测试。它被关闭的原因是，在我们运行片段着色器之前 OpenGL 搞不清像素的深度值，因为片段着色器可能会完全改变这个深度值。

因此，你需要考虑到 `gl_FragDepth` 写入所带来的性能的下降。然而从 OpenGL4.2 起，我们仍然可以对二者进行一定的调和，这需要在片段着色器的顶部使用深度条件（depth condition）来重新声明 `gl_FragDepth`：

```
layout (depth_<condition>) out float gl_FragDepth;
```

condition 可以使用下面的值：

Condition	描述
any	默认值。前置深度测试是关闭的，你失去了很多性能表现
greater	深度值只能比 <code>gl_FragCoord.z</code> 大
less	深度值只能设置得比 <code>gl_FragCoord.z</code> 小
unchanged	如果写入 <code>gl_FragDepth</code> ，你就会写 <code>gl_FragCoord.z</code>

如果把深度条件定义为 `greater` 或 `less`，OpenGL 会假定你只写入比当前的像素深度值的深度值大或小的。

下面是一个在片段着色器里增加深度值的例子，不过仍可开启前置深度测试：

```
#version 330 core

layout (depth_greater) out float gl_FragDepth;

out vec4 color;

void main()
{
 color = vec4(1.0f);
 gl_FragDepth = gl_FragCoord.z + 0.1f;
```

```
}
```

一定要记住这个功能只在 OpenGL4.2 以上版本才有。

## 接口块(Interface blocks)

到目前位置，每次我们打算从顶点向片段着色器发送数据，我们都会声明一个相互匹配的输出/输入变量。从一个着色器向另一个着色器发送数据，一次将它们声明好是最简单的方式，但是随着应用变得越来越大，你也许会打算发送的不仅仅是变量，最好还可以包括数组和结构体。

为了帮助我们组织这些变量，GLSL 为我们提供了一些叫做接口块(Interface blocks)的东西，好让我们能够组织这些变量。声明接口块和声明 struct 有点像，不同之处是它现在基于块（block），使用 in 和 out 关键字来声明，最后它将成为一个输入或输出块（block）。

```
#version 330 core

layout (location = 0) in vec3 position;

layout (location = 1) in vec2 texCoords;

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

out VS_OUT
{
 vec2 TexCoords;
} vs_out;

void main()
```

```
{
```

```
 gl_Position = projection * view * model * vec4(position, 1.0f);
```

```
 vs_out.TexCoords = texCoords;
```

```
}
```

这次我们声明一个叫做 `vs_out` 的接口块，它把我们需要发送给下个阶段着色器的所有输出变量组合起来。虽然这是一个微不足道的例子，但是你可以想象一下，它的确能够帮助我们组织着色器的输入和输出。当我们希望把着色器的输入和输出组织成数组的时候它就变得很有用，我们会在下节几何着色器(geometry)中见到。

然后，我们还需要在下一个着色器——片段着色器中声明一个输入 `interface block`。块名（`block name`）应该是一样的，但是实例名可以是任意的。

```
#version 330 core
```

```
out vec4 color;
```

```
in VS_OUT
```

```
{
```

```
 vec2 TexCoords;
```

```
} fs_in;
```

```
uniform sampler2D texture;
```

```
void main()
```

```
{
```

```
 color = texture(texture, fs_in.TexCoords);
```

```
}
```

如果两个 `interface block` 名一致，它们对应的输入和输出就会匹配起来。这是另一个可以帮助我们组织代码的有用功能，特别是在跨着色阶段的情况，比如几何着色器。

## uniform 缓冲对象 (Uniform buffer objects)

我们使用 OpenGL 很长时间了，也学到了一些很酷的技巧，但是产生了一些烦恼。比如说，当时用一个以上的着色器的时候，我们必须一次次设置 `uniform` 变量，尽管对于每个着色器来说它们都是一样的，所以为什么还麻烦地多次设置它们呢？

OpenGL 为我们提供了一个叫做 `uniform` 缓冲对象的工具，使我们能够声明一系列的全局 `uniform` 变量，它们会在几个着色器程序中保持一致。当时用 `uniform` 缓冲的对象时相关的 `uniform` 只能设置一次。我们仍需为每个着色器手工设置唯一的 `uniform`。创建和配置一个 `uniform` 缓冲对象需要费点功夫。

因为 `uniform` 缓冲对象是一个缓冲，因此我们可以使用 `glGenBuffers` 创建一个，然后绑定到 `GL_UNIFORM_BUFFER` 缓冲目标上，然后把所有相关 `uniform` 数据存入缓冲。有一些原则，像 `uniform` 缓冲对象如何储存数据，我们会在稍后讨论。首先我们我们在一个简单的顶点着色器中，用 `uniform` 块(`uniform block`)储存投影和视图矩阵：

```
#version 330 core

layout (location = 0) in vec3 position;

layout (std140) uniform Matrices
{
 mat4 projection;
 mat4 view;
};

uniform mat4 model;
```

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

前面，大多数例子里我们在每次渲染迭代，都为 `projection` 和 `view` 矩阵设置 `uniform`。这个例子里使用了 `uniform` 缓冲对象，这非常有用，因为这些矩阵我们设置一次就行了。

在这里我们声明了一个叫做 `Matrices` 的 `uniform` 块，它储存两个  $4 \times 4$  矩阵。在 `uniform` 块中的变量可以直接获取，而不用使用 `block` 名作为前缀。接着我们在缓冲中储存这些矩阵的值，每个声明了这个 `uniform` 块的着色器都能够获取矩阵。

现在你可能会奇怪 `layout(std140)` 是什么意思。它的意思是说当前定义的 `uniform` 块为它的内容使用特定的内存布局，这个声明实际上是设置 `uniform` 块布局 (`uniform block layout`)。

## uniform 块布局(uniform block layout)

一个 `uniform` 块的内容被储存到一个缓冲对象中，实际上就是在一块内存中。因为这块内存也不清楚它保存着什么类型的数据，我们就必须告诉 `OpenGL` 哪一块内存对应着色器中哪一个 `uniform` 变量。

假想下面的 `uniform` 块在一个着色器中：

```
layout (std140) uniform ExampleBlock
{
 float value;
 vec3 vector;
 mat4 matrix;
 float values[3];
```

```
bool boolean;

int integer;

};
```

我们所希望知道的是每个变量的大小（以字节为单位）和偏移量（从 **block** 的起始处），所以我们可以以各自的顺序把它们放进一个缓冲里。每个元素的大小在 OpenGL 中都很清楚，直接与 C++ 数据类型呼应，向量和矩阵是一个 **float** 序列（数组）。OpenGL 没有澄清的是变量之间的间距。这让硬件能以它认为合适的位置方式变量。比如有些硬件可以在 **float** 旁边放置一个 **vec3**。不是所有硬件都能这样做，在 **vec3** 旁边附加一个 **float** 之前，给 **vec3** 加一个边距使之成为 4 个（空间连续的）**float** 数组。功能很好，但对于来说用起来不方便。

GLSL 默认使用的 **uniform** 内存布局叫做共享布局(**shared layout**)，叫共享是因为一旦偏移量被硬件定义，它们就会持续地被多个程序所共享。使用共享布局，GLSL 可以为优化而重新放置 **uniform** 变量，只要变量的顺序保持完整。因为我们不知道每个 **uniform** 变量的偏移量是多少，所以我们也就不知道如何精确地填充 **uniform** 缓冲。我们可以使用像 `glGetUniformIndices` 这样的函数来查询这个信息，但是这超出了本节教程的范围。

由于共享布局给我们做了一些空间优化。通常在实践中并不适用分享布局，而是使用 **std140** 布局。**std140** 通过一系列的规则的规范声明了它们各自的偏移量，**std140** 布局为每个变量类型显式地声明了内存的布局。由于被显式的提及，我们就可以手工算出每个变量的偏移量。

每个变量都有一个基线对齐(**base alignment**)，它等于在一个 **uniform** 块中这个变量所占的空间（包含边距），这个基线对齐是使用 **std140** 布局原则计算出来的。然后，我们为每个变量计算出它的对齐偏移(**aligned offset**)，这是一个变量从块（**block**）开始处的字节偏移量。变量对齐的字节偏移一定等于它的基线对齐的倍数。

准确的布局规则可以在 [OpenGL 的 uniform 缓冲规范](#) 中得到，但我们会列出最常见的规范。GLSL 中每个变量类型比如 **int**、**float** 和 **bool** 被定义为 4 字节，每 4 字节被表示为 N。

类型	布局规范
像 <b>int</b> 和 <b>bool</b> 这样的标量	每个标量的基线为 N

类型	布局规范
向量	每个向量的基线是 $2N$ 或 $4N$ 大小。这意味着 <code>vec3</code> 的基线为 $4N$
标量与向量数组	每个元素的基线与 <code>vec4</code> 的相同
矩阵	被看做是存储着大量向量的数组，每个元素的基数与 <code>vec4</code> 相同
结构体	根据以上规则计算其各个元素，并且间距必须是 <code>vec4</code> 基线的倍数

像 OpenGL 大多数规范一样，举个例子就很容易理解。再次利用之前介绍的 uniform 块 `ExampleBlock`，我们用 std140 布局，计算它的每个成员的 aligned offset（对齐偏移）：

```
layout (std140) uniform ExampleBlock
{
 // base alignment ----- // aligned offset

 float value; // 4 // 0

 vec3 vector; // 16 // 16 (必须是 16 的倍
 数, 因此 4->16)

 mat4 matrix; // 16 // 32 (第 0 行)
 // 16 // 48 (第 1 行)
 // 16 // 64 (第 2 行)
 // 16 // 80 (第 3 行)

 float values[3]; // 16 (数组中的标量与 vec4 相同)//96 (values[0])
 // 16 // 112 (values[1])
 // 16 // 128 (values[2])

 bool boolean; // 4 // 144

 int integer; // 4 // 148
```

```
};
```

尝试自己计算出偏移量，把它们和表格对比，你可以把这件事当作一个练习。使用计算出来的偏移量，根据 `std140` 布局规则，我们可以用 `glBufferSubData` 这样的函数，使用变量数据填充缓冲。虽然不是很高效，但 `std140` 布局可以保证在每个程序中声明的这个 `uniform` 块的布局保持一致。

在定义 `uniform` 块前面添加 `layout (std140)` 声明，我们就能告诉 OpenGL 这个 `uniform` 块使用了 `std140` 布局。另外还有两种其他的布局可以选择，它们需要我们在填充缓冲之前查询每个偏移量。我们已经了解了分享布局（`shared layout`）和其他的布局都将被封装（`packed`）。当使用封装（`packed`）布局的时候，不能保证布局在别的程序中能够保持一致，因为它允许编译器从 `uniform` 块中优化出去 `uniform` 变量，这在每个着色器中都可能不同。

## 使用 `uniform` 缓冲

我们讨论了 `uniform` 块在着色器中的定义和如何定义它们的内存布局，但是我们还没有讨论如何使用它们。

首先我们需要创建一个 `uniform` 缓冲对象，这要使用 `glGenBuffers` 来完成。当我们拥有了一个缓冲对象，我们就把它绑定到 `GL_UNIFORM_BUFFER` 目标上，调用 `glBufferData` 来给它分配足够的空间。

```
GLuint uboExampleBlock;

glGenBuffers(1, &uboExampleBlock);

glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);

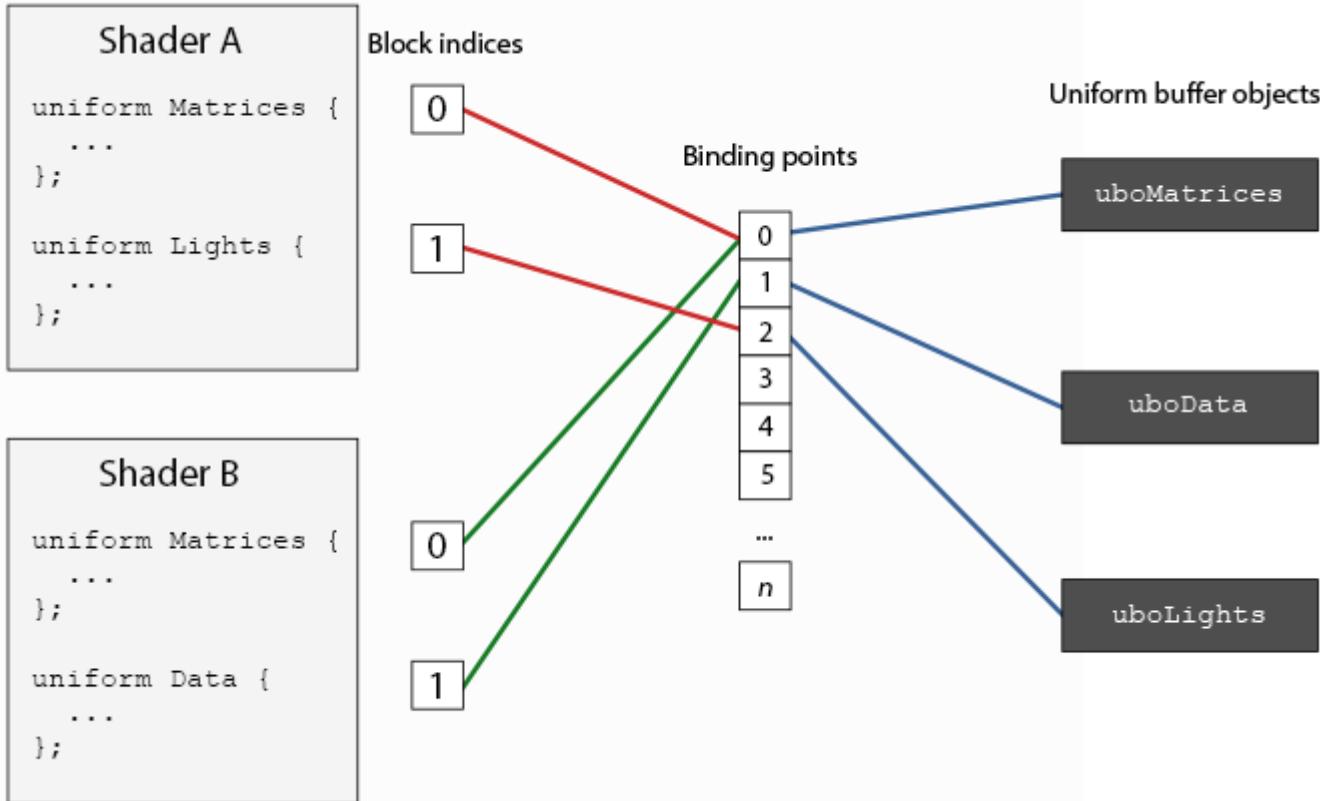
glBufferData(GL_UNIFORM_BUFFER, 150, NULL, GL_STATIC_DRAW); // 分配

150 个字节的内存空间

glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

现在任何时候当我们打算往缓冲中更新或插入数据，我们就绑定到 `uboExampleBlock` 上，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，所有的使用这个缓冲着色器就都会使用它更新的数据了。但是，OpenGL 是如何知道哪个 `uniform` 缓冲对应哪个 `uniform` 块呢？

在 OpenGL 环境（context）中，定义了若干绑定点（binding points），在哪儿我们可以把一个 uniform 缓冲链接上去。当我们创建了一个 uniform 缓冲，我们把它链接到一个这个绑定点上，我们也把着色器中 uniform 块链接到同一个绑定点上，这样就把它们链接到一起了。下面的图标表示了这点：



你可以看到，我们可以将多个 uniform 缓冲绑定到不同绑定点上。因为着色器 A 和着色器 B 都有一个链接到同一个绑定点 0 的 uniform 块，它们的 uniform 块分享同样的 uniform 数据—`uboMatrices`。有一个前提条件是两个着色器必须都定义了 `Matrices` 这个 uniform 块。

我们调用 `glUniformBlockBinding` 函数来把 uniform 块设置到一个特定的绑定点上。函数的第一个参数是一个程序对象，接着是一个 uniform 块索引（uniform block index）和打算链接的绑定点。uniform 块索引是一个着色器中定义的 uniform 块的索引位置，可以调用 `glGetUniformBlockIndex` 来获取这个值，这个函数接收一个程序对象和 uniform 块的名字。我们可以从图表设置 `Lights` 这个 uniform 块链接到绑定点 2：

```
GLuint lights_index = glGetUniformLocation(shaderA.Program,
"Lights");

glUniformBlockBinding(shaderA.Program, lights_index, 2);
```

注意，我们必须在每个着色器中重复做这件事。

从 OpenGL4.2 起，也可以在着色器中通过添加另一个布局标识符来储存一个 uniform 块的绑定点，就不用我们调用 `glGetUniformBlockIndex` 和 `glUniformBlockBinding` 了。下面的代表显式设置了 Lights 这个 uniform 块的绑定点：

```
layout(std140, binding = 2) uniform Lights { ... };
```

然后我们还需要把 uniform 缓冲对象绑定到同样的绑定点上，这个可以使用 `glBindBufferBase` 或 `glBindBufferRange` 来完成。

```
glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboExampleBlock);
```

// 或者

```
glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBlock, 0, 150);
```

函数 `glBindBufferBase` 接收一个目标、一个绑定点索引和一个 uniform 缓冲对象作为它的参数。这个函数把 `uboExampleBlock` 链接到绑定点 2 上面，自此绑定点所链接的两端都链接在一起了。你还可以使用 `glBindBufferRange` 函数，这个函数还需要一个偏移量和大小作为参数，这样你就可以只把一定范围的 uniform 缓冲绑定到一个绑定点上了。使用 `glBindBufferRange` 函数，你能够将多个不同的 uniform 块链接到同一个 uniform 缓冲对象上。

现在所有事情都做好了，我们可以开始向 uniform 缓冲添加数据了。我们可以使用 `glBufferSubData` 将所有数据添加为一个单独的字节数组或者更新缓冲的部分内容，只要我们愿意。为了更新 uniform 变量 `boolean`，我们可以这样更新 uniform 缓冲对象：

```
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
```

```
GLint b = true; // GLSL 中的布尔值是 4 个字节，因此我们将它创建为一个 4
```

字节的整数

```
glBufferSubData(GL_UNIFORM_BUFFER, 142, 4, &b);
```

```
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

同样的处理也能够应用到 `uniform` 块中其他 `uniform` 变量上。

## 一个简单的例子

我们来师范一个真实的使用 `uniform` 缓冲对象的例子。如果我们回头看看前面所有演示的代码，我们一直使用了 3 个矩阵：投影、视图和模型矩阵。所有这些矩阵中，只有模型矩阵是频繁变化的。如果我们有多个着色器使用了这些矩阵，我们可能最好还是使用 `uniform` 缓冲对象。

我们将把投影和视图矩阵储存到一个 `uniform` 块中，它被取名为 `Matrices`。我们不打算储存模型矩阵，因为模型矩阵会频繁在着色器间更改，所以使用 `uniform` 缓冲对象真的不会带来什么好处。

```
#version 330 core
```

```
layout (location = 0) in vec3 position;
```

```
layout (std140) uniform Matrices
```

```
{
```

```
 mat4 projection;
```

```
 mat4 view;
```

```
};
```

```
uniform mat4 model;
```

```
void main()
```

```
{
 gl_Position = projection * view * model * vec4(position, 1.0);
}
```

这儿没什么特别的，除了我们现在使用了一个带有 `std140` 布局的 `uniform` 块。我们在例程中将显示 4 个立方体，每个立方体都使用一个不同的着色器程序。4 个着色器程序使用同样的顶点着色器，但是它们将使用各自的片段着色器，每个片段着色器输出一个单色。

首先，我们把顶点着色器的 `uniform` 块设置为绑定点 0。注意，我们必须为每个着色器做这件事。

```
GLuint uniformBlockIndexRed =
glGetUniformBlockIndex(shaderRed.Program, "Matrices");

GLuint uniformBlockIndexGreen =
glGetUniformBlockIndex(shaderGreen.Program, "Matrices");

GLuint uniformBlockIndexBlue =
glGetUniformBlockIndex(shaderBlue.Program, "Matrices");

GLuint uniformBlockIndexYellow =
glGetUniformBlockIndex(shaderYellow.Program, "Matrices");

glUniformBlockBinding(shaderRed.Program, uniformBlockIndexRed, 0);
glUniformBlockBinding(shaderGreen.Program, uniformBlockIndexGreen,
0);
glUniformBlockBinding(shaderBlue.Program, uniformBlockIndexBlue,
0);
```

```
glUniformBlockBinding(shaderYellow.Program,
```

```
uniformBlockIndexYellow, 0);
```

然后，我们创建真正的 uniform 缓冲对象，并把缓冲绑定到绑定点 0：

```
GLuint 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));
```

我们先为缓冲分配足够的内存，它等于 `glm::mat4` 的 2 倍。`GLM` 的矩阵类型的大小直接对应于 `GLSL` 的 `mat4`。然后我们把一个特定范围的缓冲链接到绑定点 0，这个例子中应该是整个缓冲。

现在所有要做的事只剩下填充缓冲了。如果我们把视野（`field of view`）值保持为恒定的投影矩阵（这样就不会有摄像机缩放），我们只要在程序中定义它一次就行了，这也意味着我们只需向缓冲中把它插入一次。因为我们已经在缓冲对象中分配了足够的内存，我们可以在我们进入游戏循环之前使用 `glBufferSubData` 来储存投影矩阵：

```
glm::mat4 projection = glm::perspective(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);
```

这里我们用投影矩阵储存了 uniform 缓冲的前半部分。在我们在每次渲染迭代绘制物体前，我们用视图矩阵更新缓冲的第二个部分：

```
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);
```

这就是 uniform 缓冲对象。每个包含着 **Matrices** 这个 uniform 块的顶点着色器都将对应 **uboMatrices** 所储存的数据。所以如果我们现在使用 4 个不同的着色器绘制 4 个立方体，它们的投影和视图矩阵都是一样的：

```
glBindVertexArray(cubeVAO);

shaderRed.Use();

glm::mat4 model;

model = glm::translate(model, glm::vec3(-0.75f, 0.75f, 0.0f)); // 移
动到左上方

glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

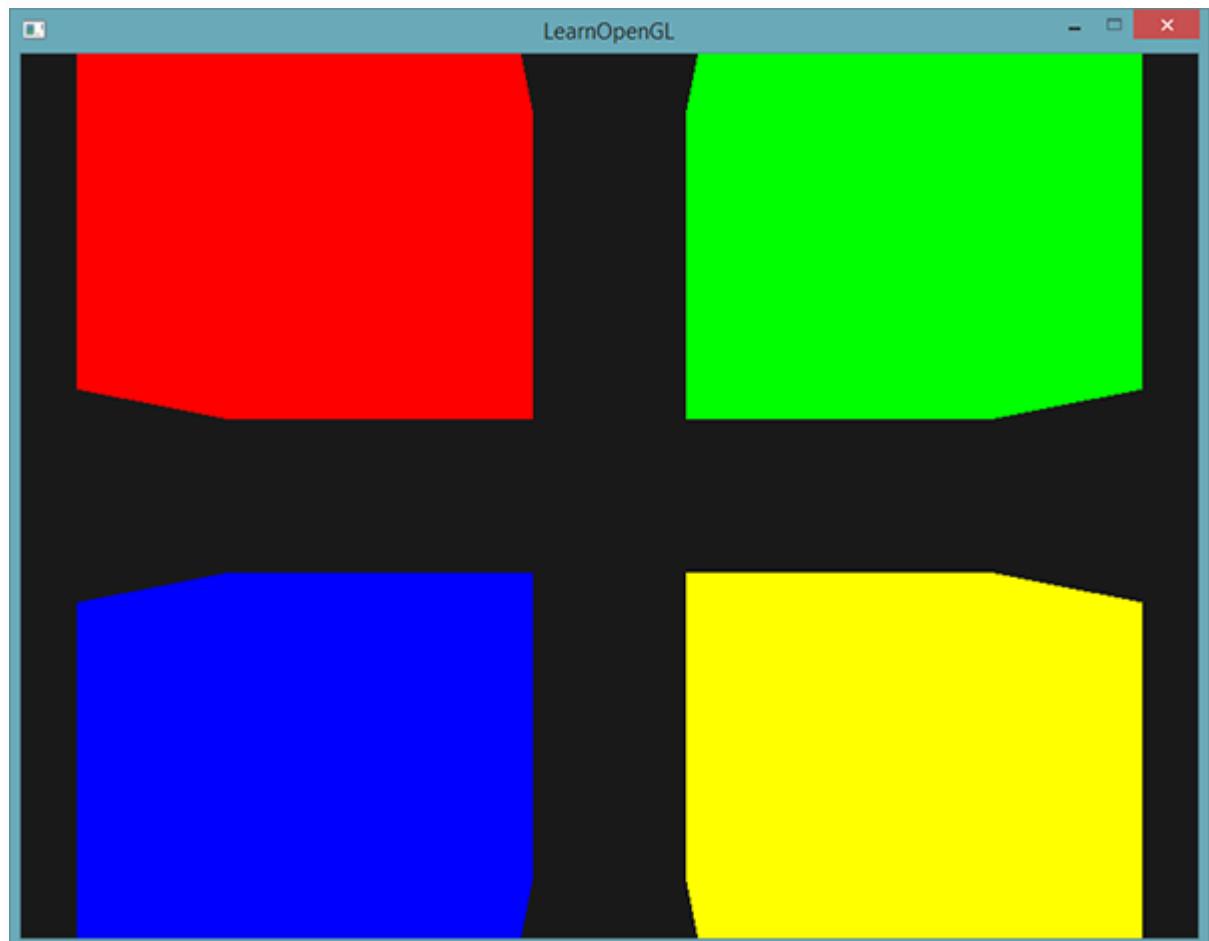
glDrawArrays(GL_TRIANGLES, 0, 36);

// ... 绘制绿色立方体

// ... 绘制蓝色立方体
```

```
// ... 绘制黄色立方体
glBindVertexArray(0);
```

我们只需要在去设置一个 `model` 的 `uniform` 即可。在一个像这样的场景中使用 `uniform` 缓冲对象在每个着色器中可以减少 `uniform` 的调用。最后效果看起来像这样：



通过改变模型矩阵，每个立方体都移动到窗口的一边，由于片段着色器不同，物体的颜色也不同。这是一个相对简单的场景，我们可以使用 `uniform` 缓冲对象，但是任何大型渲染程序有成百上千的活动着色程序，彼时 `uniform` 缓冲对象就会闪闪发光了。

你可以[在这里获得例程的完整源码](#)。

`uniform` 缓冲对象比单独的 `uniform` 有很多好处。第一，一次设置多个 `uniform` 比一次设置一个速度快。第二，如果你打算改变一个横跨多个着色器的 `uniform`，在 `uniform` 缓冲中只需更改一次。最后一个好处可能不是很明显，使用 `uniform`

缓冲对象你可以在着色器中使用更多的 uniform。OpenGL 有一个对可使用 uniform 数据的数量的限制，可以用 `GL_MAX_VERTEX_UNIFORM_COMPONENTS` 来获取。当使用 uniform 缓冲对象中，这个限制的阈限会更高。所以无论何时，你达到了 uniform 的最大使用数量（比如做骨骼动画的时候），你可以使用 uniform 缓冲对象。

## 几何着色器

原文	<a href="#">Geometry Shader</a>
作者	JoeyDeVries
翻译	<a href="#">Django</a>
校对	<a href="#">Geequlim</a>

## 几何着色器(Geometry Shader)

在顶点和片段着色器之间有一个可选的着色器，叫做几何着色器（geometry shader）。几何着色器以一个或多个表示为一个单独基本图形（primitive）的顶点作为输入，比如可以是一个点或者三角形。几何着色器在将这些顶点发送到下一个着色阶段之前，可以将这些顶点转变为它认为合适的内容。几何着色器有意思的地方在于它可以把（一个或多个）顶点转变为完全不同的基本图形（primitive），从而生成比原来多得多的顶点。

我们直接用一个例子深入了解一下：

```
#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();
```

```
}
```

每个几何着色器开始位置我们需要声明输入的基本图形(primitive)类型，这个输入是我们从顶点着色器中接收到的。我们在 `in` 关键字前面声明一个 `layout` 标识符。这个输入 `layout` 修饰符可以从一个顶点着色器接收以下基本图形值：

基本图形	描述
points	绘制 <code>GL_POINTS</code> 基本图形的时候 (1)
lines	当绘制 <code>GL_LINES</code> 或 <code>GL_LINE_STRIP</code> (2) 时
lines_adjacency	<code>GL_LINES_ADJACENCY</code> 或 <code>GL_LINE_STRIP_ADJACENCY</code> (4)
triangles	<code>GL_TRIANGLES</code> , <code>GL_TRIANGLE_STRIP</code> 或 <code>GL_TRIANGLE_FAN</code> (3)
triangles_adjacency	<code>GL_TRIANGLES_ADJACENCY</code> 或 <code>GL_TRIANGLE_STRIP_ADJACENCY</code> (6)

这是我们能够给渲染函数的几乎所有的基本图形。如果我们选择以 `GL_TRIANGLES` 绘制顶点，我们要把输入修饰符设置为 `triangles`。括号里的数字代表一个基本图形所能包含的最少的顶点数。

当我们需要指定一个几何着色器所输出的基本图形类型时，我们就在 `out` 关键字前面加一个 `layout` 修饰符。和输入 `layout` 标识符一样，输出的 `layout` 标识符也可以接受以下基本图形值：

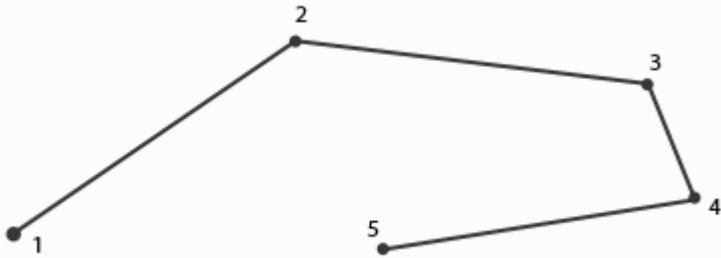
- points
- line\_strip
- triangle\_strip

使用这 3 个输出修饰符我们可以从输入的基本图形创建任何我们想要的形状。为了生成一个三角形，我们定义一个 `triangle_strip` 作为输出，然后输出 3 个顶点。

几何着色器同时希望我们设置一个它能输出的顶点数量的最大值（如果你超出了这个数值，OpenGL 就会忽略剩下的顶点），我们可以在 `out` 关键字的 `layout`

标识符上做这件事。在这个特殊的情况下，我们将使用最大值为 2 个顶点，来输出一个 `line_strip`。

这种情况，你会奇怪什么是线条：一个线条是把多个点链接起来表示出一个连续的线，它最少有两个点来组成。每个后一个点在前一个新渲染的点后面渲染，你可以看看下面的图，其中包含 5 个顶点：



上面的着色器，我们只能输出一个线段，因为顶点的最大值设置为 2。

为生成更有意义的结果，我们需要某种方式从前一个着色阶段获得输出。**GLSL**为我们提供了一个内建变量，它叫做 `gl_in`，它的内部看起来可能像这样：

```
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，前面的教程已经讨论过)，它包含几个有意思的变量，其中最有意思的是 `gl_Position`，它包含着和我们设置的顶点着色器的输出相似的向量。

要注意的是，它被声明为一个数组，因为大多数渲染基本图形由一个以上顶点组成，几何着色器接收一个基本图形的所有顶点作为它的输入。

使用从前一个顶点着色阶段的顶点数据，我们就可以开始生成新的数据了，这是通过 2 个几何着色器函数 `EmitVertex` 和 `EndPrimitive` 来完成的。几何着色器需要你去生成/输出至少一个你定义为输出的基本图形。在我们的例子里我们打算至少生成一个线条（line strip）基本图形。

```
void main() {

 gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);

 EmitVertex();

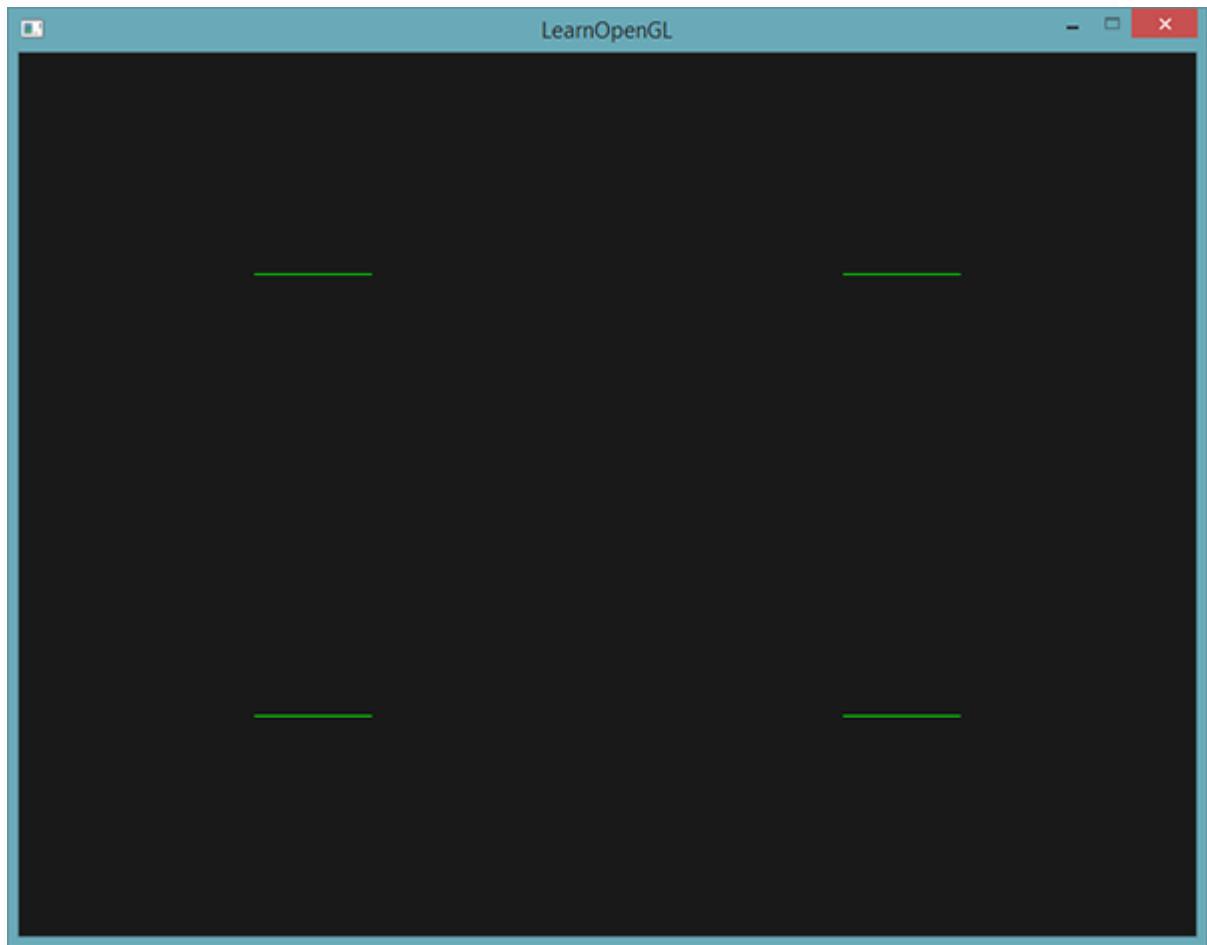
 gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);

 EmitVertex();

 EndPrimitive();
}
```

每次我们调用 `EmitVertex`，当前设置到 `gl_Position` 的向量就会被添加到基本图形上。无论何时调用 `EndPrimitive`，所有为这个基本图形发射出去的顶点都将结合为一个特定的输出渲染基本图形。一个或多个 `EmitVertex` 函数调用后，重复调用 `EndPrimitive` 就能生成多个基本图形。这个特殊的例子里，发射了两个顶点，它们被从顶点原来的位置平移了一段距离，然后调用 `EndPrimitive` 将这两个顶点结合为一个单独的有两个顶点的线条。

现在你了解了几何着色器的工作方式，你就可能猜出这个几何着色器做了什么。这个几何着色器接收一个基本图形——点，作为它的输入，使用输入点作为它的中心，创建了一个水平线基本图形。如果我们渲染它，结果就会像这样：



并不是非常引人注目，但是考虑到它的输出是使用下面的渲染命令生成的就很有意思了：

```
glDrawArrays(GL_POINTS, 0, 4);
```

这是个相对简单的例子，它向你展示了我们如何使用几何着色器来动态地在运行时生成新的形状。本章的后面，我们会讨论一些可以使用几何着色器获得有趣的效果，但是现在我们将以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使用，我们将渲染一个简单的场景，在场景中我们只绘制4个点，这4个点在标准化设备坐标的空间平面上。这些点的坐标是：

```
GLfloat points[] = {
 -0.5f, 0.5f, // 左上方
```

```
 0.5f, 0.5f, // 右上方
 0.5f, -0.5f, // 右下方
 -0.5f, -0.5f // 左下方
};
```

顶点着色器只在 z 平面绘制点，所以我们只需要一个基本顶点着色器：

```
#version 330 core

layout (location = 0) in vec2 position;

void main()
{
 gl_Position = vec4(position.x, position.y, 0.0f, 1.0f);
}
```

我们会简单地为所有点输出绿色，我们直接在片段着色器里进行硬编码：

```
#version 330 core

out vec4 color;

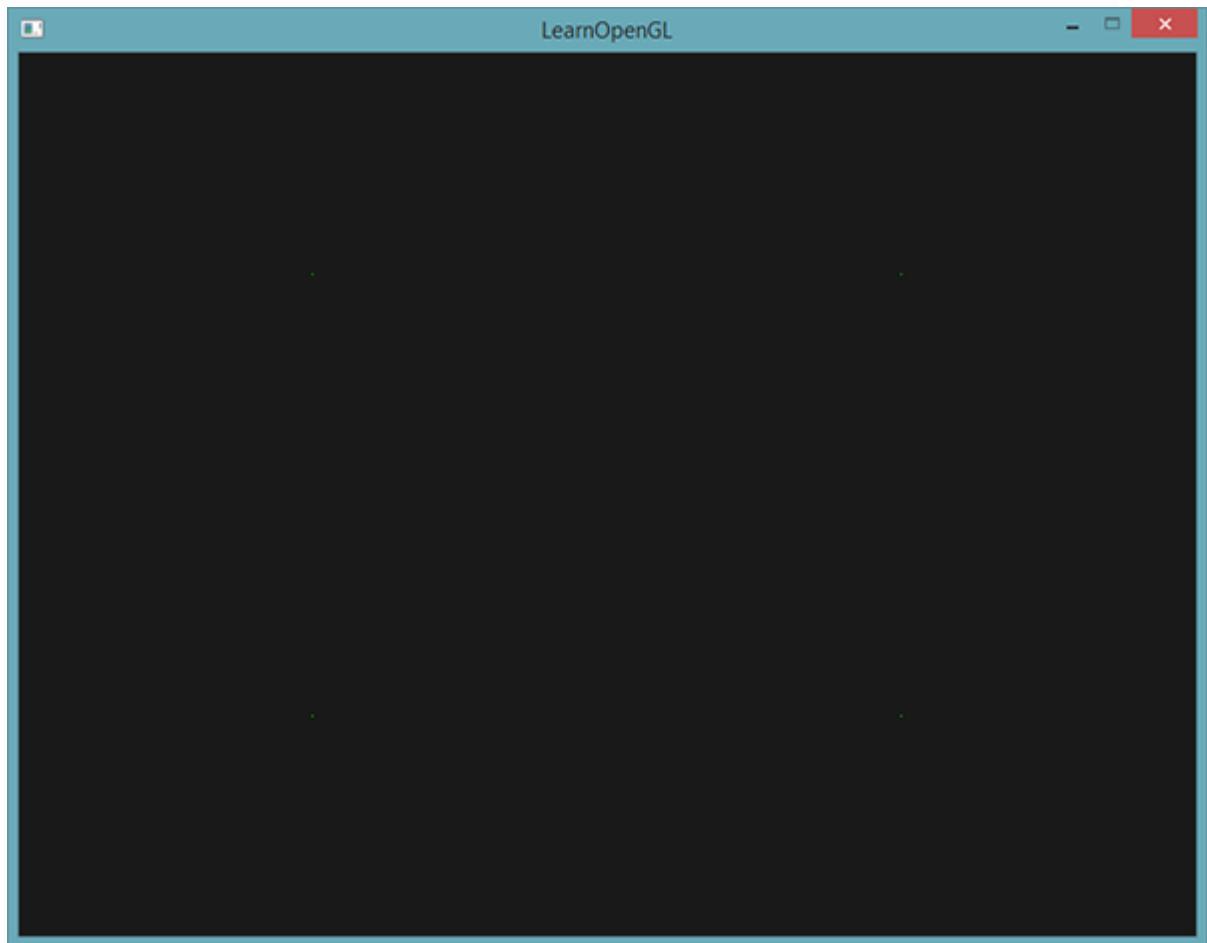
void main()
{
 color = vec4(0.0f, 1.0f, 0.0f, 1.0f);
}
```

为点的顶点生成一个 VAO 和 VBO，然后使用 `glDrawArrays` 进行绘制：

```
shader.Use();
```

```
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, 4);
glBindVertexArray(0);
```

效果是黑色场景中有四个绿点（虽然很难看到）：



但我们不是已经学到了所有内容了吗？对，现在我们将通过为场景添加一个几何着色器来为这个小场景增加点活力。

出于学习的目的我们将创建一个叫 **pass-through** 的几何着色器，它用一个 **point** 基本图形作为它的输入，并把它无修改地传（**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。它简单地将它接收到的输入的无修改的顶点位置发射出去，然后生成一个 point 基本图形。

一个几何着色器需要像顶点和片段着色器一样被编译和链接，但是这次我们将使用 `GL_GEOMETRY_SHADER` 作为着色器的类型来创建这个着色器：

```
geometryShader = glCreateShader(GL_GEOMETRY_SHADER);

glShaderSource(geometryShader, 1, &gShaderCode, NULL);

glCompileShader(geometryShader);

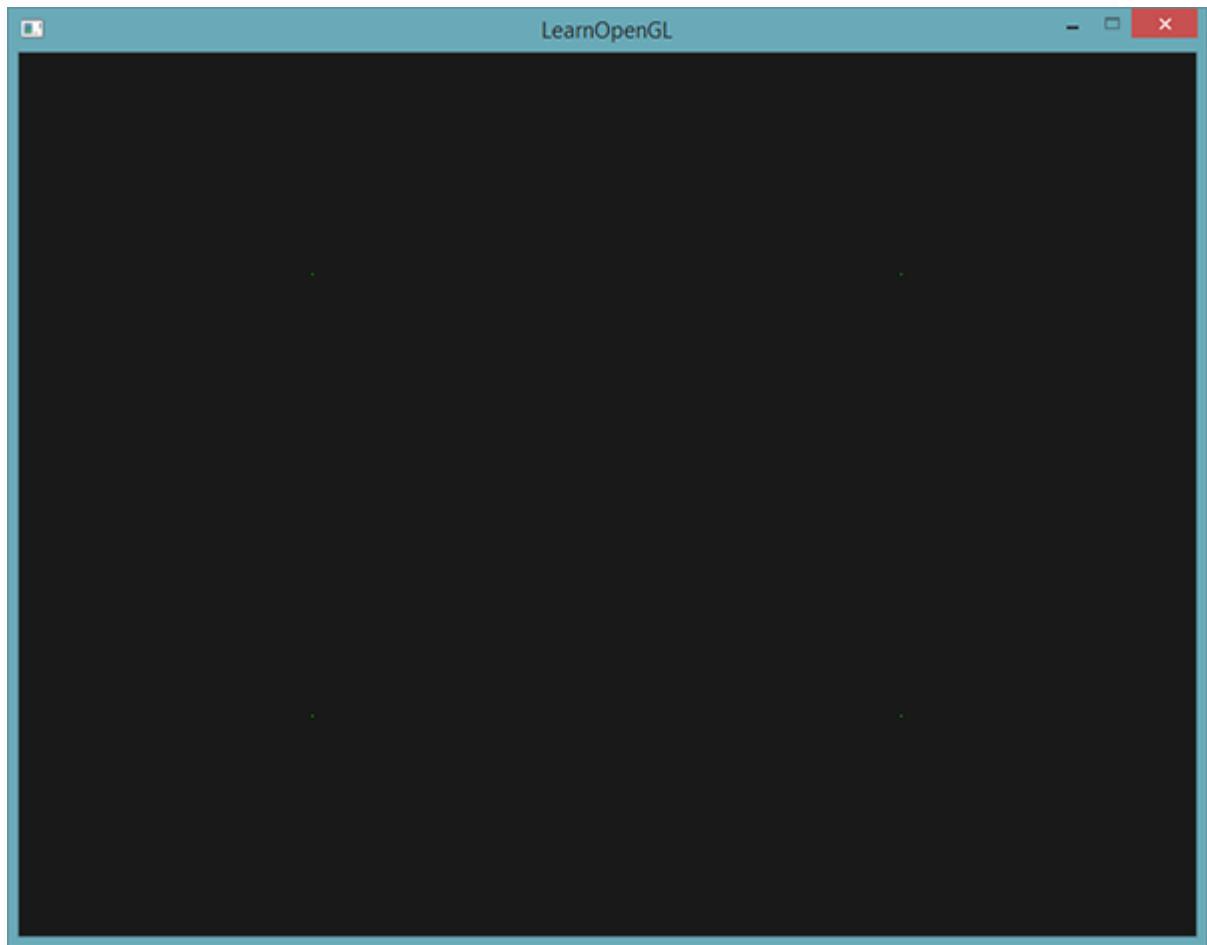
...

glAttachShader(program, geometryShader);

glLinkProgram(program);
```

编译着色器的代码和顶点、片段着色器的基本一样。要记得检查编译和链接错误！

如果你现在编译和运行，就会看到和下面相似的结果：

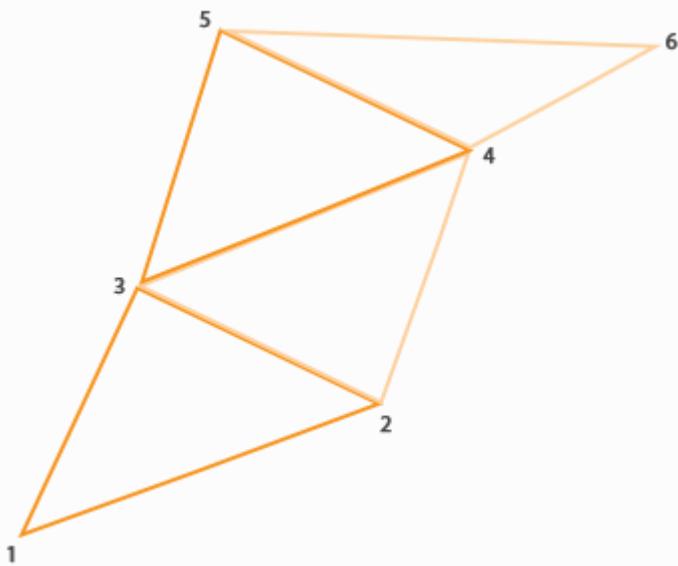


它和没用几何着色器一样！我承认有点无聊，但是事实上，我们仍能绘制证明几何着色器工作了的点，所以现在是时候来做点更有意思的事了！

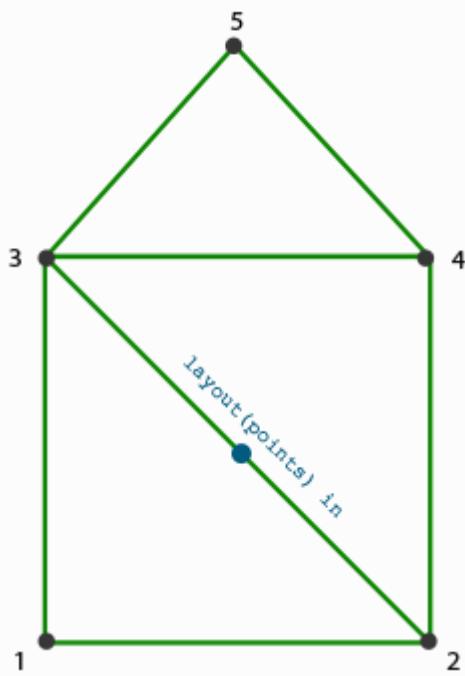
## 创建几个房子

绘制点和线没什么意思，所以我们将在每个点上使用几何着色器绘制一个房子。我们可以通过把几何着色器的输出设置为 `triangle_strip` 来达到这个目的，总共要绘制 3 个三角形：两个用来组成方形和另表示一个屋顶。

在 OpenGL 中三角形带(`triangle strip`)绘制起来更高效，因为它所使用的顶点更少。第一个三角形绘制完以后，每个后续的顶点会生成一个毗连前一个三角形的新三角形：每 3 个毗连的顶点都能构成一个三角形。如果我们有 6 个顶点，它们以三角形带的方式组合起来，那么我们会得到这些三角形：(1, 2, 3)、(2, 3, 4)、(3, 4, 5)、(4, 5, 6) 因此总共可以表示出 4 个三角形。一个三角形带至少要用 3 个顶点才行，它能生曾  $N-2$  个三角形；6 个顶点我们就能创建  $6-2=4$  个三角形。下面的图片表达了这点：



使用一个三角形带作为一个几何着色器的输出，我们可以轻松创建房子的形状，只要以正确的顺序来生成 3 个毗连的三角形。下面的图像显示，我们需要以何种顺序来绘制点，才能获得我们需要的三角形，图上的蓝点代表输入点：



上图的内容转变为几何着色器：

```
#version 330 core
```

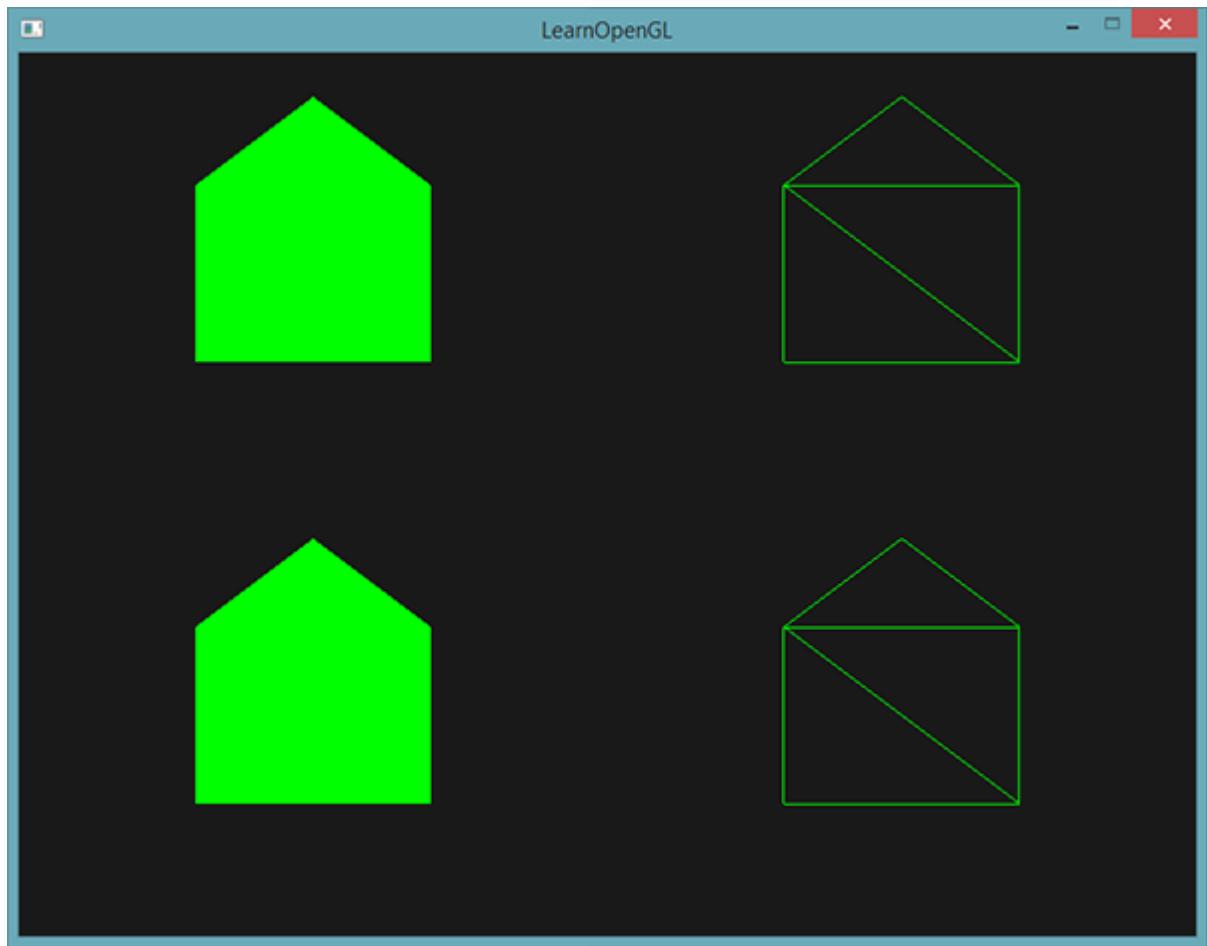
```
layout (points) in;

layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position)
{
 gl_Position = position + vec4(-0.2f, -0.2f, 0.0f, 0.0f); // 1: 左
 下角
 EmitVertex();
 gl_Position = position + vec4(0.2f, -0.2f, 0.0f, 0.0f); // 2: 右
 下角
 EmitVertex();
 gl_Position = position + vec4(-0.2f, 0.2f, 0.0f, 0.0f); // 3: 左
 上
 EmitVertex();
 gl_Position = position + vec4(0.2f, 0.2f, 0.0f, 0.0f); // 4: 右
 上
 EmitVertex();
 gl_Position = position + vec4(0.0f, 0.4f, 0.0f, 0.0f); // 5: 屋
 顶
 EmitVertex();
 EndPrimitive();
}
```

```
void main()
{
 build_house(gl_in[0].gl_Position);
}
```

这个几何着色器生成 5 个顶点，每个顶点是点（point）的位置加上一个偏移量，来组成一个大三角形带。接着最后的基本图形被像素化，片段着色器处理整三角形带，结果是为我们绘制的每个点生成一个绿房子：



可以看到，每个房子实则是由 3 个三角形组成，都是仅仅使用空间中一点来绘制的。绿房子看起来还是不够漂亮，所以我们再给每个房子加一个不同的颜色。我们将在顶点着色器中为每个顶点增加一个额外的代表颜色信息的顶点属性。

下面是更新了的顶点数据：

```
GLfloat 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，使用一个接口块来向几何着色器发送颜色属性：

```
#version 330 core

layout (location = 0) in vec2 position;

layout (location = 1) in vec3 color;

out VS_OUT {
 vec3 color;
} vs_out;

void main()
{
 gl_Position = vec4(position.x, position.y, 0.0f, 1.0f);
 vs_out.color = color;
}
```

接着我们还需要在几何着色器中声明同样的接口块(使用一个不同的接口名)：

```
in VS_OUT {
 vec3 color;
```

```
} gs_in[];
```

因为几何着色器把多个顶点作为它的输入，从顶点着色器来的输入数据总是被以数组的形式表示出来，即使现在我们只有一个顶点。

## Important

我们不是必须使用接口块来把数据发送到几何着色器中。我们还可以这么写：

```
in vec3 vColor[];
```

如果顶点着色器发送的颜色向量是 `out vec3 vColor` 那么接口块就会在比如几何着色器这样的着色器中更轻松地完成工作。事实上，几何着色器的输入可以非常大，把它们组成一个大的接口块数组会更有意义。

然后我们还要为下一个像素着色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（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; // 只有一个输出颜色，所以直接设置为gs_in[0]
```

```
gl_Position = position + vec4(-0.2f, -0.2f, 0.0f, 0.0f); // 1:左
```

下

```
EmitVertex();
```

```
gl_Position = position + vec4(0.2f, -0.2f, 0.0f, 0.0f); // 2:右
```

下

```
EmitVertex();
```

```
gl_Position = position + vec4(-0.2f, 0.2f, 0.0f, 0.0f); // 3:左
```

上

```
EmitVertex();
```

```
gl_Position = position + vec4(0.2f, 0.2f, 0.0f, 0.0f); // 4:右
```

上

```
EmitVertex();
```

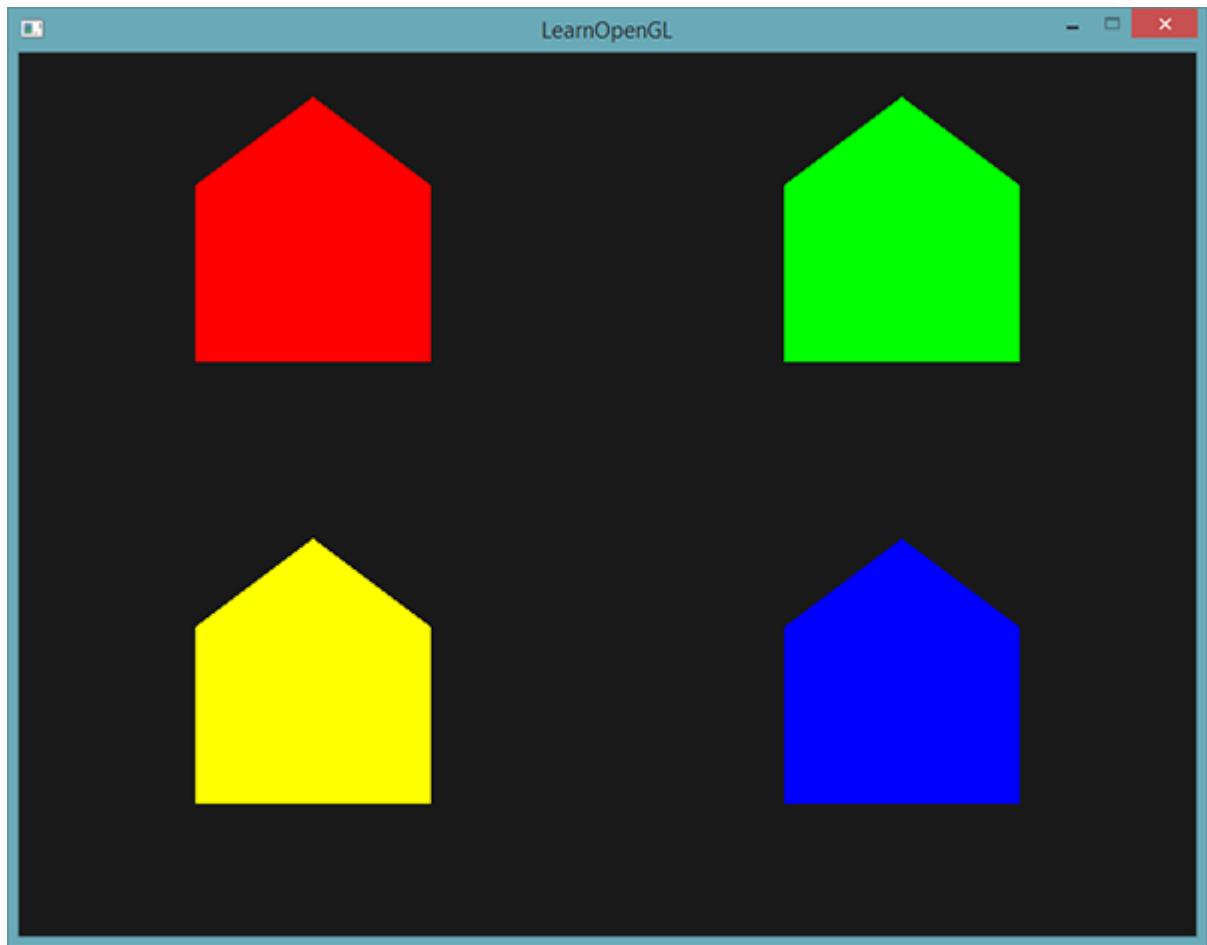
```
gl_Position = position + vec4(0.0f, 0.4f, 0.0f, 0.0f); // 5:屋
```

顶

```
EmitVertex();
```

```
EndPrimitive();
```

所有发射出去的顶点都把最后储存在 **fColor** 中的值嵌入到他们的数据中，和我们在他们的属性中定义的顶点颜色相同。所有的分房子便都有了自己的颜色：



为了好玩儿，我们还可以假装这是在冬天，给最后一个顶点一个自己的白色，就像在屋顶上落了一些雪。

```
fColor = gs_in[0].color;

gl_Position = position + vec4(-0.2f, -0.2f, 0.0f, 0.0f);

EmitVertex();

gl_Position = position + vec4(0.2f, -0.2f, 0.0f, 0.0f);

EmitVertex();

gl_Position = position + vec4(-0.2f, 0.2f, 0.0f, 0.0f);

EmitVertex();

gl_Position = position + vec4(0.2f, 0.2f, 0.0f, 0.0f);
```

```
EmitVertex();

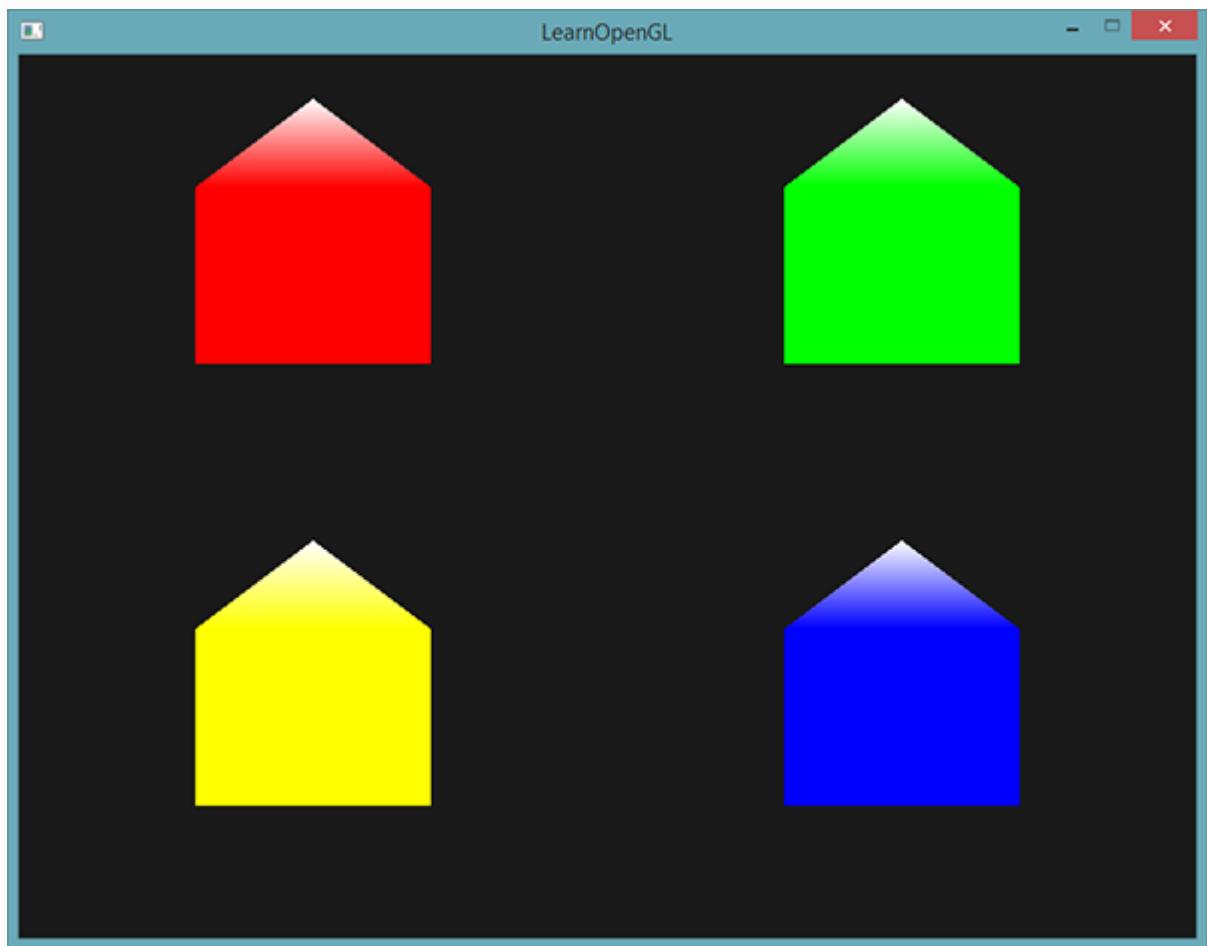
gl_Position = position + vec4(0.0f, 0.4f, 0.0f, 0.0f);

fColor = vec3(1.0f, 1.0f, 1.0f);

EmitVertex();

EndPrimitive();
```

结果就像这样:



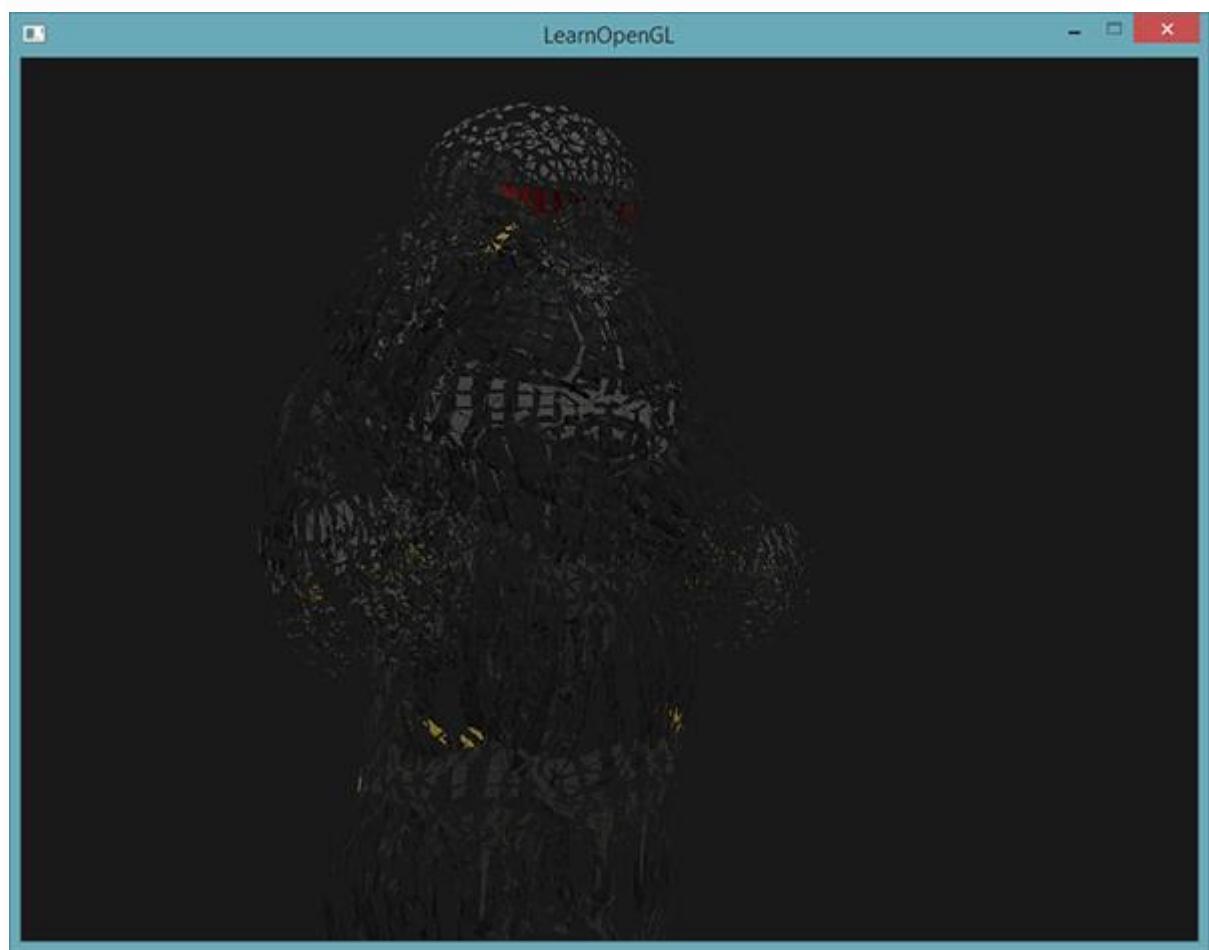
你可以对比一下你的[源码](#)和[着色器](#)。

你可以看到，使用几何着色器，你可以使用最简单的基本图形就能获得漂亮的新玩意。因为这些形状是在你的 GPU 超快硬件上动态生成的，这要比使用顶点缓冲自己定义这些形状更为高效。几何缓冲在简单的经常被重复的形状比如体素（voxel）的世界和室外的草地上，是一种非常强大的优化工具。

## 爆炸式物体

绘制房子的确很有趣，但我们不会经常这么用。这就是为什么现在我们将撬起物体缺口，形成爆炸式物体的原因！虽然这个我们也不会经常用到，但是它能向你展示一些几何着色器的强大之处。

当我们说对一个物体进行爆破的时候并不是说我们将要把之前的那堆顶点炸掉，但是我们打算把每个三角形沿着它们的法线向量移动一小段距离。效果是整个物体上的三角形看起来就像沿着它们的法线向量爆炸了一样。纳米服上的三角形的爆炸式效果看起来是这样的：



这样一个几何着色器效果的一大好处是，它可以用到任何物体上，无论它们多复杂。

因为我们打算沿着三角形的法线向量移动三角形的每个顶点，我们需要先计算它的法线向量。我们要做的是计算出一个向量，它垂直于三角形的表面，使用这三个我们已经的到的顶点就能做到。你可能记得变换教程中，我们可以使用叉乘获

取一个垂直于两个其他向量的向量。如果我们有两个向量 **a** 和 **b**，它们平行于三角形的表面，我们就可以对这两个向量进行叉乘得到法线向量了。下面的几何着色器函数做的正是这件事，它使用 3 个输入顶点坐标获取法线向量：

```
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向量 **a** 和 **b**，它们平行于三角形的表面。两个向量相减得到一个两个向量的差值，由于所有 3 个点都在三角形平面上，任何向量相减都会得到一个平行于平面的向量。一定要注意，如果我们调换了 **a** 和 **b** 的叉乘顺序，我们得到的法线向量就会使反的，顺序很重要！

知道了如何计算法线向量，我们就能创建一个 **explode** 函数，函数返回的是一个新向量，它把位置向量沿着法线向量方向平移：

```
vec4 explode(vec4 position, vec3 normal)
{
 float magnitude = 2.0f;
 vec3 direction = normal * ((sin(time) + 1.0f) / 2.0f) * magnitude;
 return position + vec4(direction, 0.0f);
}
```

函数本身并不复杂，**sin**（正弦）函数把一个 **time** 变量作为它的参数，它根据时间来返回一个-1.0 到 1.0 之间的值。因为我们不想让物体坍缩，所以我们把 **sin** 返回的值做成 0 到 1 的范围。最后的值去乘以法线向量，**direction** 向量被添加到位置向量上。

爆炸效果的完整的几何着色器是这样的，它使用我们的模型加载器，绘制出一个模型：

```
#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();

}
```

注意我们同样在发射一个顶点前输出了合适的纹理坐标。

也不要忘记在 OpenGL 代码中设置 `time` 变量：

```
glUniform1f(glGetUniformLocation(shader.Program, "time"),
 glfwGetTime());

...

}
```

最后的结果是一个随着时间持续不断地爆炸的 3D 模型（不断爆炸不断回到正常状

态）。尽管没什么大用处，它却向你展示出很多几何着色器的高级用法。你可以用 [完

整的源

码] ([http://LearnOpenGL.com/code\\_viewer.php?code=advanced/geometry\\_shader\\_explode](http://LearnOpenGL.com/code_viewer.php?code=advanced/geometry_shader_explode)) 和 [着色

器] ([http://LearnOpenGL.com/code\\_viewer.php?code=advanced/geometry\\_shader\\_explode\\_shaders](http://LearnOpenGL.com/code_viewer.php?code=advanced/geometry_shader_explode_shaders)) 对比一下你自己的。

#### 把法线向量显示出来

在这部分我们将使用几何着色器写一个例子，非常有用：显示一个法线向量。当编写光照着色器的时候，你最终会遇到奇怪的视频输出问题，你很难决定是什么导致了这个问题。通常导致光照错误的是，不正确的加载顶点数据，以及给它们指定了不合理的顶点属性，又或是在着色器中不合理的管理，导致产生了不正确的法线向量。我们所希望的是有某种方式可以检测出法线向量是否正确。把法线向量显示出来正是这样一种方法，恰好几何着色器能够完美地达成这个目的。

思路是这样的：我们先不用几何着色器，正常绘制场景，然后我们再次绘制一遍场景，但这次只显示我们通过几何着色器生成的法线向量。几何着色器把一个三角形基本图形作为输入类型，用它们生成 3 条和法线向量同向的线段，每个顶点一条。伪代码应该是这样的：

```
```c++
shader.Use();
DrawScene();
normalDisplayShader.Use();
DrawScene();
```

这次我们会创建一个使用模型提供的顶点法线，而不是自己去生成。为了适应缩放和旋转我们在把它变换到裁切空间坐标前，使用法线矩阵来法线（几何着色器用他的位置向量做为裁切空间坐标，所以我们还要把法线向量变换到同一个空间）。这些都能在顶点着色器中完成：

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
```

```

out VS_OUT {
    vec3 normal;
} vs_out;

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

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = normalize(vec3(projection * vec4(normalMatrix *
normal, 1.0)));
}

```

经过变换的裁切空间法线向量接着通过一个接口块被传递到下个着色阶段。几何着色器接收每个顶点（带有位置和法线向量），从每个位置向量绘制出一个法线向量：

```

#version 330 core

layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
    vec3 normal;
}

```

```
    } gs_in[];
```

```
const float MAGNITUDE = 0.4f;
```

```
void GenerateLine(int index)
```

```
{
```

```
    gl_Position = gl_in[index].gl_Position;
```

```
    EmitVertex();
```

```
    gl_Position = gl_in[index].gl_Position +
```

```
    vec4(gs_in[index].normal, 0.0f) * MAGNITUDE;
```

```
    EmitVertex();
```

```
    EndPrimitive();
```

```
}
```

```
void main()
```

```
{
```

```
    GenerateLine(0); // First vertex normal
```

```
    GenerateLine(1); // Second vertex normal
```

```
    GenerateLine(2); // Third vertex normal
```

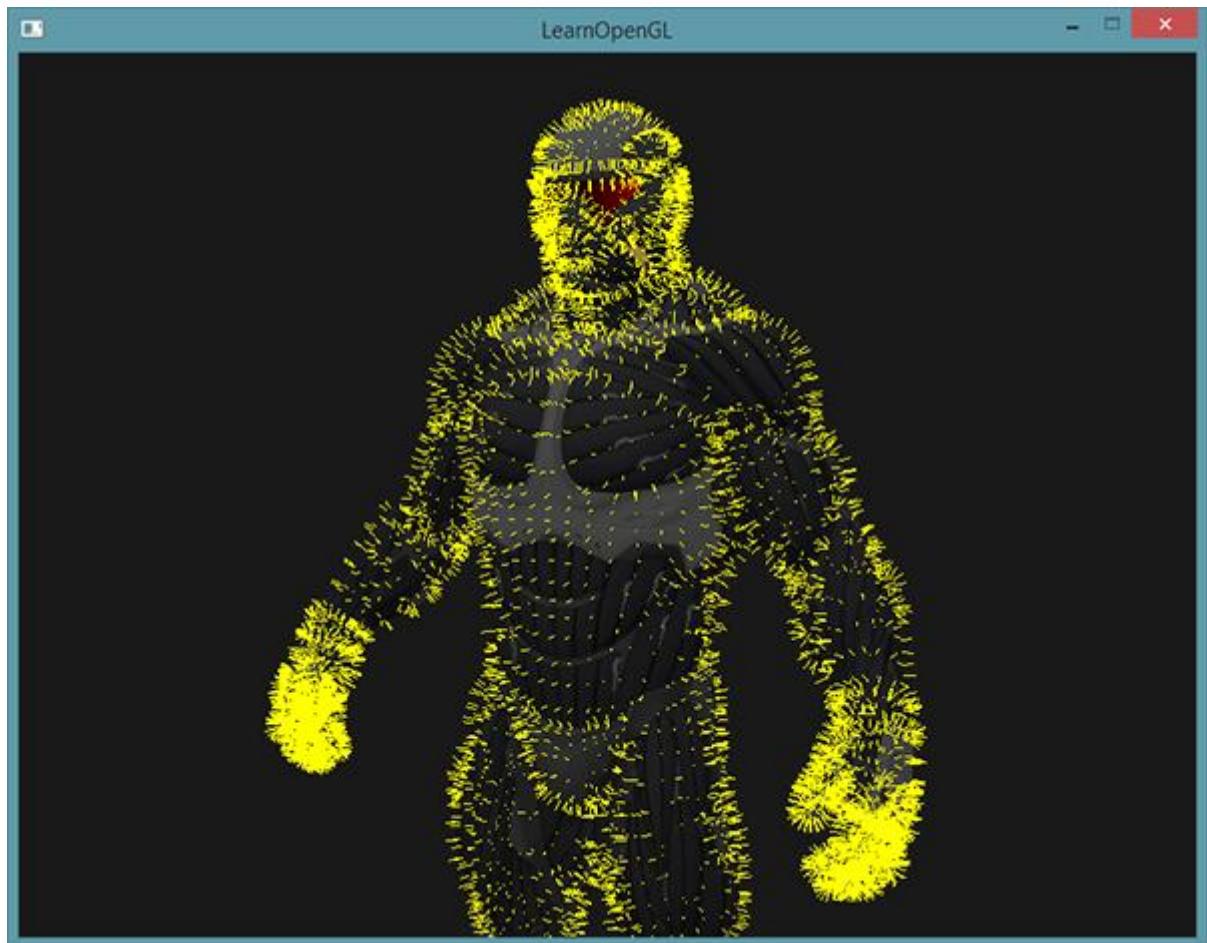
```
}
```

到现在为止，像这样的几何着色器的内容就不言自明了。需要注意的是我们我们把法线向量乘以一个 MAGNITUDE 向量来限制显示出的法线向量的大小（否则它们就太大了）。

由于把法线显示出来通常用于调试的目的，我们可以在片段着色器的帮助下把它们显示为单色的线（如果你愿意也可以更炫一点）。

```
#version 330 core  
  
out vec4 color;  
  
void main()  
{  
    color = vec4(1.0f, 1.0f, 0.0f, 1.0f);  
}
```

现在先使用普通着色器来渲染你的模型，然后使用特制的法线可视着色器，你会看到这样的效果：



除了我们的纳米服现在看起来有点像一个带着隔热手套的全身多毛的家伙外，它给了我们一种非常有效的检查一个模型的法线向量是否有错误的方式。你可以想象下这样的几何着色器也经常能被用在给物体添加毛发上。

你可以从这里找到[源码](#)和可显示法线的[着色器](#)。

实例化(Instancing)

原文	Instancing
作者	JoeyDeVries
翻译	Django
校对	Geequlim

假如你有一个有许多模型的场景，而这些模型的顶点数据都一样，只是进行了不同的世界空间的变换。想象一下，有一个场景中充满了草叶：每根草都是几个三角形组成的。你可能需要绘制很多的草叶，最终一次渲染循环中就肯能有成千上万个草需要绘制了。因为每个草叶只是由几个三角形组成，绘制一个几乎是即刻完成，但是数量巨大以后，执行起来就很慢了。

如果我们渲染这样多的物体的时候，也许代码会写成这样：

```
for(GLuint i = 0; i < amount_of_models_to_draw; i++)  
{  
    DoSomePreparations(); //在这里绑定VAO、绑定纹理、设置uniform变量  
    等  
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);  
}
```

像这样绘制出你模型的其他实例，多次绘制之后，很快将达到一个瓶颈，这是因为你 `glDrawArrays` 或 `glDrawElements` 这样的函数(Draw call)过多。这样渲染顶点数据，会明显降低执行效率，这是因为 OpenGL 在它可以绘制你的顶点数据之前必须做一些准备工作(比如告诉 GPU 从哪个缓冲读取数据，以及在哪里找到顶点属性，所有这些都会使 CPU 到 GPU 的总线变慢)。所以即使渲染顶点超快，而多次给你的 GPU 下达这样的渲染命令却未必。

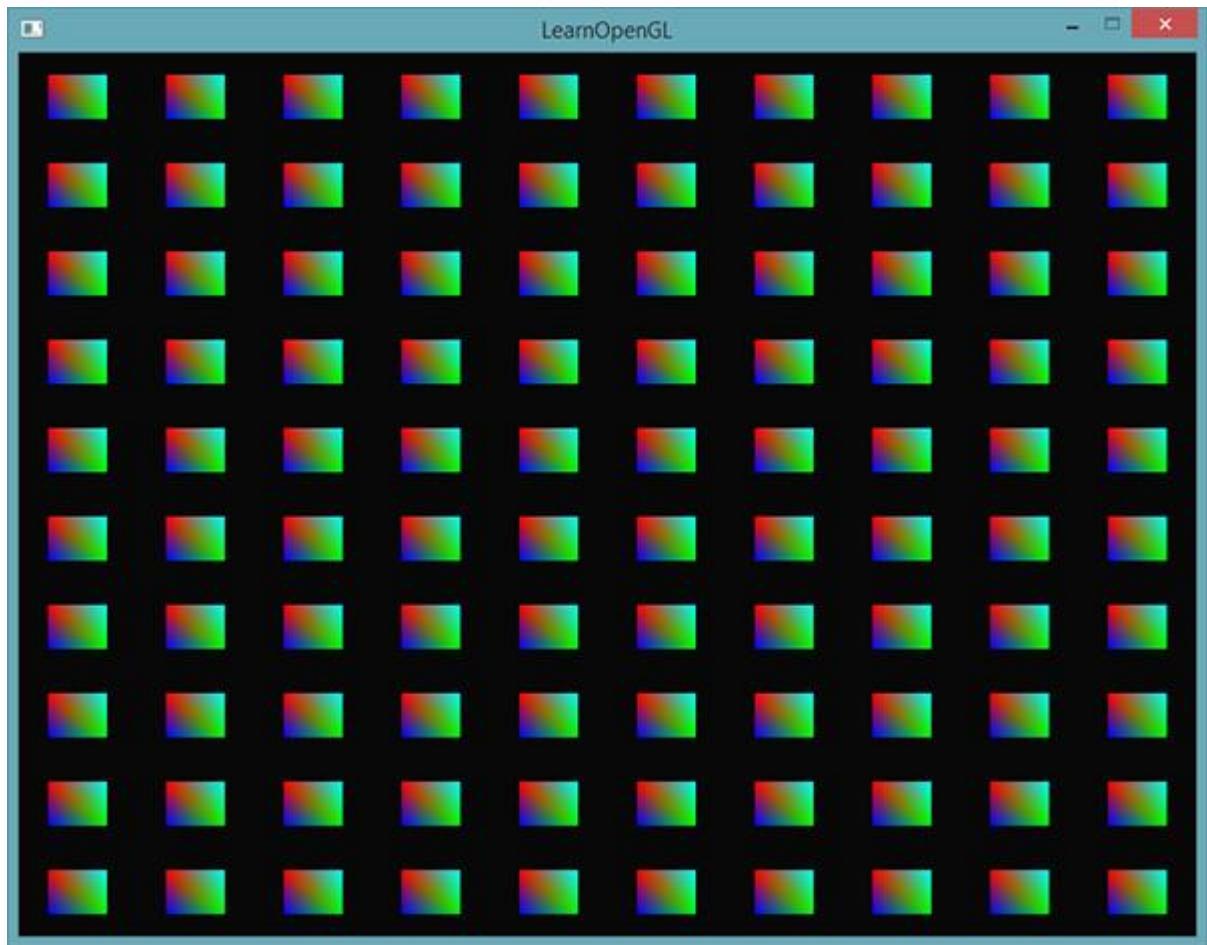
如果我们能够将数据一次发送给 GPU，就会更方便，然后告诉 OpenGL 使用一个绘制函数，将这些数据绘制为多个物体。这就是我们将要展开讨论的**实例化(instancing)**。

实例化(instancing)是一种只调用一次渲染函数却能绘制出很多物体的技术，它节省渲染物体时从 CPU 到 GPU 的通信时间，而且只需做一次即可。要使用实例化渲染，我们必须将 `glDrawArrays` 和 `glDrawElements` 各自改为 `glDrawArraysInstanced` 和 `glDrawElementsInstanced`。这些用于实例化的函数版本需要设置一个额外的参数，叫做**实例数量(instance count)**，它设置我们打算渲染实例的数量。这样我们就只需要把所有需要的数据发送给 GPU 一次就行了，然后告诉 GPU 它该如何使用一个函数来绘制所有这些实例。

就其本身而言，这个函数用处不大。渲染同一个物体一千次对我们来说没用，因为每个渲染出的物体不仅相同而且还在同一个位置；我们只能看到一个物体！出于这个原因 GLSL 在着色器中嵌入了另一个内建变量，叫做 `gl_InstanceID`。

在通过实例化绘制时，`gl_InstanceID` 的初值是 0，它在每个实例渲染时都会增加 1。如果我们渲染 43 个实例，那么在顶点着色器 `gl_InstanceID` 的值最后就是 42。每个实例都拥有唯一的值意味着我们可以索引到一个位置数组，并将每个实例摆放在世界空间的不同的位置上。

我们调用一个实例化渲染函数，在标准化设备坐标中绘制一百个 2D 四边形来看看实例化绘制的效果是怎样的。通过对一个储存着 100 个偏移量向量的索引，我们为每个实例四边形添加一个偏移量。最后，窗口被排列精美的四边形网格填满：



每个四边形是 2 个三角形所组成的，因此总共有 6 个顶点。每个顶点包含一个 2D 标准设备坐标位置向量和一个颜色向量。下面是例子中所使用的顶点数据，每个三角形为了适应屏幕都很小：

```
GLfloat 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  
};
```

片段着色器接收从顶点着色器发送来的颜色向量，设置为它的颜色输出，从而为四边形上色：

```
#version 330 core  
  
in vec3 fColor;  
  
out vec4 color;
```

```
void main()  
{  
    color = vec4(fColor, 1.0f);  
}
```

到目前为止没有什么新内容，但顶点着色器开始变得有意思了：

```
#version 330 core  
  
layout (location = 0) in vec2 position;  
layout (location = 1) in vec3 color;  
  
out vec3 fColor;  
  
uniform vec2 offsets[100];  
  
void main()  
{  
    vec2 offset = offsets[gl_InstanceID];
```

```
gl_Position = vec4(position + offset, 0.0f, 1.0f);  
  
fColor = color;  
  
}
```

这里我们定义了一个 `uniform` 数组，叫 `offsets`，它包含 100 个偏移量向量。在顶点着色器里，我们接收一个对应着当前实例的偏移量，这是通过使用 `gl_InstanceID` 来索引 `offsets` 得到的。如果我们使用实例化绘制 100 个四边形，使用这个顶点着色器，我们就能得到 100 位于不同位置的四边形。

我们一定要设置偏移位置，在游戏循环之前我们用一个嵌套 `for` 循环计算出它来：

```
glm::vec2 translations[100];  
  
int index = 0;  
  
GLfloat offset = 0.1f;  
  
for(GLint y = -10; y < 10; y += 2)  
  
{  
  
    for(GLint x = -10; x < 10; x += 2)  
  
    {  
  
        glm::vec2 translation;  
  
        translation.x = (GLfloat)x / 10.0f + offset;  
  
        translation.y = (GLfloat)y / 10.0f + offset;  
  
        translations[index++] = translation;  
  
    }  
  
}
```

这里我们创建 100 个平移向量，它包含着 10×10 格子所有位置。除了生成 `translations` 数组外，我们还需要把这些位移数据发送到顶点着色器的 `uniform` 数组：

```

        shader.Use();

        for(GLuint i = 0; i < 100; i++)
    {
        stringstream ss;
        string index;
        ss << i;
        index = ss.str();

        GLint location = glGetUniformLocation(shader.Program,
            ("offsets[" + index + "]").c_str());
        glUniform2f(location, translations[i].x, translations[i].y);
    }
}

```

这一小段代码中，我们将 `for` 循环计数器 `i` 变为 `string`，接着就能动态创建一个为请求的 `uniform` 的 `location` 创建一个 `location` 字符串。将 `offsets` 数组中的每个条目，我们都设置为相应的平移向量。

现在所有的准备工作都结束了，我们可以开始渲染四边形了。用实例化渲染来绘制四边形，我们需要调用 `glDrawArraysInstanced` 或 `glDrawElementsInstanced`，由于我们使用的不是索引绘制缓冲，所以我们用的是 `glDrawArrays` 对应的那个版本 `glDrawArraysInstanced`：

```

glBindVertexArray(quadVAO);

glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);

glBindVertexArray(0);

```

`glDrawArraysInstanced` 的参数和 `glDrawArrays` 一样，除了最后一个参数设置了我们打算绘制实例的数量。我们想展示 100 个四边形，它们以 10×10 网格形式展现，所以这儿就是 100。运行代码，你会得到 100 个相似的有色四边形。

实例化数组(instanced arrays)

在这种特定条件下，前面的实现很好，但是当我们有 100 个实例的时候（这很正常），最终我们将碰到 `uniform` 数据数量的上线。为避免这个问题另一个可替代方案是实例化数组，它使用顶点属性来定义，这样就允许我们使用更多的数据了，当顶点着色器渲染一个新实例时它才会被更新。

使用顶点属性，每次运行顶点着色器都将让 `GLSL` 获取到下个顶点属性集合，它们属于当前顶点。当把顶点属性定义为实例数组时，顶点着色器只更新每个实例的顶点属性的内容而不是顶点的内容。这使我们在每个顶点数据上使用标准顶点属性，用实例数组来储存唯一的实例数据。

为了展示一个实例化数组的例子，我们将采用前面的例子，把偏移 `uniform` 表示为一个实例数组。我们不得不增加另一个顶点属性，来更新顶点着色器。

```
#version 330 core

layout (location = 0) in vec2 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 offset;

out vec3 fColor;

void main()
{
    gl_Position = vec4(position + offset, 0.0f, 1.0f);
    fColor = color;
}
```

我们不再使用 `gl_InstanceID`，可以直接用 `offset` 属性，不用先在一个大 `uniform` 数组里进行索引。

因为一个实例化数组实际上就是一个和位置和颜色一样的顶点属性，我们还需要把它的内容储存为一个顶点缓冲对象里，并把它配置为一个属性指针。我们首先将平移变换数组贮存到一个新的缓冲对象上：

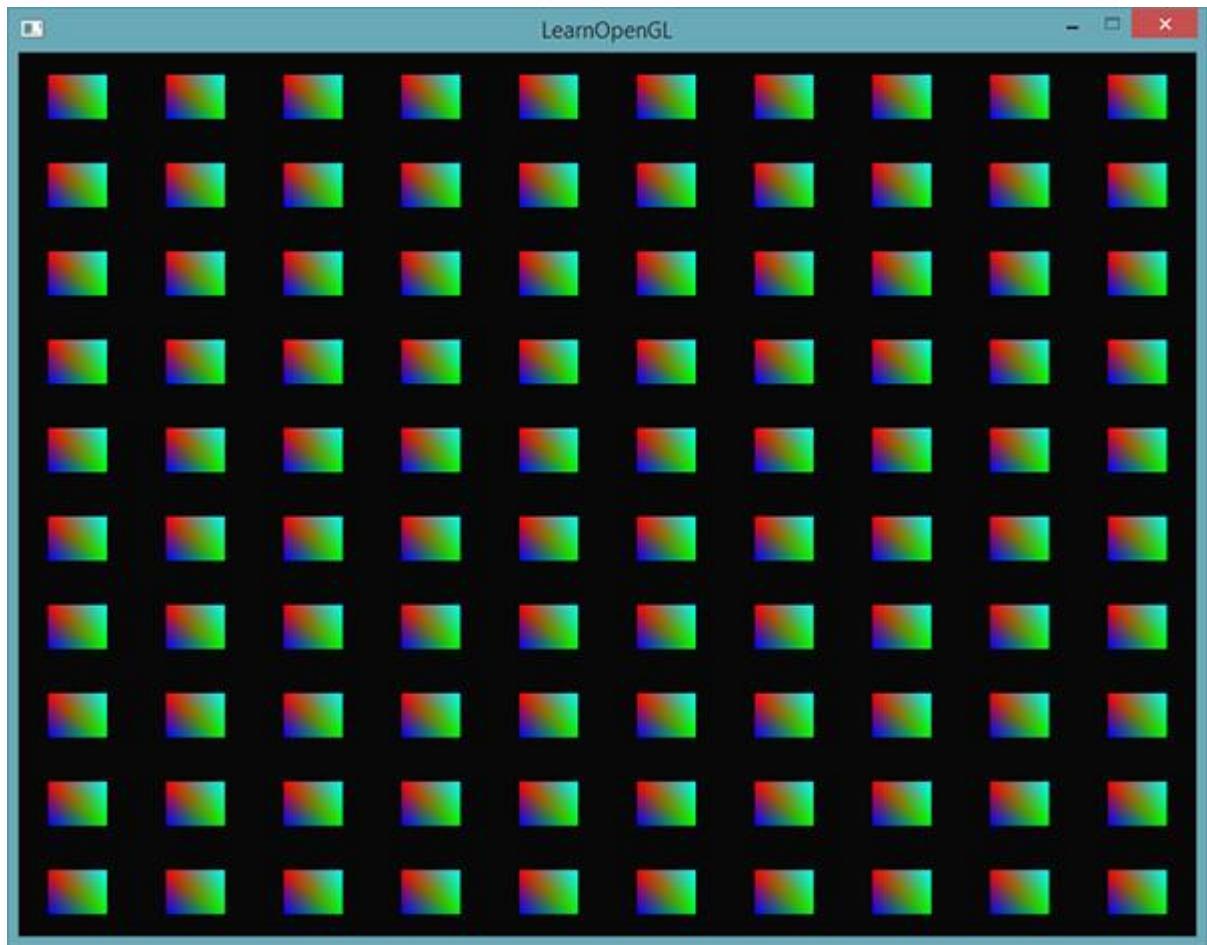
```
GLuint instanceVBO;  
  
glGenBuffers(1, &instanceVBO);  
  
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);  
  
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec2) * 100,  
&translations[0], GL_STATIC_DRAW);  
  
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

之后我们还需要设置它的顶点属性指针，并开启顶点属性：

```
 glEnableVertexAttribArray(2);  
  
 glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);  
  
 glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat),  
 (GLvoid*)0);  
  
 glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
 glVertexAttribDivisor(2, 1);
```

代码中有意思的地方是，最后一行，我们调用了 `glVertexAttribDivisor`。这个函数告诉 OpenGL 什么时候去更新顶点属性的内容到下个元素。它的第一个参数是提到的顶点属性，第二个参数是属性除数（attribute divisor）。默认属性除数是 0，告诉 OpenGL 在顶点着色器的每次迭代更新顶点属性的内容。把这个属性设置为 1，我们就是告诉 OpenGL 我们打算在开始渲染一个新的实例的时候更新顶点属性的内容。设置为 2 代表我们每 2 个实例更新内容，依此类推。把属性除数设置为 1，我们可以高效地告诉 OpenGL，location 是 2 的顶点属性是一个实例数组（instanced array）。

如果我们现在再次使用 `glDrawArraysInstanced` 渲染四边形，我们会得到下面的输出：

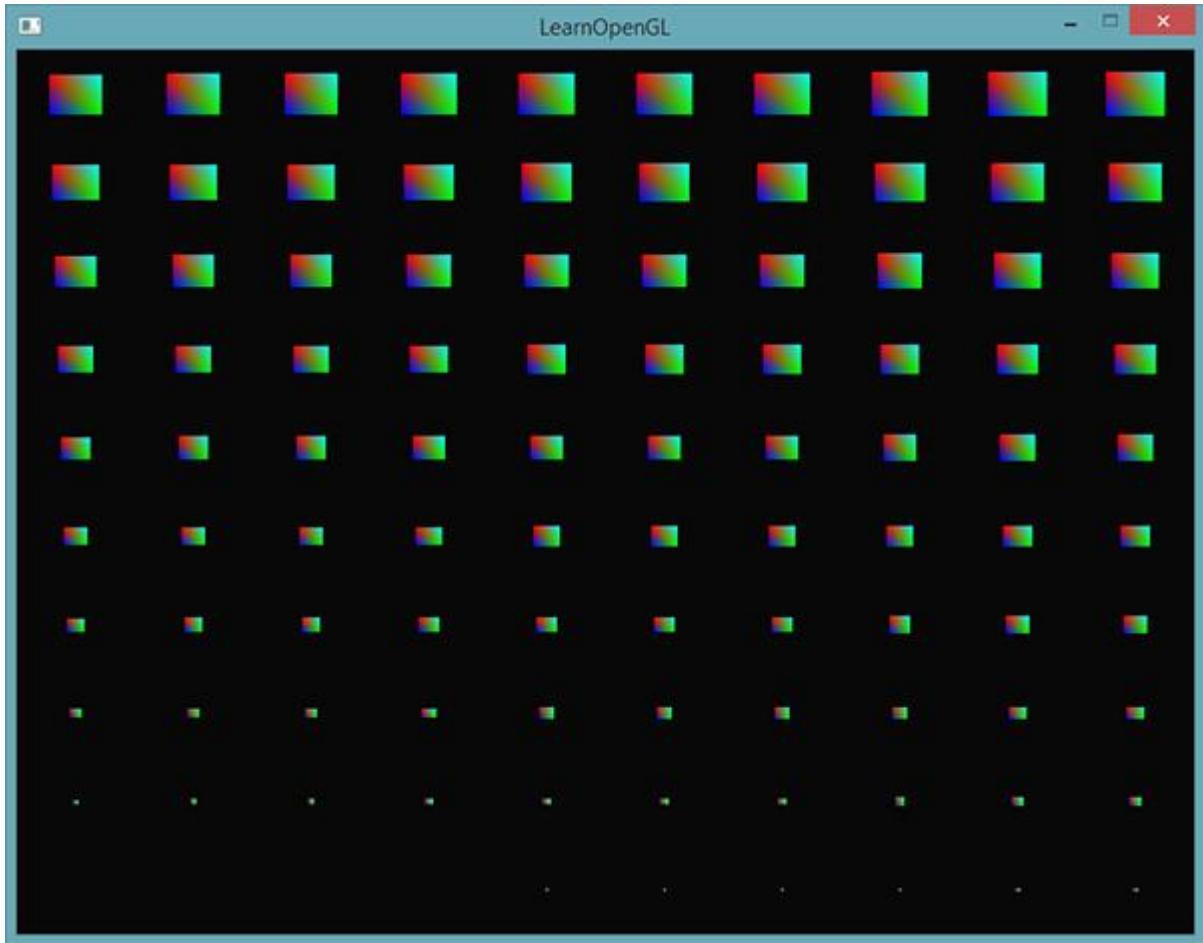


和前面的一样，但这次是使用实例数组实现的，它使我们为绘制实例向顶点着色器传递更多的数据（内存允许我们存多少就能存多少）。

你还可以使用 `gl_InstanceID` 从右上向左下缩小每个四边形。

```
void main()
{
    vec2 pos = position * (gl_InstanceID / 100.0f);
    gl_Position = vec4(pos + offset, 0.0f, 1.0f);
    fColor = color;
}
```

结果是第一个实例的四边形被绘制的非常小，随着绘制实例的增加，`gl_InstanceID`越来越接近 100，这样更多的四边形会更接近它们原来的大小。这是一种很好的将 `gl_InstanceID` 与实例数组结合使用的法则：



如果你仍然不确定实例渲染如何工作，或者想看看上面的代码是如何组合起来的，你可以在[这里找到应用的源码](#)。

这些例子不是实例的好例子，不过挺有意思的。它们可以让你对实例的工作方式有一个概括的理解，但是当绘制拥有极大数量的相同物体的时候，它极其有用，现在我们还没有展示呢。出于这个原因，我们将在接下来的部分进入太空来看看实例渲染的威力。

小行星带

想象一下，在一个场景中有一个很大的行星，行星周围有一圈小行星带。这样一个小行星大可能包含成千上万的石块，对于大多数显卡来说几乎是难以完成的渲染任务。这个场景对于实例渲染来说却不再话下，由于所有小行星可以使用一个

模型来表示。每个小行星使用一个变换矩阵就是一个经过少量变化的独一无二的小行星了。

为了展示实例渲染的影响，我们先不使用实例渲染，来渲染一个小行星围绕行星飞行的场景。这个场景的大天体可以[从这里下载](#)，此外要把小行星放在合适的位置上。小行星可以[从这里下载](#)。

为了得到我们理想中的效果，我们将为每个小行星生成一个变换矩阵，作为它们的模型矩阵。变换矩阵先将小行星平移到行星带上，我们还要添加一个随机位移值来作为偏移量，这样才能使行星带更自然。接着我们应用一个随机缩放，以及一个随机旋转向量。最后，变换矩阵就会将小行星变换到行星的周围，同时使它们更自然，每个行星都有别于其他的。

```
GLuint amount = 1000;

glm::mat4* modelMatrices;

modelMatrices = new glm::mat4[amount];

srand(glfwGetTime()); // initialize random seed

GLfloat radius = 50.0;

GLfloat offset = 2.5f;

for(GLuint i = 0; i < amount; i++)

{

    glm::mat4 model;

    // 1. Translation: displace along circle with 'radius' in range

    [-offset, offset]

    GLfloat angle = (GLfloat)i / (GLfloat)amount * 360.0f;

    GLfloat displacement = (rand() % (GLint)(2 * offset * 100)) / 100.0f

    - offset;
```

```

GLfloat x = sin(angle) * radius + displacement;

displacement = (rand() % (GLint)(2 * offset * 100)) / 100.0f -
offset;

GLfloat y = displacement * 0.4f; // y value has smaller displacement

displacement = (rand() % (GLint)(2 * offset * 100)) / 100.0f -
offset;

GLfloat z = cos(angle) * radius + displacement;

model = glm::translate(model, glm::vec3(x, y, z));

// 2. Scale: Scale between 0.05 and 0.25f

GLfloat scale = (rand() % 20) / 100.0f + 0.05;

model = glm::scale(model, glm::vec3(scale));

// 3. Rotation: add random rotation around a (semi)randomly picked
rotation axis vector

GLfloat rotAngle = (rand() % 360);

model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

// 4. Now add to list of matrices

modelMatrices[i] = model;
}

```

这段代码看起来还是有点吓人，但我们基本上是沿着一个半径为 `radius` 的圆圈变换小行星的 `x` 和 `y` 的值，让每个小行星在 `-offset` 和 `offset` 之间随机生成一个位置。我们让 `y` 变化的更小，这让这个环带就会成为扁平的。接着我们缩放和旋转变换，把结果储存到一个 `modelMatrices` 矩阵里，它的大小是 `amount`。这里我们生成 1000 个模型矩阵，每个小行星一个。

加载完天体和小行星模型后，编译着色器，渲染代码是这样的：

```
// 绘制行星

shader.Use();

glm::mat4 model;

model = glm::translate(model, glm::vec3(0.0f, -5.0f, 0.0f));

model = glm::scale(model, glm::vec3(4.0f, 4.0f, 4.0f));

glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

planet.Draw(shader);
```

```
// 绘制石头

for(GLuint i = 0; i < amount; i++)

{

    glUniformMatrix4fv(modelLoc, 1, GL_FALSE,

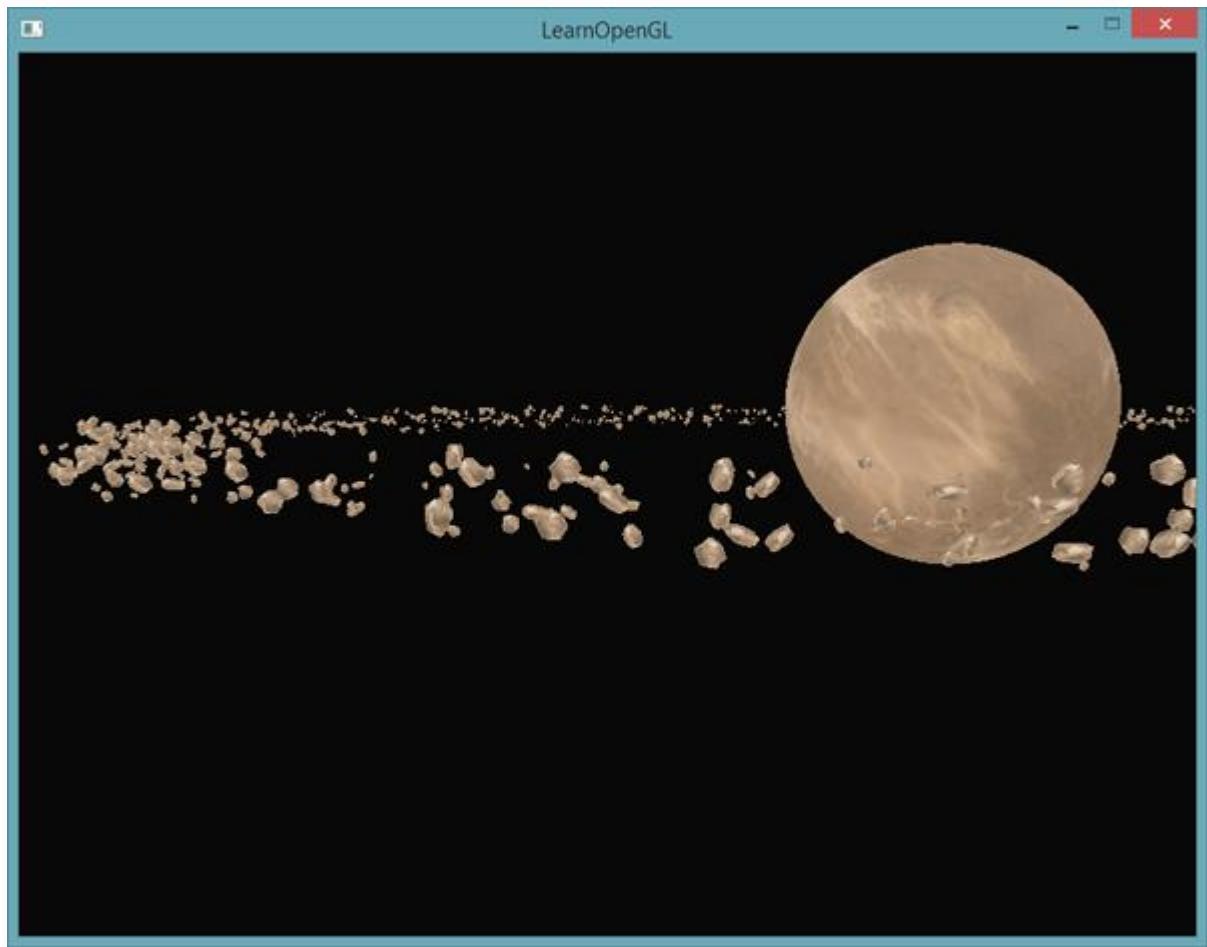
    glm::value_ptr(modelMatrices[i]));

    rock.Draw(shader);

}
```

我们先绘制天体模型，要把它平移和缩放一点以适应场景，接着，我们绘制 `amount` 数量的小行星，它们按照我们所计算的结果进行变换。在我们绘制每个小行星之前，我们还得先在着色器中设置相应的模型变换矩阵。

结果是一个太空样子的场景，我们可以看到有一个自然的小行星带：



这个场景包含 1001 次渲染函数调用，每帧渲染 1000 个小行星模型。你可以在这里找到[场景的源码](#)，以及[顶点](#)和[片段](#)着色器。

当我们开始增加数量的时候，很快就会注意到帧数的下降，而且下降的厉害。当我们设置为 2000 的时候，场景慢得已经难以移动了。

我们再次使用实例渲染来渲染同样的场景。我们先得对顶点着色器进行一点修改：

```
#version 330 core  
  
layout (location = 0) in vec3 position;  
  
layout (location = 2) in vec2 texCoords;  
  
layout (location = 3) in mat4 instanceMatrix;
```

```
out vec2 TexCoords;  
  
uniform mat4 projection;  
  
uniform mat4 view;  
  
void main()  
{  
    gl_Position = projection * view * instanceMatrix * vec4(position,  
1.0f);  
    TexCoords = texCoords;  
}
```

我们不再使用模型 `uniform` 变量，取而代之的是把一个 `mat4` 的顶点属性，送一我们可以将变换矩阵储存为一个实例数组（`instanced array`）。然而，当我们声明一个数据类型为顶点属性的时候，它比一个 `vec4` 更大，是有些不同的。顶点属性被允许的最大数据量和 `vec4` 相等。因为一个 `mat4` 大致和 4 个 `vec4` 相等，我们为特定的矩阵必须保留 4 个顶点属性。因为我们将它的位置赋值为 3 个列的矩阵，顶点属性的位置就会是 3、4、5 和 6。

然后我们必须为这 4 个顶点属性设置属性指针，并将其配置为实例数组：

```
for(GLuint i = 0; i < rock.meshes.size(); i++)  
{  
    GLuint VAO = rock.meshes[i].VAO;  
    // Vertex Buffer Object  
    GLuint buffer;  
    glBindVertexArray(VAO);  
    glGenBuffers(1, &buffer);
```

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);

glBufferData(GL_ARRAY_BUFFER, amount * sizeof(glm::mat4),
&modelMatrices[0], GL_STATIC_DRAW);

// Vertex Attributes

GLsizei vec4Size = sizeof(glm::vec4);

glEnableVertexAttribArray(3);

glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size,
(GLvoid*)0);

glEnableVertexAttribArray(4);

glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size,
(GLvoid*)(vec4Size));

glEnableVertexAttribArray(5);

glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size,
(GLvoid*)(2 * vec4Size));

glEnableVertexAttribArray(6);

glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size,
(GLvoid*)(3 * vec4Size));

glVertexAttribDivisor(3, 1);

glVertexAttribDivisor(4, 1);

glVertexAttribDivisor(5, 1);

glVertexAttribDivisor(6, 1);
```

```
glBindVertexArray(0);
```

```
}
```

要注意的是我们将 **Mesh** 的 **VAO** 变量声明为一个 **public** (公有) 变量，而不是一个 **private** (私有) 变量，所以我们可以获取它的顶点数组对象。这不是最干净的方案，但这能较好的适应本教程。若没有这点 **hack**，代码就干净了。我们声明了 **OpenGL** 该如何为每个矩阵的顶点属性的缓冲进行解释，每个顶点属性都是一个实例数组。

下一步我们再次获得网格的 **VAO**，这次使用 **glDrawElementsInstanced** 进行绘制：

```
// Draw meteorites

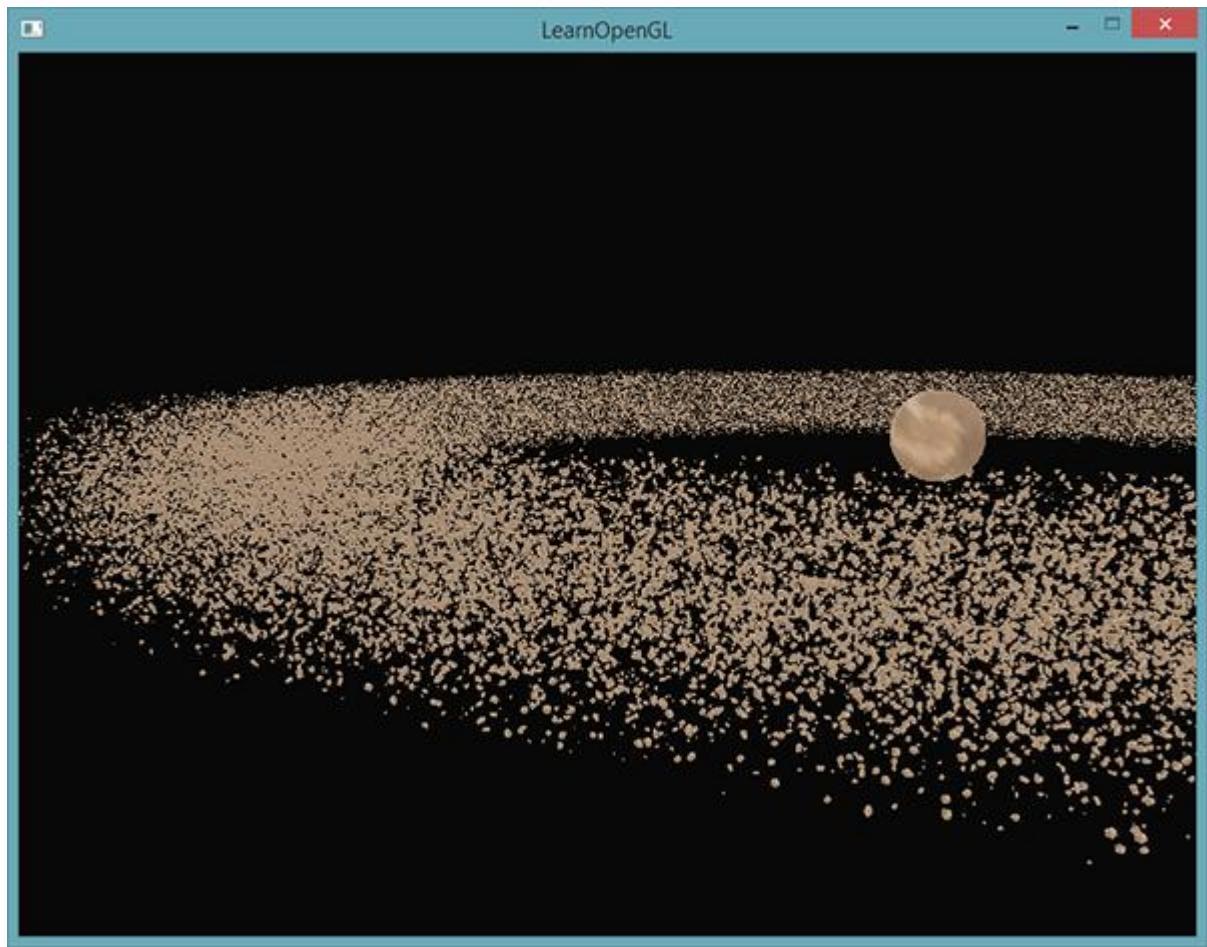
instanceShader.Use();

for(GLuint i = 0; i < rock.meshes.size(); i++)
{
    glBindVertexArray(rock.meshes[i].VAO);

    glDrawElementsInstanced(
        GL_TRIANGLES, rock.meshes[i].vertices.size(),
        GL_UNSIGNED_INT, 0, amount
    );

    glBindVertexArray(0);
}
```

这里我们绘制和前面的例子里一样数量 (**amount**) 的小行星，只不过是使用的实例渲染。结果是相似的，但你会看在开始增加数量以后效果的不同。不实例渲染，我们可以流畅渲染 1000 到 1500 个小行星。而使用了实例渲染，我们可以设置为 100000，每个模型由 576 个顶点，这几乎有 5 千 7 百万个顶点，而且帧率没有丝毫下降！



上图渲染了十万小行星，半径为 `150.0f`，偏移等于 `25.0f`。你可以在这里找到这个演示实例渲染的[源码](#)。

Important

有些机器渲染十万可能会有点吃力，所以尝试修改这个数量知道你能获得可以接受的帧率。

就像你所看到的，在合适的条件下，实例渲染对于你的显卡来说和普通渲染有很大不同。处于这个理由，实例渲染通常用来渲染草、草丛、粒子以及像这样的场景，基本上来讲只要场景中有很多重复物体，使用实例渲染都会获得好处。

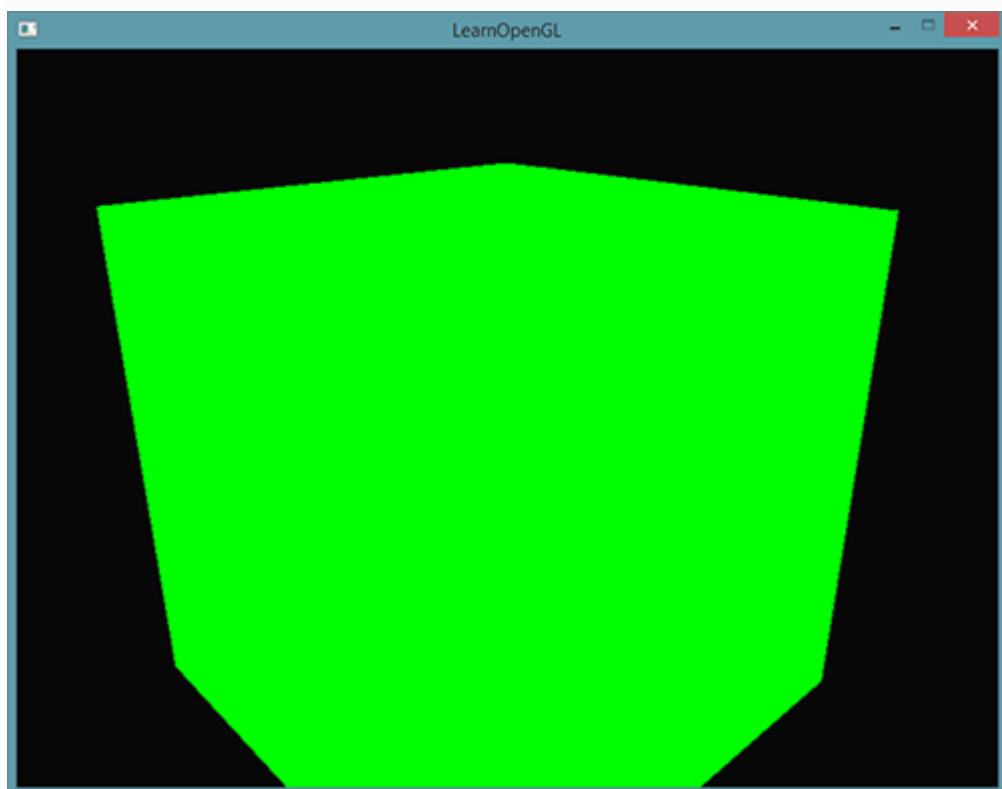
抗锯齿(Anti Aliasing)

原文 [Anti Aliasing](#)

作者 JoeyDeVries

原文	Anti Aliasing
翻译	Django
校对	Geequlim

在你的渲染大冒险中，你可能会遇到模型边缘有锯齿的问题。锯齿边出现的原因是由顶点数据像素化之后成为片段的方式所引起的。下面是一个简单的立方体，它体现了锯齿边的效果：



也许不是立即可见的，如果你更近的看看立方体的边，你就会发现锯齿了。如果我们放大就会看到下面的情境：



这当然不是我们在最终版本的应用里想要的效果。这个效果，很明显能看到边是由像素所构成的，这种现象叫做走样（aliasing）。有很多技术能够减少走样，产生更平滑的边缘，这些技术叫做反走样技术(anti-aliasing,也被称为抗锯齿技术)。

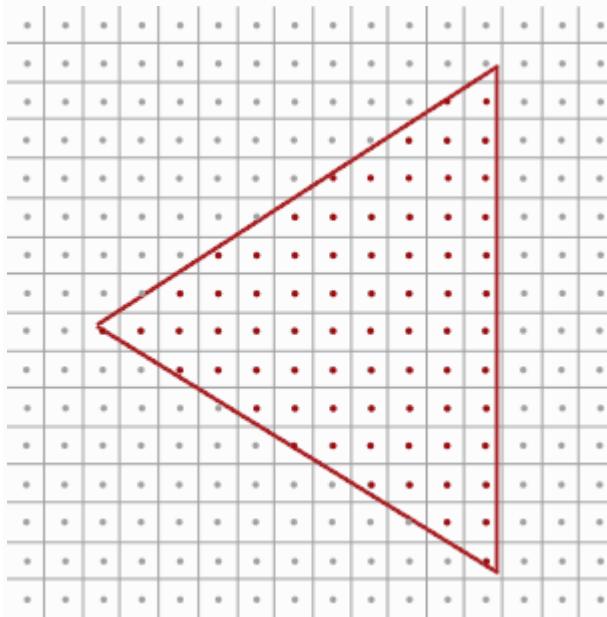
首先，我们有一个叫做超级采样抗锯齿技术(super sample anti-aliasing SSAA)，它暂时使用一个更高的解析度（以超级采样方式）来渲染场景，当视频输出在帧缓冲中被更新时，解析度便降回原来的普通解析度。这个额外的解析度被用来防止锯齿边。虽然它确实为我们提供了一种解决走样问题的方案，但却由于必须绘制比平时更多的片段而降低了性能。所以这个技术只流行了一段时间。

这个技术的基础上诞生了更为现代的技术，叫做多采样抗锯齿(multisample anti-aliasing)或叫MSAA，虽然它借用了SSAA的理念，但却以更加高效的方式实现了它。这节教程我们会展开讨论这个MSAA技术，它是OpenGL内建的。

多重采样(Multisampling)

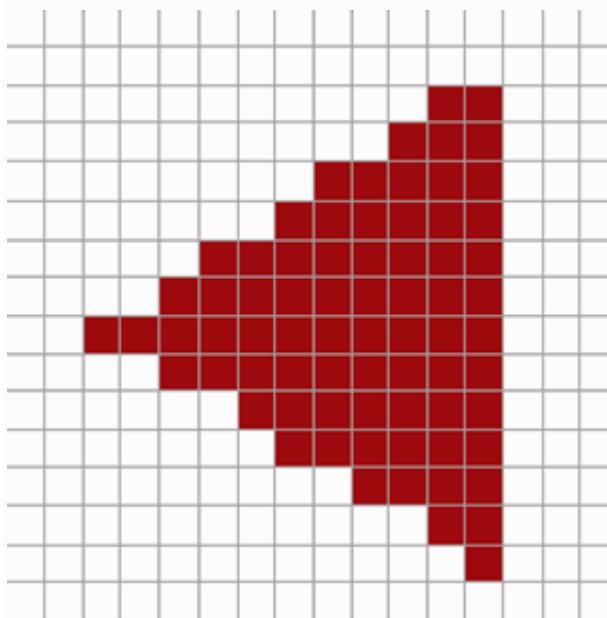
为了理解什么是多重采样，以及它是如何解决锯齿问题的，我们先要更深入了解一个OpenGL光栅化的工作方式。

光栅化是你的最终的经处理的顶点和片段着色器之间的所有算法和处理的集合。光栅化将属于一个基本图形的所有顶点转化为一系列片段。顶点坐标理论上可以含有任何坐标，但片段却不是这样，这是因为它们与你的窗口的解析度有关。几乎永远都不会有顶点坐标和片段的一对一映射，所以光栅化必须以某种方式决定每个特定顶点最终结束于哪个片段/屏幕坐标上。



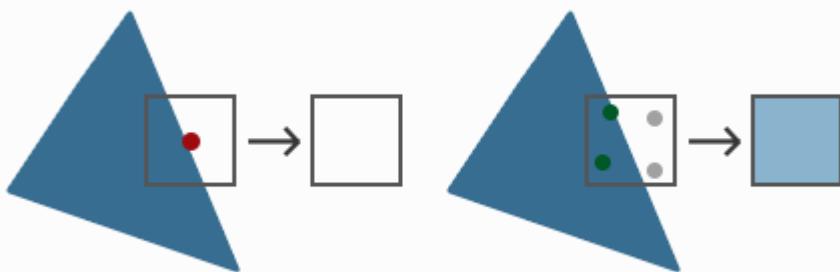
这里我们看到一个屏幕像素网格，每个像素中心包含一个采样点（sample point），它被用来决定一个像素是否被三角形所覆盖。红色的采样点如果被三角形覆盖，那么就会为这个被覆盖像素生成一个片段。即使三角形覆盖了部分屏幕像素，但是采样点没被覆盖，这个像素仍然不会受到任何片段着色器影响到。

你可能已经明白走样的原因来自何处了。三角形渲染后的版本最后在你的屏幕上是这样的：



由于屏幕像素总量的限制，有些边上的像素能被渲染出来，而有些则不会。结果就是我们渲染出的基本图形的非光滑边缘产生了上图的锯齿边。

多采样所做的正是不再使用单一采样点来决定三角形的覆盖范围，而是采用多个采样点。我们不再使用每个像素中心的采样点，取而代之的是 4 个子样本（**subsample**），用它们来决定像素的覆盖率。这意味着颜色缓冲的大小也由于每个像素的子样本的增加而增加了。



左侧的图显示了我们普通决定一个三角形的覆盖范围的方式。这个像素并不会运行一个片段着色器（这就仍保持空白），因为它的采样点没有被三角形所覆盖。右边的图展示了多采样的版本，每个像素包含 4 个采样点。这里我们可以看到只有 2 个采样点被三角形覆盖。

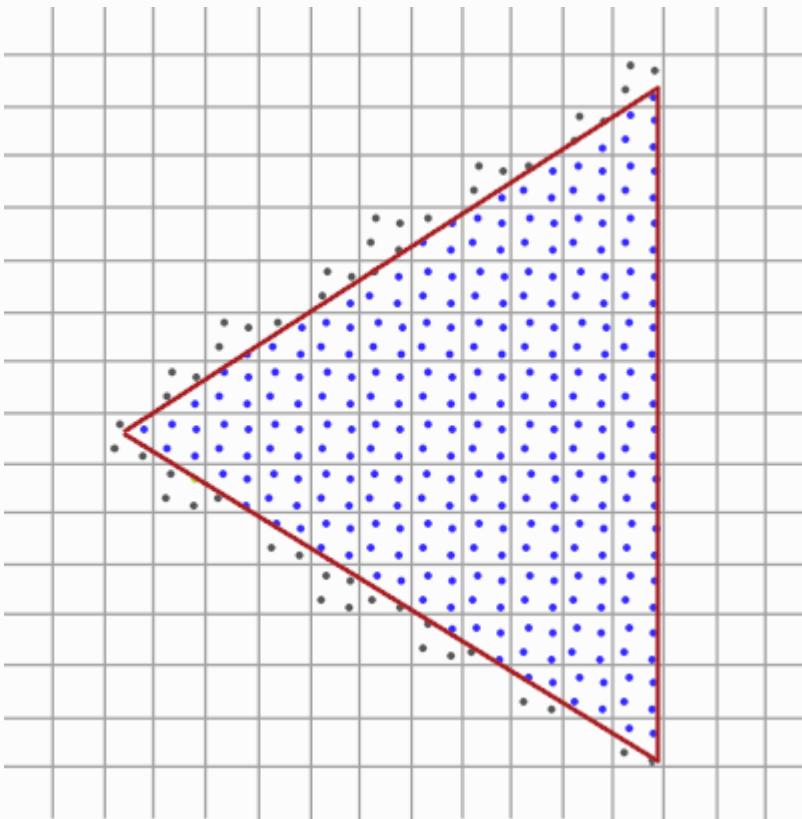
Important

采样点的数量是任意的，更多的采样点能带来更精确的覆盖率。

多采样开始变得有趣了。2 个子样本被三角覆盖，下一步是决定这个像素的颜色。我们原来猜测，我们会为每个被覆盖的子样本运行片段着色器，然后对每个像素的子样本的颜色进行平均化。例子的那种情况，我们在插值的顶点数据的每个子样本上运行片段着色器，然后将这些采样点的最终颜色储存起来。幸好，它不是这么运作的，因为这等于说我们必须运行更多的片段着色器，会明显降低性能。

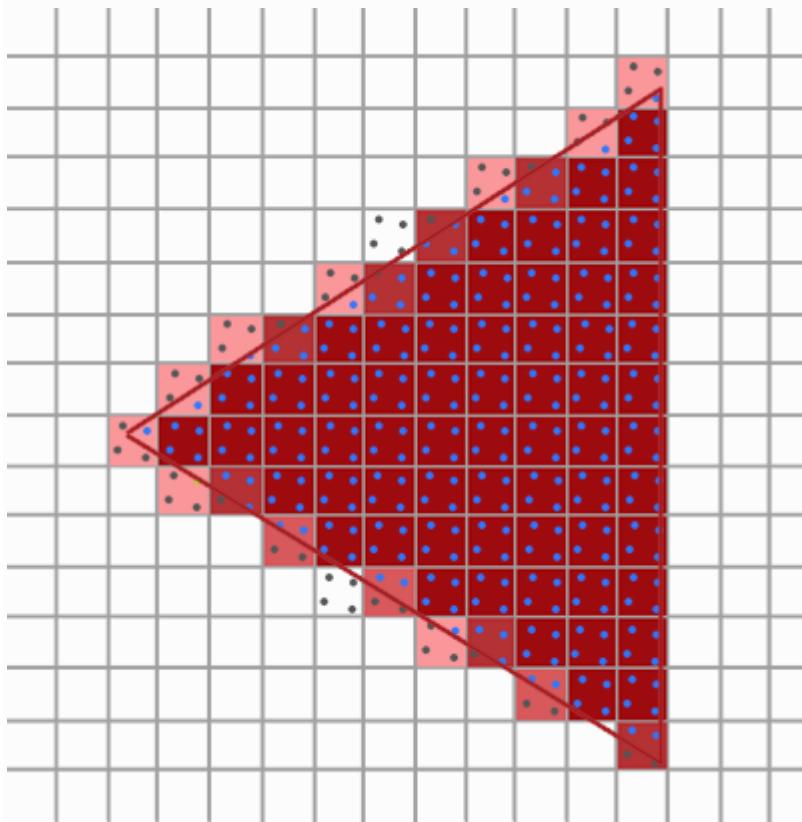
MSAA 的真正工作方式是，每个像素只运行一次片段着色器，无论多少子样本被三角形所覆盖。片段着色器运行着插值到像素中心的顶点数据，最后颜色被储存近每个被覆盖的子样本中，每个像素的所有颜色接着将平均化，每个像素最终有了一个唯一颜色。在前面的图片中 4 个样本中只有 2 个被覆盖，像素的颜色将以三角形的颜色进行平均化，颜色同时也被储存到其他 2 个采样点，最后生成的是一种浅蓝色。

结果是，颜色缓冲中所有基本图形的边都生成了更加平滑的样式。让我们看看当再次决定前面的三角形覆盖范围时多样本看起来是这样的：



这里每个像素包含着 **4** 个子样本（不相关的已被隐藏）蓝色的子样本是被三角形覆盖了的，灰色的没有被覆盖。三角形内部区域中的所有像素都会运行一次片段着色器，它输出的颜色被储存到所有 **4** 个子样本中。三角形的边缘并不是所有的子样本都会被覆盖，所以片段着色器的结果仅储存在部分子样本中。根据被覆盖子样本的数量，最终的像素颜色由三角形颜色和其他子样本所储存的颜色所决定。

大致上来说，如果更多的采样点被覆盖，那么像素的颜色就会更接近于三角形。如果我们用早期使用的三角形的颜色填充像素，我们会获得这样的结果：



对于每个像素来说，被三角形覆盖的子样本越少，像素受到三角形的颜色的影响也越少。现在三角形的硬边被比实际颜色浅一些的颜色所包围，因此观察者从远处看上去就比较平滑了。

不仅颜色值被多采样影响，深度和模板测试也同样使用了多采样点。比如深度测试，顶点的深度值在运行深度测试前被插值到每个子样本中，对于模板测试，我们为每个子样本储存模板值，而不是每个像素。这意味着深度和模板缓冲的大小随着像素子样本的增加也增加了。

到目前为止我们所讨论的不过是多采样发走样工作的方式。光栅化背后实际的逻辑要比我们讨论的复杂，但你现在可以理解多采样抗锯齿背后的概念和逻辑了。

OpenGL 中的 MSAA

如果我们打算在 OpenGL 中使用 MSAA，那么我们必须使用一个可以为每个像素储存一个以上的颜色值的颜色缓冲(因为多采样需要我们为每个采样点储存一个颜色)。我们这就需要一个新的缓冲类型，它可以储存要求数量的多重采样样本，它叫做**多样本缓冲(multisample buffer)**。

多数窗口系统可以为我们提供一个多样本缓冲，以代替默认的颜色缓冲。GLFW同样给了我们这个功能，我们所要作的就是提示 GLFW，我们希望使用一个带有 N 个样本的多样本缓冲，而不是普通的颜色缓冲，这要在创建窗口前调用 `glfwWindowHint` 来完成：

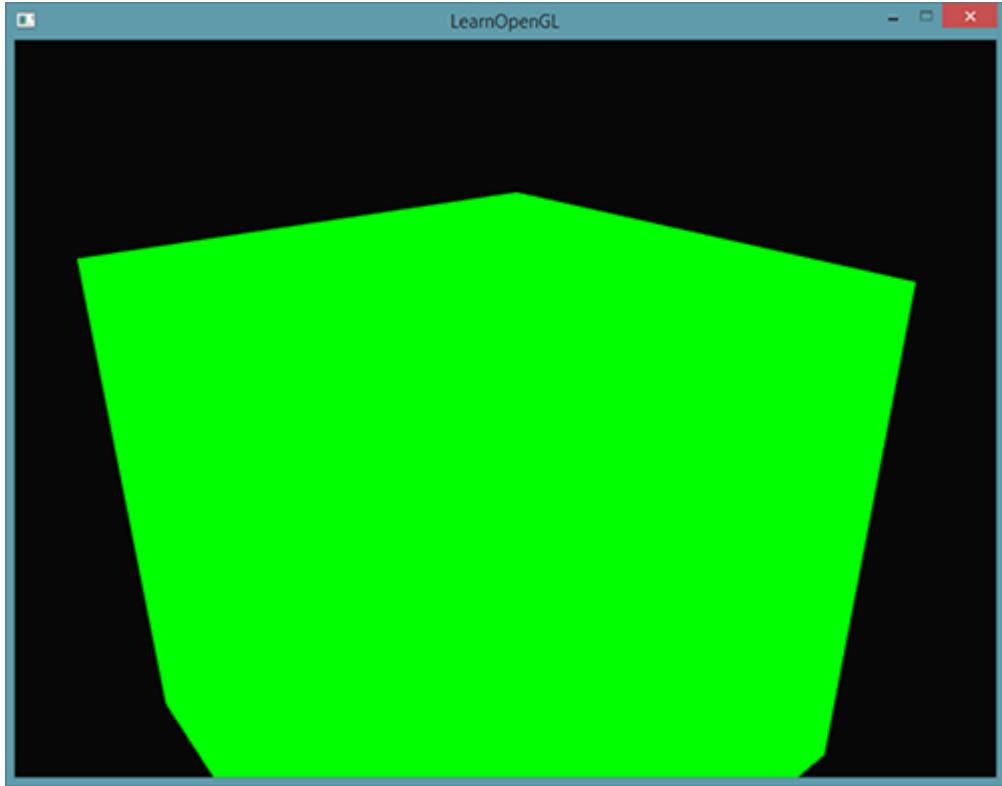
```
glfwWindowHint(GLFW_SAMPLES, 4);
```

当我们现在调用 `glfwCreateWindow`，用于渲染的窗口就被创建了，这次每个屏幕坐标使用一个包含 4 个子样本的颜色缓冲。这意味着所有缓冲的大小都增长 4 倍。

现在我们请求 GLFW 提供了多样本缓冲，我们还要调用 `glEnable` 来开启多采样，参数是 `GL_MULTISAMPLE`。大多数 OpenGL 驱动，多采样默认是开启的，所以这个调用有点多余，但通常记得开启它是个好主意。这样所有 OpenGL 实现的多采样都开启了。

```
glEnable(GL_MULTISAMPLE);
```

当默认帧缓冲有了多采样缓冲附件的时候，我们所要做的全部就是调用 `glEnable` 开启多采样。因为实际的多采样算法在 OpenGL 驱动光栅化里已经实现了，所以我们无需再做什么了。如果我们现在来渲染教程开头的那个绿色立方体，我们会看到边缘变得平滑了：



这个箱子看起来平滑多了，在场景中绘制任何物体都可以利用这个技术。可以[从这里找到](#)这个简单的例子。

离屏 MSAA

因为 GLFW 负责创建多采样缓冲，开启 MSAA 非常简单。如果我们打算使用我们自己的帧缓冲，来进行离屏渲染，那么我们就必须自己生成多采样缓冲了；现在我们需要自己负责创建多采样缓冲。

有两种方式可以创建多采样缓冲，并使其成为帧缓冲的附件：纹理附件和渲染缓冲附件，和[帧缓冲教程](#)里讨论过的普通的附件很相似。

多采样纹理附件

为了创建一个支持储存多采样点的纹理，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);
```

第二个参数现在设置了我们打算让纹理拥有的样本数。如果最后一个参数等于 `GL_TRUE`，图像上的每一个纹理像素（texel）将会使用相同的样本位置，以及同样的子样本数量。

为将多采样纹理附加到帧缓冲上，我们使用 `glFramebufferTexture2D`，不过这次纹理类型是 `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有了一个纹理图像形式的多采样颜色缓冲。

多采样渲染缓冲对象（Multisampled renderbuffer objects）

和纹理一样，创建一个多采样渲染缓冲对象不难。而且很简单，因为我们所要做的全部就是当我们指定渲染缓冲的内存的时候将 `glRenderbufferStorage` 改为 `glRenderbufferStorageMultisample`：

```
glRenderbufferStorageMultisample(GL_RENDERBUFFER, 4,  
GL_DEPTH24_STENCIL8, width, height);
```

有一样东西在这里有变化，就是缓冲目标后面那个额外的参数，我们将其设置为样本数量，当前的例子中应该是 4.

渲染到多采样帧缓冲

渲染到多采样帧缓冲对象是自动的。当我们绘制任何东西时，帧缓冲对象就绑定了，光栅化会对负责所有多采样操作。我们接着得到了一个多采样颜色缓冲，以及深度和模板缓冲。因为多采样缓冲有点特别，我们不能为其他操作直接使用它们的缓冲图像，比如在着色器中进行采样。

一个多采样图像包含了比普通图像更多的信息，所以我们需要做的是压缩或还原图像。还原一个多采样帧缓冲，通常用 `glBlitFramebuffer` 来完成，它从一个帧缓冲中复制一个区域粘贴另一个里面，同时也将任何多采样缓冲还原。

`glBlitFramebuffer` 把一个 4 屏幕坐标源区域传递到一个也是 4 空间坐标的目标区域。你可能还记得帧缓冲教程中，如果我们绑定到 `GL_FRAMEBUFFER`，我们实际上就同时绑定到了读和写的帧缓冲目标。我们还可以通过 `GL_READ_FRAMEBUFFER` 和 `GL_DRAW_FRAMEBUFFER` 绑定到各自的目标上。`glBlitFramebuffer` 函数从这两个目标读取，并决定哪一个是源哪一个是目标帧缓冲。接着我们就可以通过把图像位块传送(Blitting)到默认帧缓冲里，将多采样帧缓冲输出传递到实际的屏幕了：

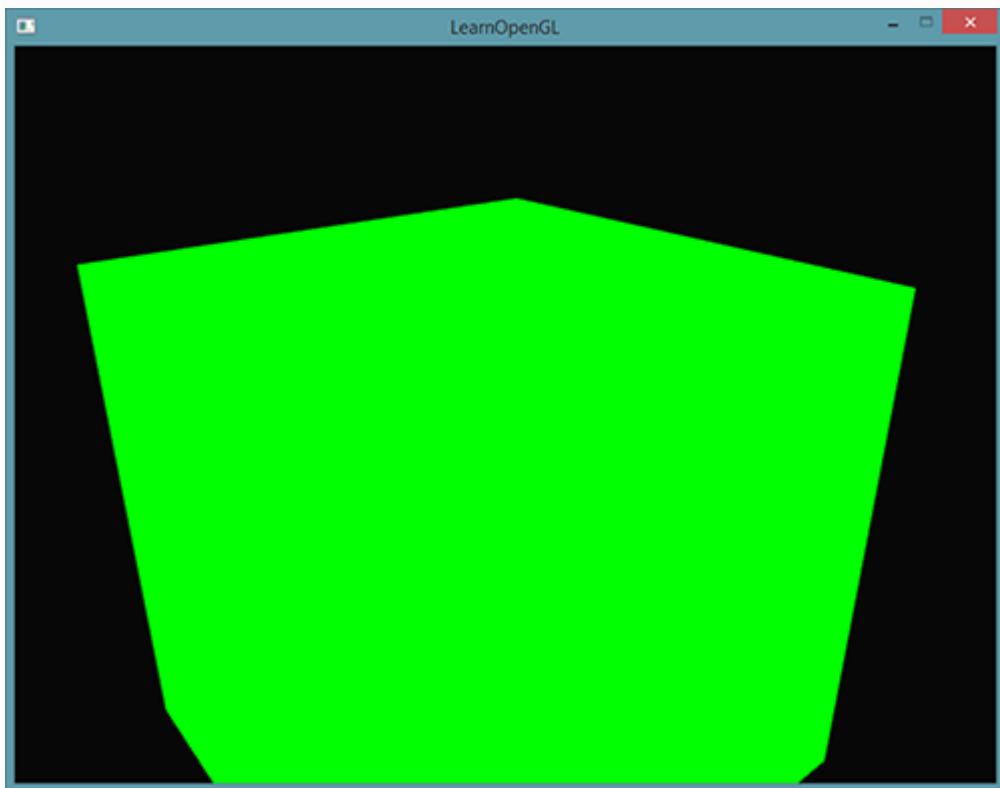
```
glBindFramebuffer(GL_READ_FRAMEBUFFER, multisampledFBO);
```

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
```

```
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height,
```

```
GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

如果我们渲染应用，我们将得到和没用帧缓冲一样的结果：一个绿色立方体，它使用 MSAA 显示出来，但边缘锯齿明显少了：



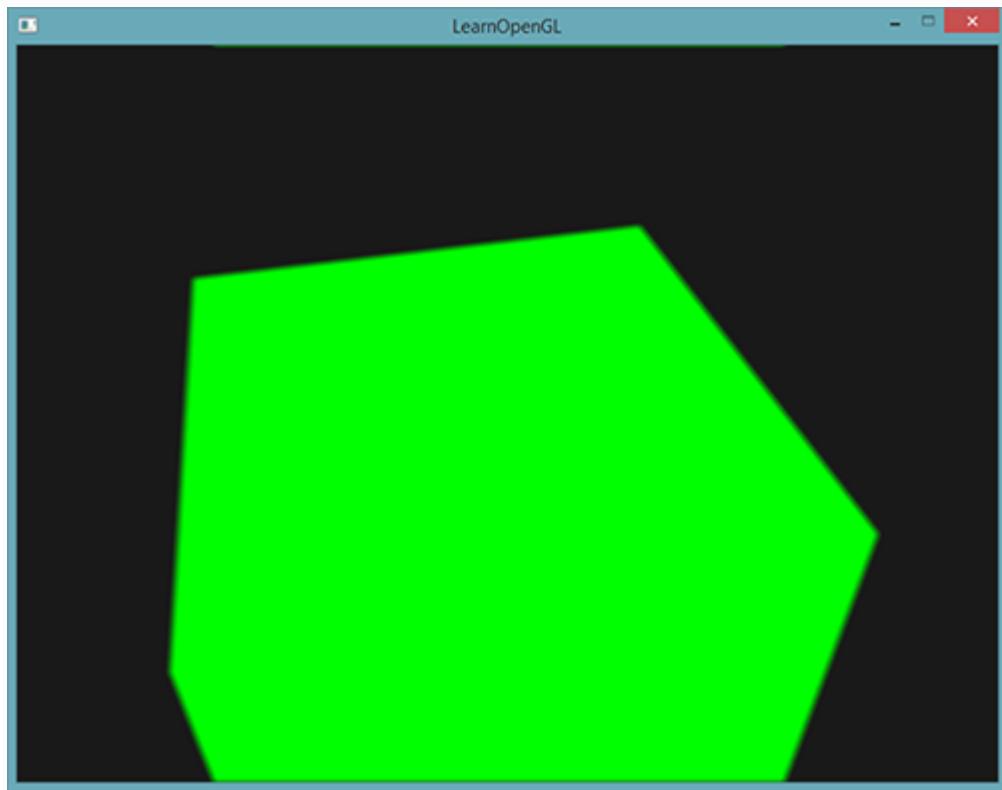
[你可以在这里找到源代码。](#)

但是如果我们打算使用一个多采样帧缓冲的纹理结果来做这件事，就像后处理一样会怎样？我们不能在片段着色器中直接使用多采样纹理。我们可以做的事情是把多缓冲位块传送(Blit)到另一个带有非多采样纹理附件的 FBO 中。之后我们使用这个普通的颜色附件纹理进行后处理，通过多采样来对一个图像渲染进行后处理效率很高。这意味着我们必须生成一个新的 FBO，它仅作为一个将多采样缓冲还原为一个我们可以在片段着色器中使用的普通 2D 纹理中介。伪代码是这样的：

```
GLuint msFBO = CreateFBOWithMultiSampledAttachments();  
  
// Then create another FBO with a normal texture color attachment  
  
...  
  
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,  
GL_TEXTURE_2D, screenTexture, 0);  
  
...  
  
while(!glfwWindowShouldClose(window))  
  
{  
    ...  
  
    glBindFramebuffer(msFBO);  
  
    ClearFrameBuffer();  
  
    DrawScene();  
  
    // Now resolve multisampled buffer(s) into intermediate FBO  
  
    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);  
  
// Now scene is stored as 2D texture image, so use that image for  
post-processing  
  
glBindFramebuffer(GL_FRAMEBUFFER, 0);  
  
ClearFramebuffer();  
  
glBindTexture(GL_TEXTURE_2D, screenTexture);  
  
DrawPostProcessingQuad();  
  
...  
}
```

如果我们实现帧缓冲教程中讲的后处理代码，我们就能创造出没有锯齿边的所有效果很酷的后处理特效。使用模糊 kernel 过滤器，看起来会像这样：



你可以在这里找到所有 MSAA 版本的后处理源码。

Important

因为屏幕纹理重新变回了只有一个采样点的普通纹理，有些后处理过滤器，比如边检测（edge-detection）将会再次导致锯齿边问题。为了修正此问题，之后你应该对纹理进行模糊处理，或者创建你自己的抗锯齿算法。

当我们希望将多采样和离屏渲染结合起来时，我们需要自己负责一些细节。所有细节都是值得付出这些额外努力的，因为多采样可以明显提升场景视频输出的质量。要注意，开启多采样会明显降低性能，样本越多越明显。本文写作时，MSAA4 样本很常用。

自定义抗锯齿算法

可以直接把一个多采样纹理图像传递到着色器中，以取代必须先还原的方式。GLSL 给我们一个选项来为每个子样本进行纹理图像采样，所以我们可以创建自己的抗锯齿算法，在比较大的图形应用中，通常这么做。

为获取每个子样本的颜色值，你必须将纹理 uniform 采样器定义为 sampler2DMS，而不是使用 sampler2D：

```
uniform sampler2DMS screenTextureMS;
```

使用 `texelFetch` 函数，就可以获取每个样本的颜色值了：

```
vec4 colorSample = texelFetch(screenTextureMS, TexCoords, 3); // 4th
```

subsample

我们不会深究自定义抗锯齿技术的创建细节，但是会给你自己去实现它提供一些提示。

高级光照

原文	Advanced Lighting
作者	JoeyDeVries
翻译	Django

原文

[Advanced Lighting](#)

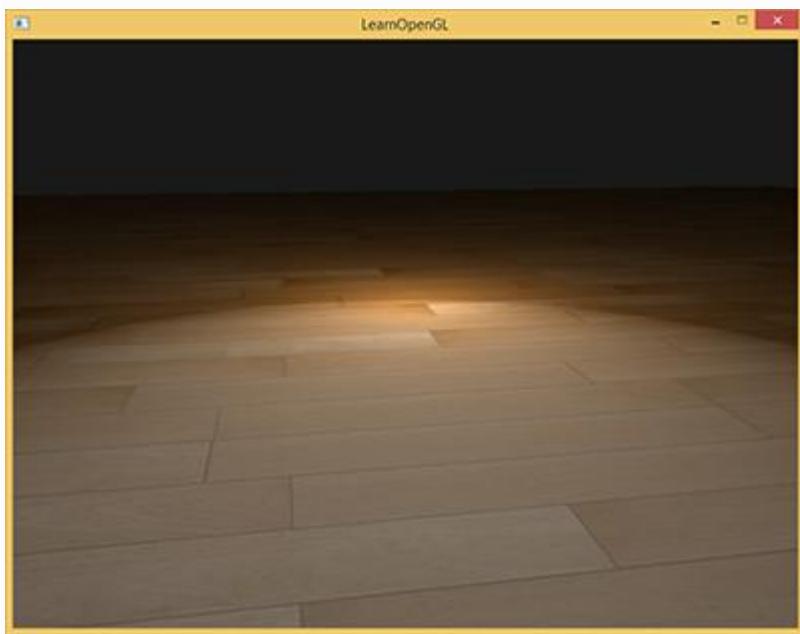
校对

gjy_1992

在光照教程中，我们简单的介绍了 Phong 光照模型，它给我们的场景带来的基本的现实感。Phong 模型看起来还不错，但本章我们把重点放在一些细微差别上。

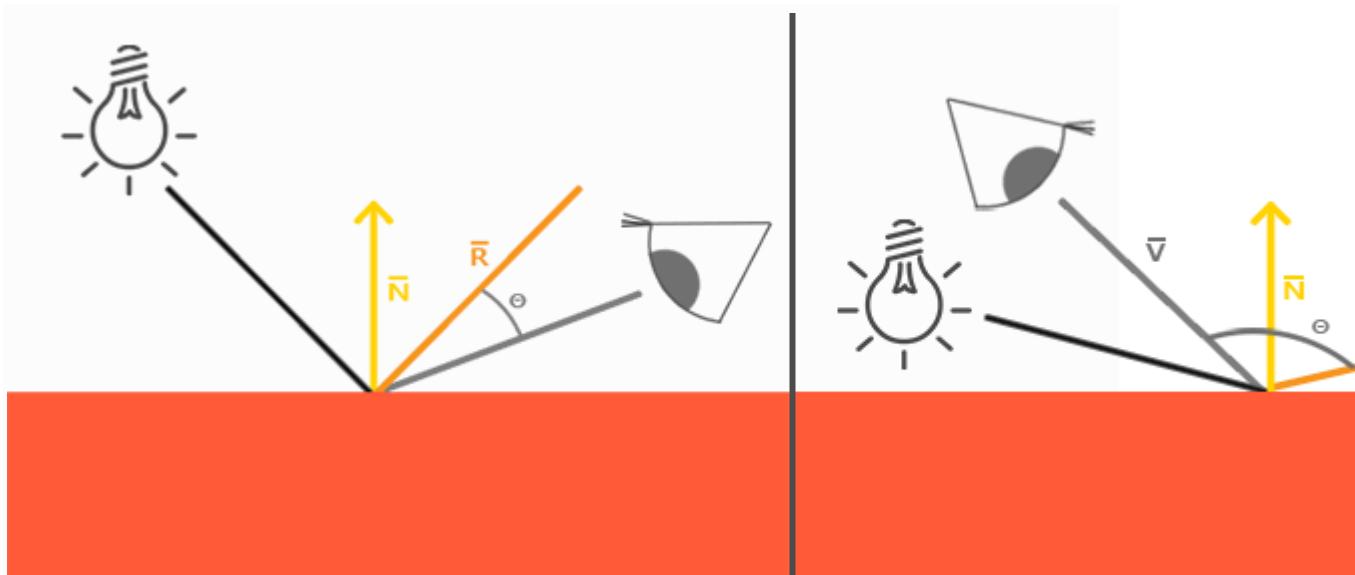
Blinn-Phong

Phong 光照很棒，而且性能较高，但是它的镜面反射在某些条件下会失效，特别是当发光值属性低的时候，对应一个非常大的粗糙的镜面区域。下面的图片展示了，当我们使用镜面的发光值为 1.0 时，一个带纹理地板的效果：



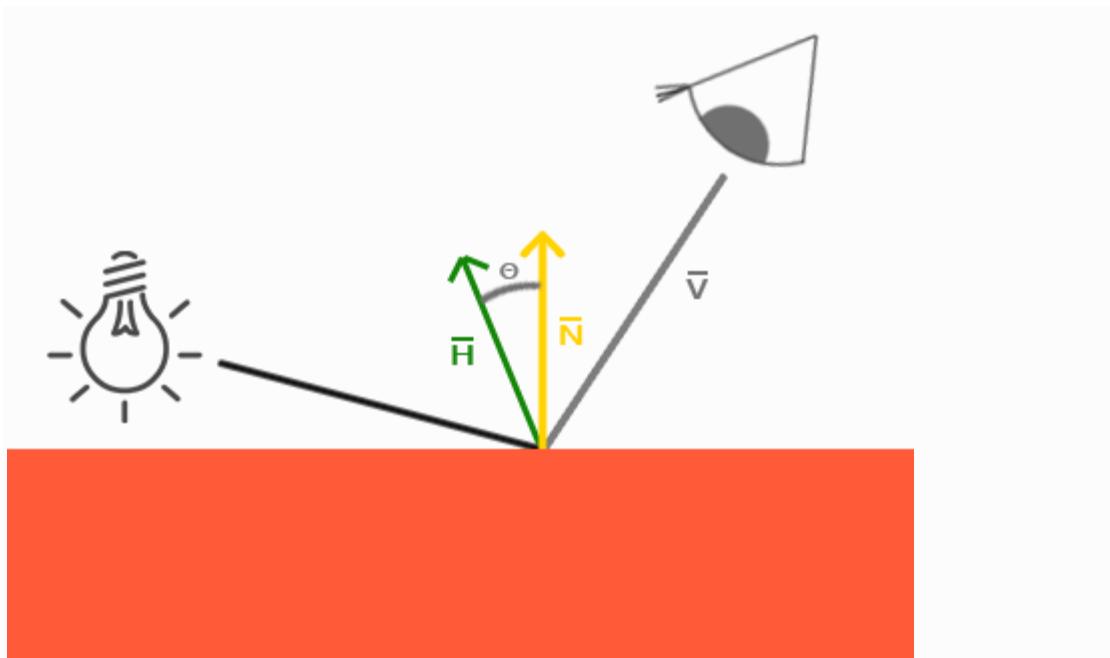
你可以看到，镜面区域边缘迅速减弱并截止。出现这个问题的原因是在视线向量和反射向量的角度不允许大于 90 度。如果大于 90 度的话，点乘的结果就会是负数，镜面的贡献成分就会变成 0。你可能会想，这不是一个问题，因为大于 90 度时我们不应看到任何光，对吧？

错了，这只适用于漫散射部分，当法线和光源之间的角度大于 90 度时意味着光源在被照亮表面的下方，这样光的散射成分就会是 0.0。然而，对于镜面光照，我们不会测量光源和法线之间的角度，而是测量视线和反射方向向量之间的。看看下面的两幅图：



现在看来问题就很明显了。左侧图片显示 Phong 反射的 θ 小于 90 度的情况。我们可以看到右侧图片视线和反射之间的角 θ 大于 90 度，这样镜面反射成分将会被消除。通常这也不是问题，因为视线方向距离反射方向很远，但如果我们使用一个数值较低的发光值参数的话，镜面半径就会足够大，以至于能够贡献一些镜面反射的成份了。在例子中，我们在角度大于 90 度时消除了这个贡献（如第一个图片所示）。

1977 年 James F. Blinn 引入了 Blinn-Phong 着色，它扩展了我们目前所使用的 Phong 着色。Blinn-Phong 模型很大程度上和 Phong 是相似的，不过它稍微改进了 Phong 模型，使之能够克服我们所讨论到的问题。它放弃使用反射向量，而是基于我们现在所说的一个叫做半程向量（halfway vector）的向量，这是个单位向量，它在视线方向和光线方向的中间。半程向量和表面法线向量越接近，镜面反射成份就越大。



当视线方向恰好与反射向量对称时，半程向量就与法线向量重合。这样观察者距离原来的反射方向越近，镜面反射的高光就会越强。

这里，你可以看到无论观察者往哪里看，半程向量和表面法线之间的夹角永远都不会超过 90 度（当然除了光源远远低于表面的情况）。这样会产生和 Phong 反射稍稍不同的结果，但这时看起来会更加可信，特别是发光值参数比较低的时候。Blinn-Phong 着色模型也正是早期 OpenGL 固定函数输送管道（fixed function pipeline）所使用的着色模型。

得到半程向量很容易，我们将光的方向向量和视线向量相加，然后将结果归一化（normalize）；

$$\bar{H} = \frac{\bar{L} + \bar{V}}{\|\bar{L} + \bar{V}\|}$$

翻译成 GLSL 代码如下：

```
vec3 lightDir = normalize(lightPos - FragPos);
vec3 viewDir = normalize(viewPos - FragPos);
vec3 halfwayDir = normalize(lightDir + viewDir);
```

实际的镜面反射的计算，就成为计算表面法线和半程向量的点乘，并对其进行约束（大于或等于 0），然后获取它们之间角度的余弦，再添加上发光值参数：

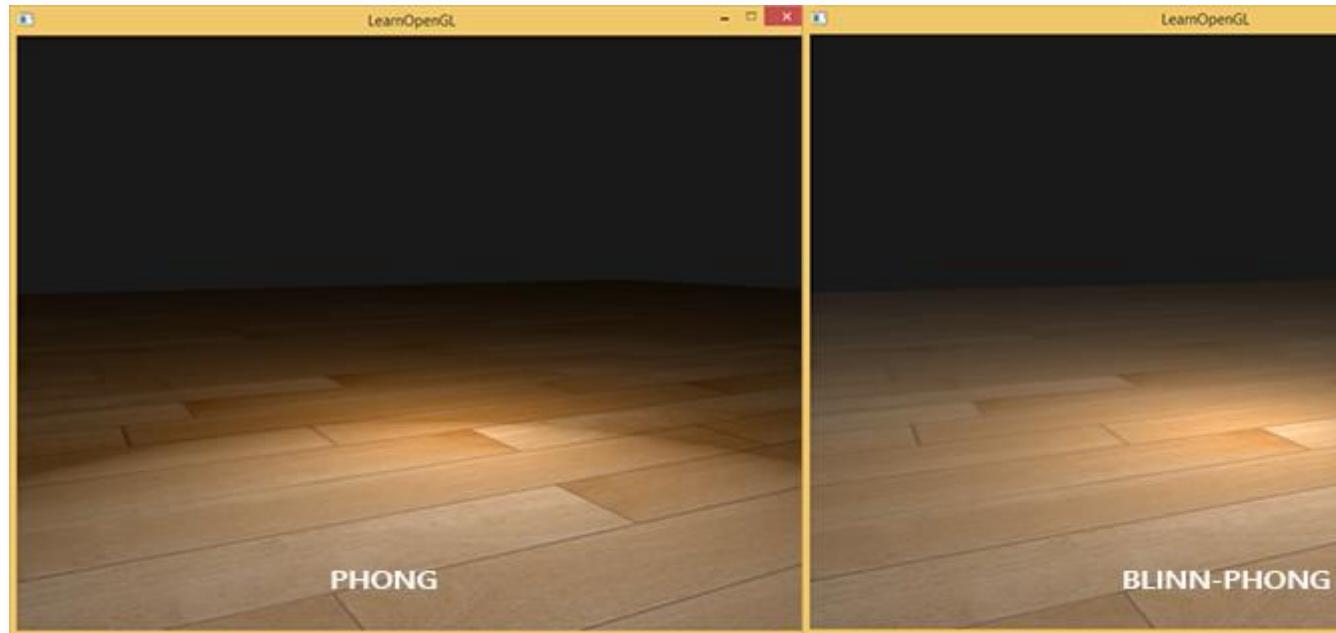
```
float spec = pow(max(dot(normal, halfwayDir), 0.0), shininess);
vec3 specular = lightColor * spec;
```

除了我们刚刚讨论的，Blinn-Phong 没有更多的内容了。Blinn-Phong 和 Phong 的镜面反射唯一不同之处在于，现在我们要测量法线和半程向量之间的角度，而半程向量是视线方向和反射向量之间的夹角。

Important

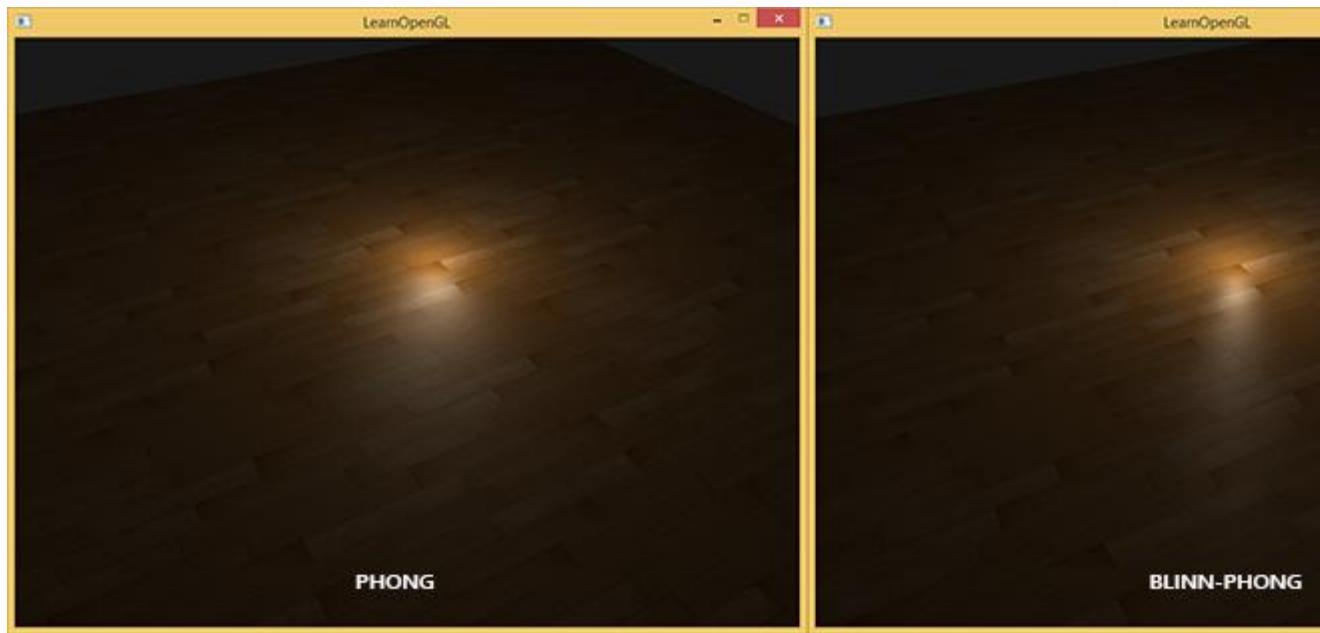
Blinn-Phong 着色的一个附加好处是，它比 Phong 着色性能更高，因为我们不必计算更加复杂的反射向量了。

引入了半程向量来计算镜面反射后，我们再也不会遇到 Phong 着色的骤然截止问题了。下图展示了两种不同方式下发光值指数为 0.5 时镜面区域的不同效果：



Phong 和 Blinn-Phong 着色之间另一个细微差别是，半程向量和表面法线之间的角度经常会比视线和反射向量之间的夹角更小。结果就是，为了获得和 Phong 着色相似的效果，必须把发光值参数设置的大一点。通常的经验是将其设置为 Phong 着色的发光值参数的 2 至 4 倍。

下图是 Phong 指数为 8.0 和 Blinn-Phong 指数为 32 的时候，两种 specular 反射模型的对比：



你可以看到 Blinn-Phong 的镜面反射成分要比 Phong 锐利一些。这通常需要使用一点小技巧才能获得之前你所看到的 Phong 着色的效果，但 Blinn-Phong 着色的效果比默认的 Phong 着色通常更加真实一些。

这里我们用到了一个简单像素着色器，它可以在普通 Phong 反射和 Blinn-Phong 反射之间进行切换：

```
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简单的 [demo 的源码](#) 以及 [顶点](#) 和 [片段](#) 着色器。按下 **b** 键，这个 demo 就会从 Phong 切换到 Blinn-Phong 光照，反之亦然。

本文作者 JoeyDeVries，由 Django 翻译自 <http://learnopengl.com>

Gamma 校正(Gamma Correction)

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）。

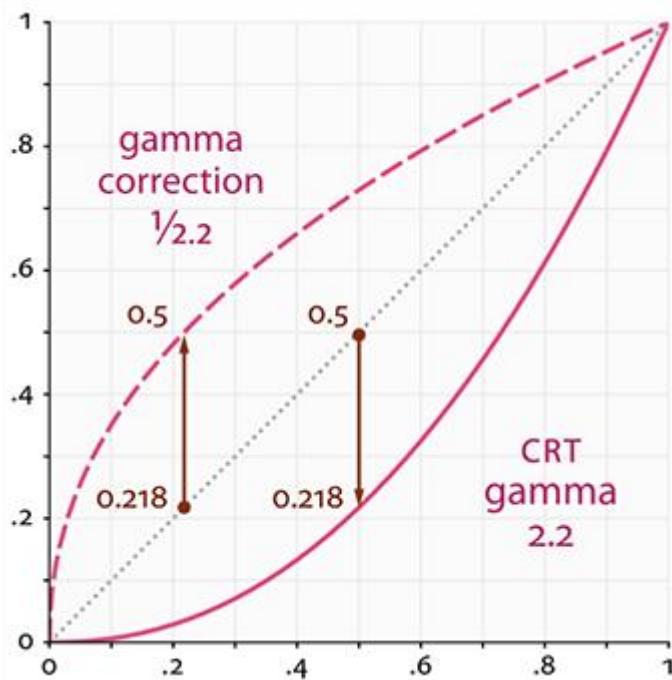
人类所感知的亮度恰好和 CRT 所显示出来相似的指数关系非常匹配。为了更好的理解所有含义，请看下面的图片：

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：这里的意思是说比如一个东西的亮度 0.3，让人感觉它比原来变亮一倍，那么现在这个亮度应该成为 0.6，而不是 0.4，也就是说人眼感知到的亮度的变化并非线性均匀分布的。问题的关键在于这样的一倍相当于一个亮度级，例如假设 0.1、0.2、0.4、0.8 是我们定义的四个亮度级别，在 0.1 和 0.2 之间人眼只能识别出 0.15 这个中间级，而虽然 0.4 到 0.8 之间的差距更大，这个区间人眼也只能识别出一个颜色）。然而，当我们谈论光的物理亮度，也就是光子的数量的多少的时候，底部的灰阶显示出的是这时讨论的亮度。底部的灰阶显示出的是双倍的亮度所返回的物理亮度（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），所以它看起来很奇怪。

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 $L=(0.5, 0.0, 0.0)$ 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 $(1.0, 0.0, 0.0)$ ，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 $(0.218, 0.0, 0.0)$ 。在这儿问题就出现了：当我们将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），但最终还是要把所哟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 $(0.5, 0.0, 0.0)$ 。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 2.2 次幂的亮度，所以倒数就是 $1/2.2$ 次幂。**Gamma** 校正后的暗红色就会成为

$$\{(0.5, 0.0, 0.0)\}^{1/2.2} = \{(0.5, 0.0, 0.0)\}^{0.45} = \{0.73, 0.0,$$
$$0.0\}$$

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

$$(0.73, 0.0, 0.0)^{2.2} = (0.5, 0.0, 0.0)$$

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。

Important

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，如果你使用多个怎还冲，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

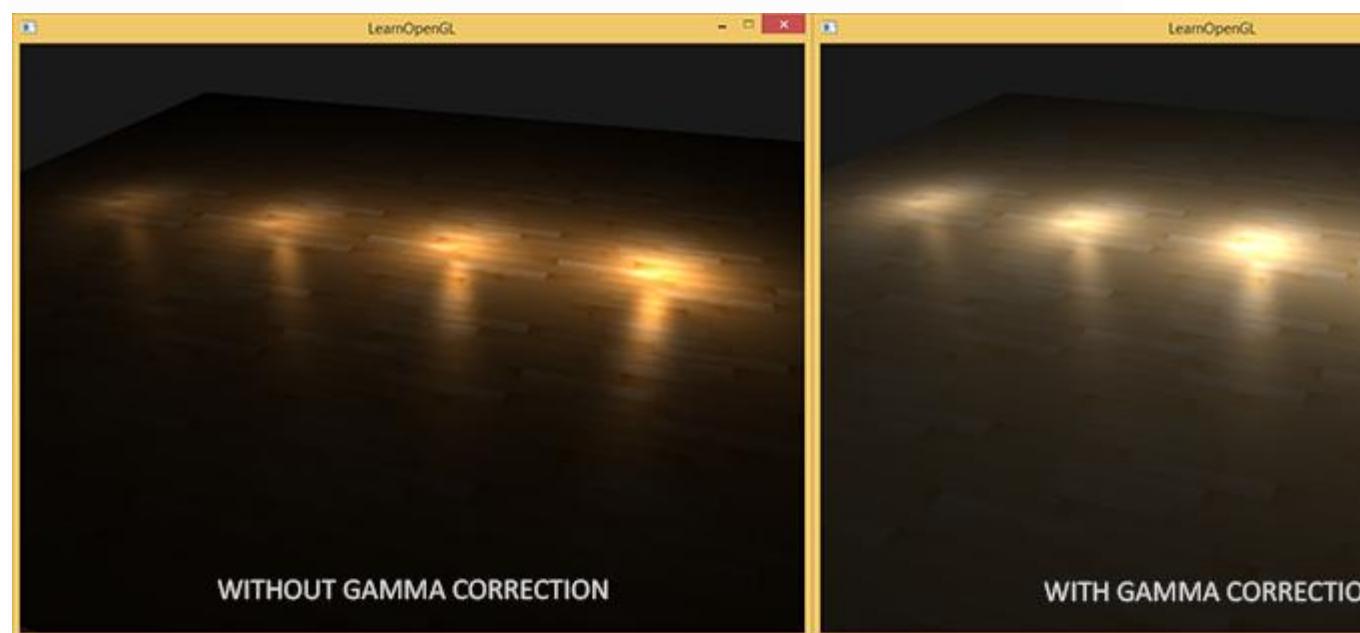
中引入后处理阶段，在后处理四边形上应用 **gamma** 校正，这样你只要做一次就好了。

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。我们可以这样把这个纹理指定为一个 `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

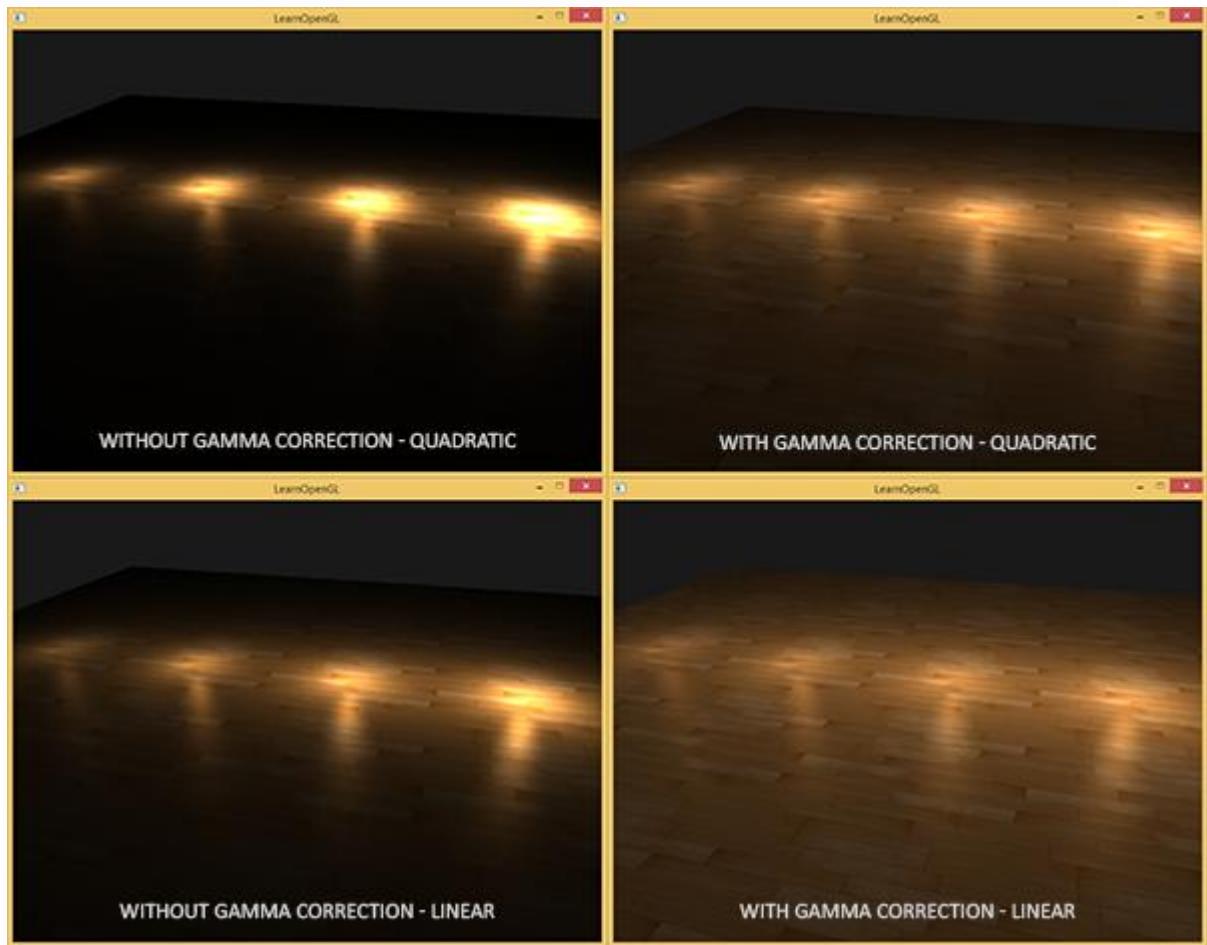
在使用了 `gamma` 校正之后，另一个不同之处是光照衰减。真实的物理世界中，光照的衰减和光源的距离的平方成反比。

```
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，不过但当我们开启 `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：

$$\{(1.0 / \text{distance}^2)\}^{2.2}$$

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

$$\{(1.0 / \text{distance})\}^{2.2} = 1.0 / \text{distance}^{2.2}$$

这和物理公式是很相似的。

Important

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。

renderwont.com: 一些额外的实践上的思考。

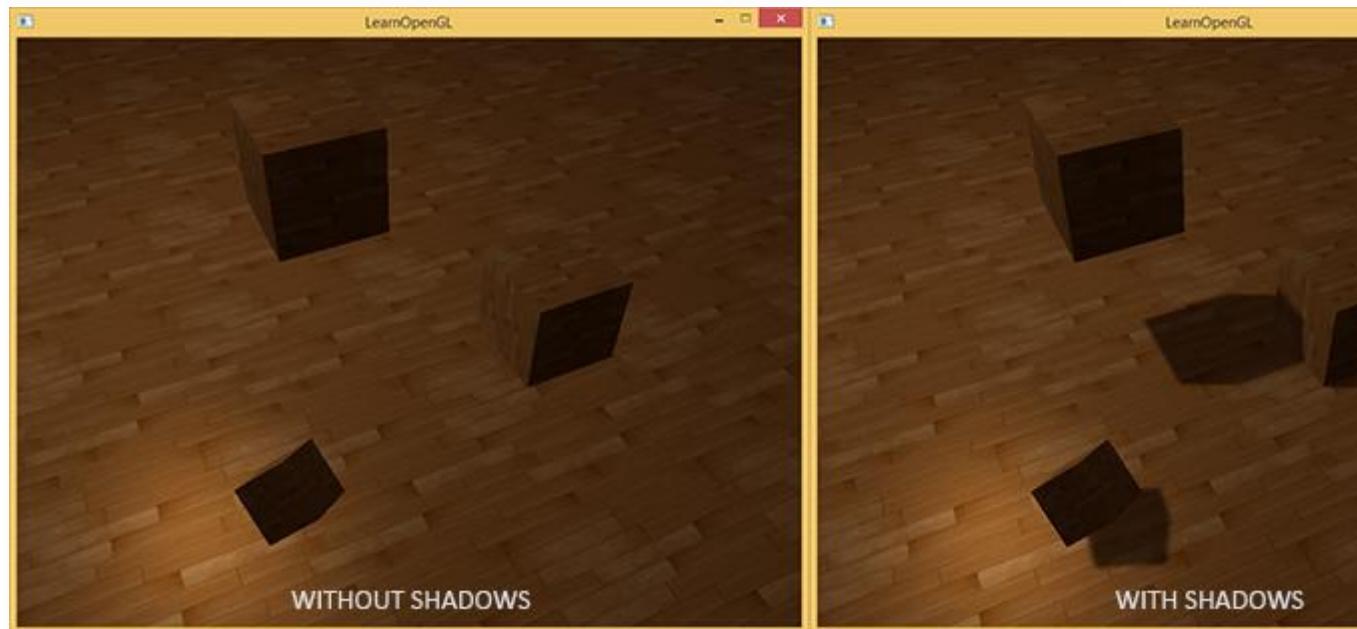
Shadow Mapping

阴影映射(Shadow Mapping)

本文作者 JoeyDeVries，由 Django 翻译自 <http://learnopengl.com>

原文	Shadow Mapping
作者	JoeyDeVries
翻译	Django
校对	gjy_1992

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：



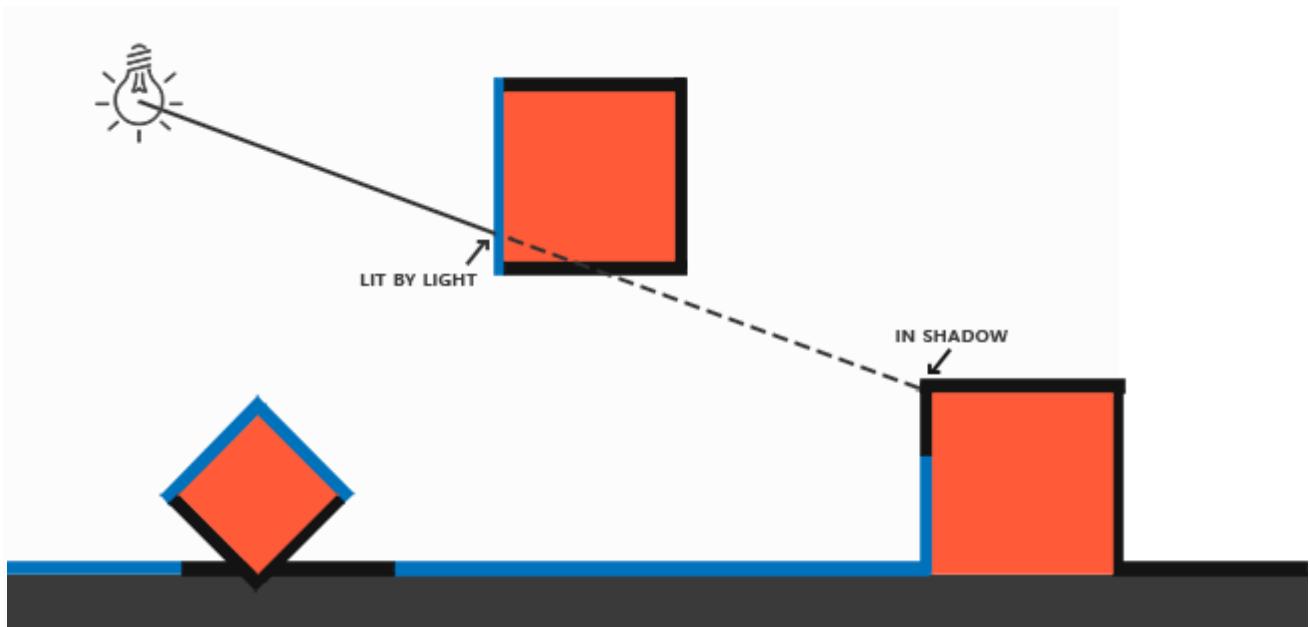
你可以看到，有阴影的时候你能更容易地区分出物体之间的位置关系，例如，当使用阴影的时候浮在地板上的立方体的事实更加清晰。

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

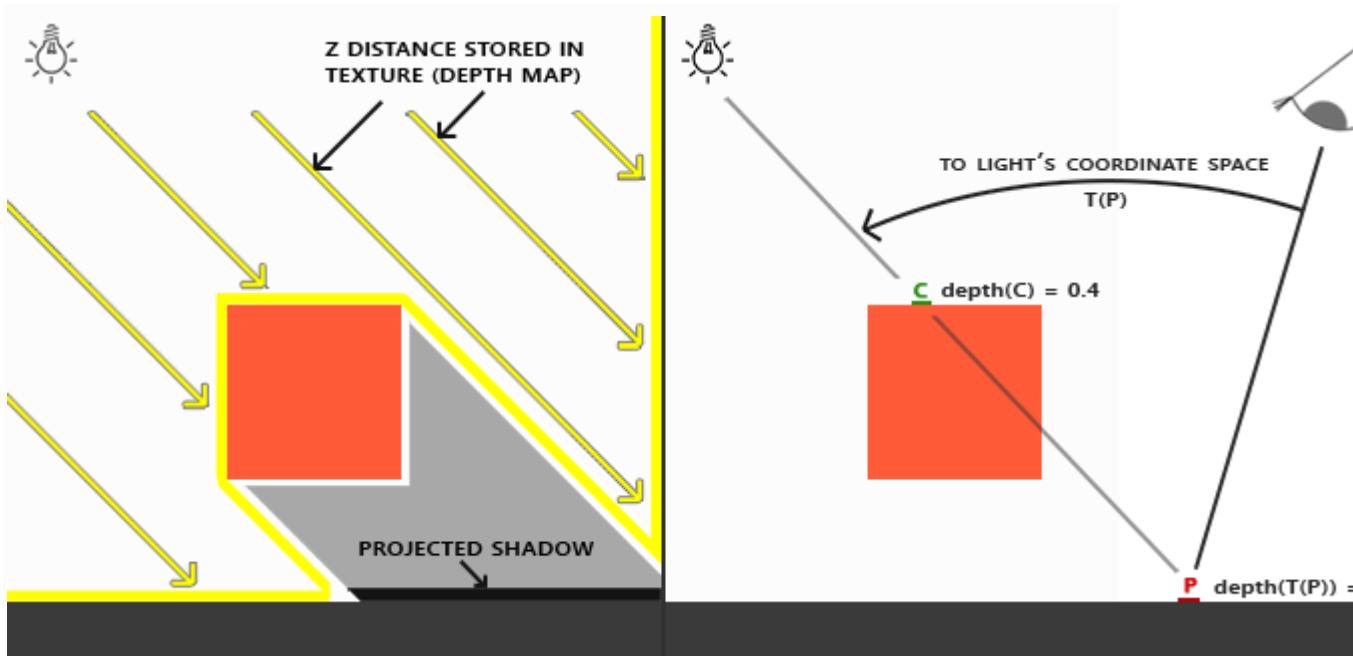
阴影映射背后的思路非常简单：我们以光的位置为视角进行渲染，我们能看到的东西都将被点亮，看不见的一定是在阴影之中了。假设有一个地板，在光源和它之间有一个大盒子。由于光源处向光线方向看去，可以看到这个盒子，但看不到地板的一部分，这部分就应该在阴影中了。



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元上的线段或射线，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，对应于一个片元的一个 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元了。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，用以决定片元是否在阴影中。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 T 变换，它可以将任何三维位置转变到光源的可见坐标空间。

Important

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，我们必须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 P 处的片元，需要决定它是否在阴影中。我们先得使用 T 把 P 变换到光源的坐标空间里。既然点 P 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 P 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 C ，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 P 的深度，我们可以断定 P 被挡住了，它在阴影中了。

深度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元是否在阴影之中。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 (depth map)

第一步我们需要生成一张深度贴图。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`: 这是深度贴图的解析度。

把我们把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。然而帧缓冲对象不是完全不包含颜色缓冲的，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（通常是窗口解析度）有着不同的解析度，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 (light space transform)

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。当物体和片元不在深度贴图中时，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::vec(0.0f), glm::vec(1.0));
```

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` 正是我们前面称为 `T` 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，并非所有片元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。

渲染出深度贴图

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，最后的片元不需要任何处理，所以我们可以简单地使用一个空像素着色器：

```
#version 330 core

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

这个空像素着色器什么也不干，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);

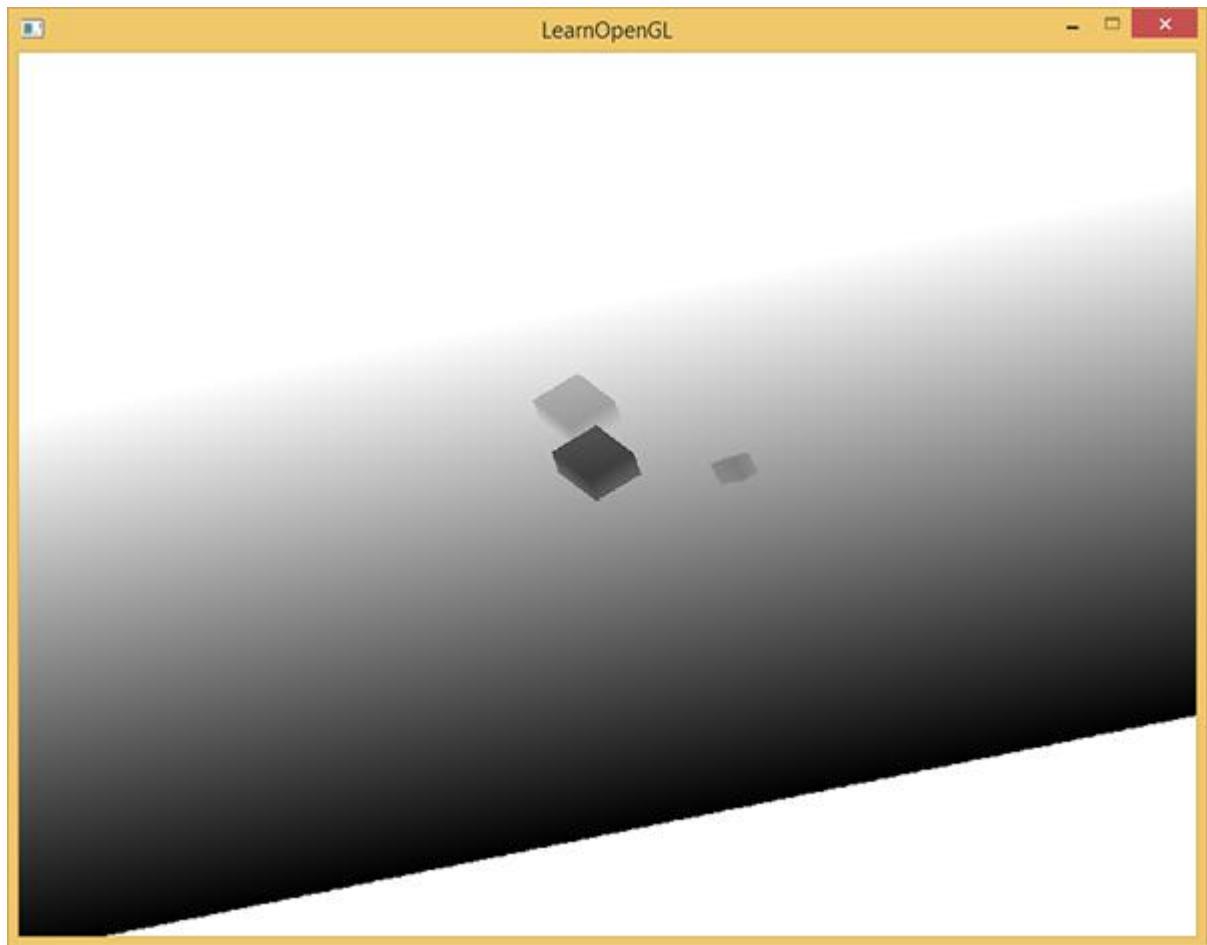
glClear(GL_DEPTH_BUFFER_BIT);

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元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像素着色器：

```
#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。这段代码在像素着色器中执行，用来检验一个片元是否在阴影之中，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` 给像素着色器。

像素着色器使用 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);

}

```

像素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。像素着色器的最后，我们把 `diffuse` 和 `specular` 乘以(1-阴影元素)，这表示这个片元有多大成分不在阴影中。这个像素着色器还需要两个额外输入，一个是光空间的片元位置和第一个渲染阶段得到的深度贴图。

首先要检查一个片元是否在阴影中，把光空间片元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 自动进行一个透视线除法，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` 传到像素着色器里，我们必须自己做透视线除法：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // 执行透视线除法
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    [...]
}
```

返回了片元在光空间的-1 到 1 的范围。

Important

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，当使用透视线投射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;
```

为了得到片元的当前深度，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元的深度。

```
float currentDepth = projCoords.z;
```

实际的对比就是简单检查 `currentDepth` 是否高于 `closestDepth`, 如果是, 那么片元就在阴影中。

```
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;
```

```
// 取得当前片元在光源视角下的深度
```

```
float currentDepth = projCoords.z;
```

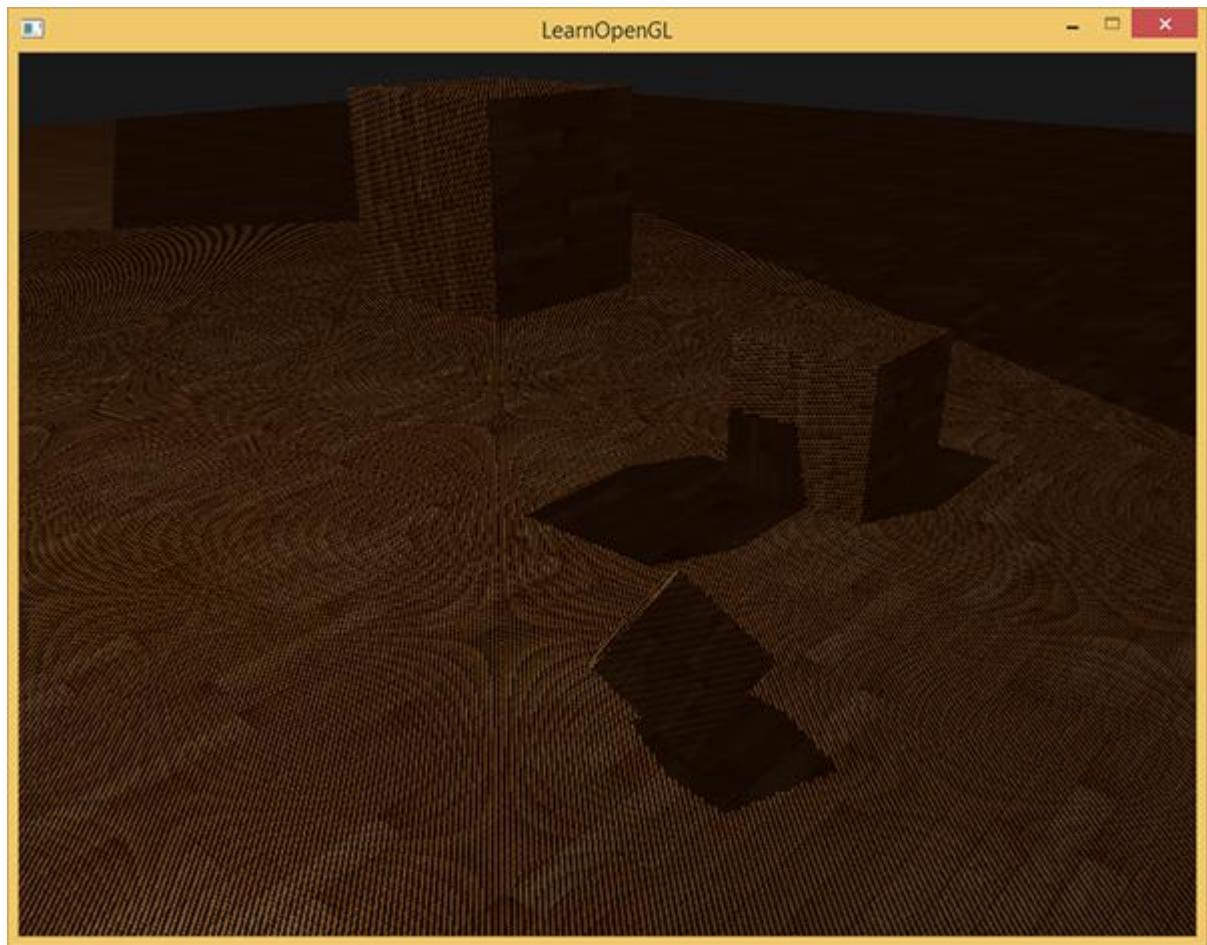
```
// 检查当前片元是否在阴影中
```

```
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:



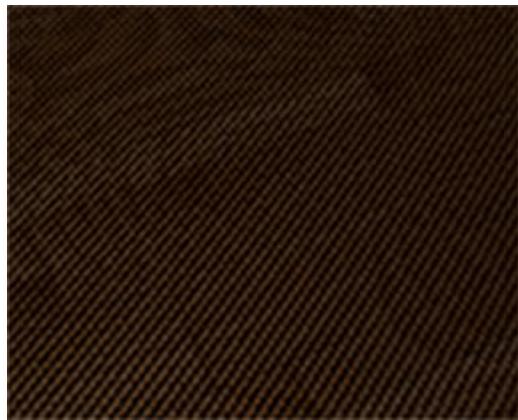
如果你做对了，你会看到地板和上有立方体的阴影。你可以从这里找到 demo 程序的[源码](#)。

改进阴影贴图

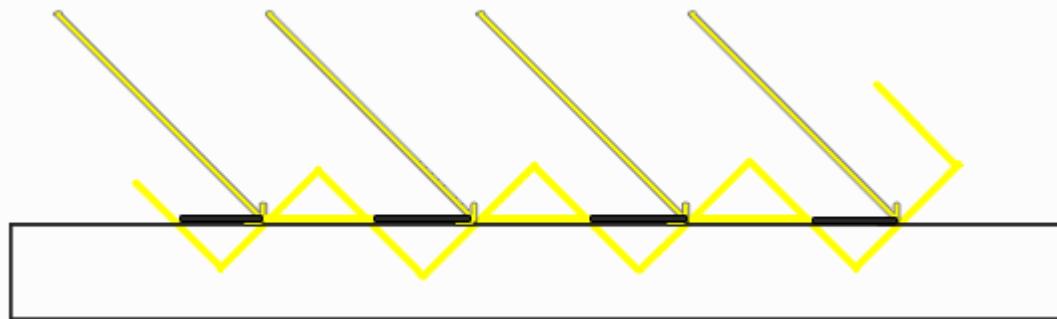
我们试图让阴影映射工作，但是你也看到了，阴影映射还是有点不真实，我们修复它才能获得更好的效果，这是下面的部分所关注的焦点。

阴影失真（shadow acne）

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：



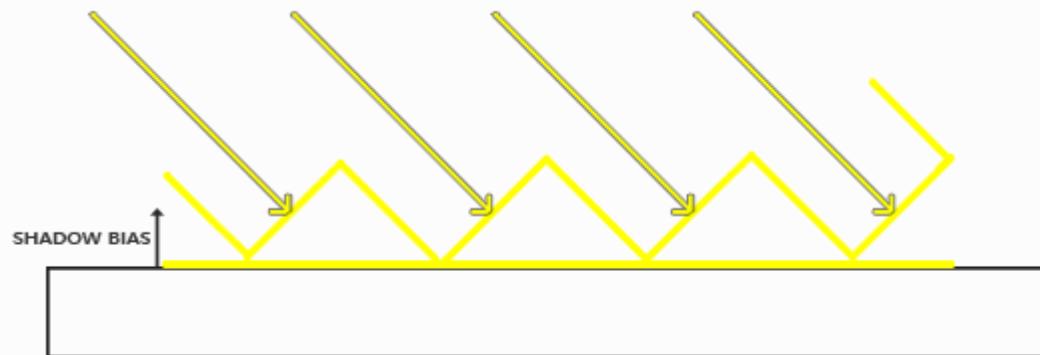
我们可以看到地板四边形渲染出很大一块交替黑线。这种阴影贴图的不真实感叫做阴影失真，下图解释了成因：



因为阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元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，多个片元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。多个片元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，有些片元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，这样片元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：

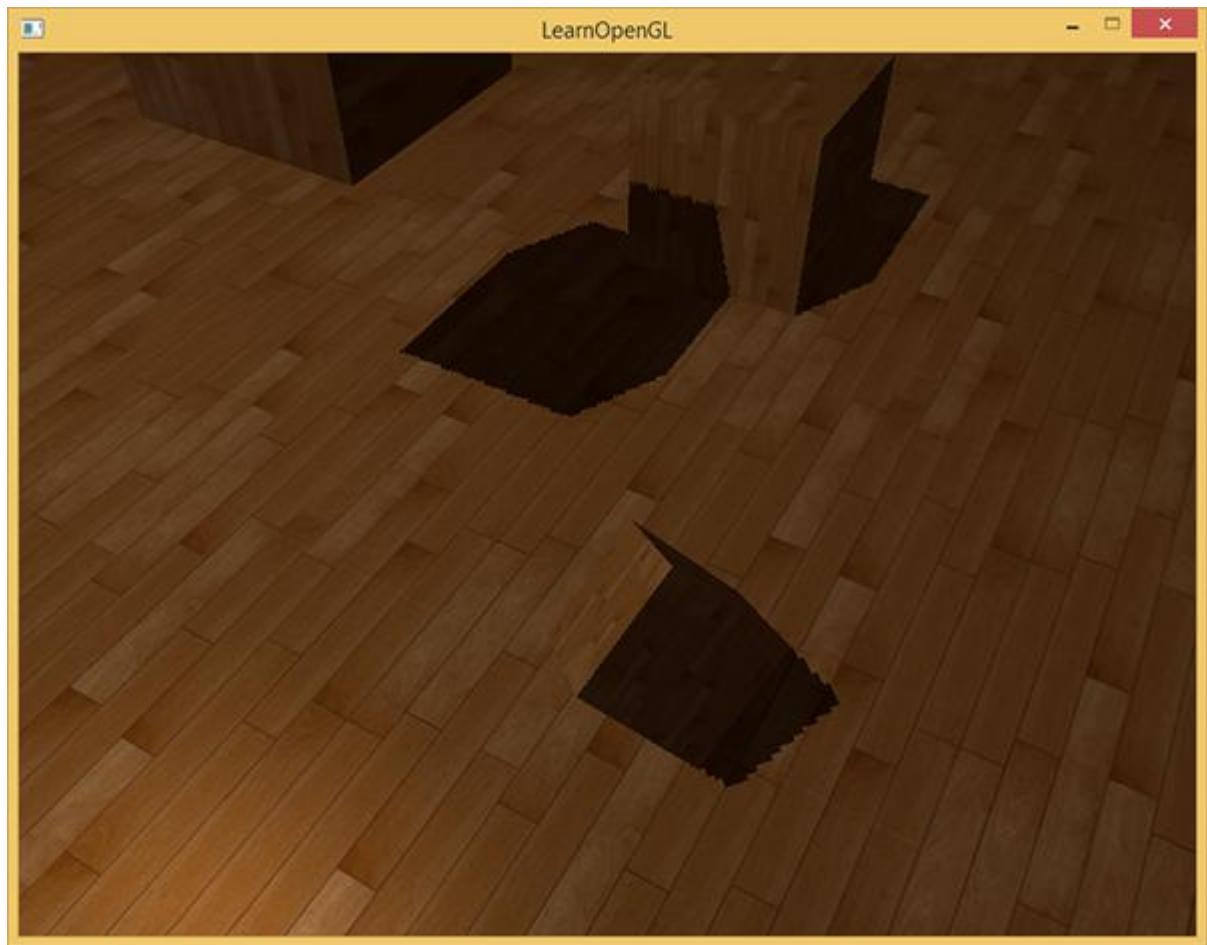
```
float bias = 0.005;
```

```
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);
```

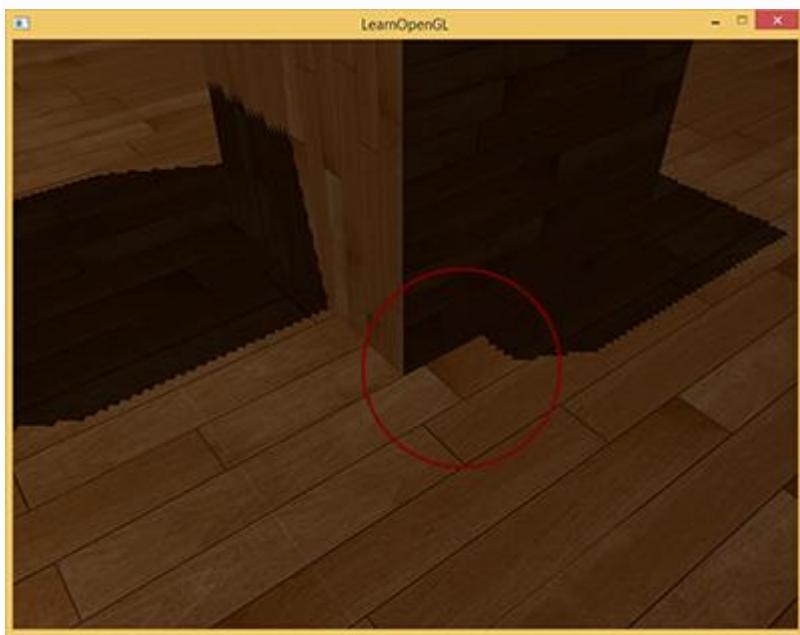
这里我们有一个偏移量的最大值 0.05，和一个最小值 0.005，它们是基于表面法线和光照方向的。这样像地板这样的表面几乎与光源垂直，得到的偏移就很小，而比如立方体的侧面这种表面得到的偏移就更大。下图展示了同一个场景，但使用了阴影偏移，效果的确更好：



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。

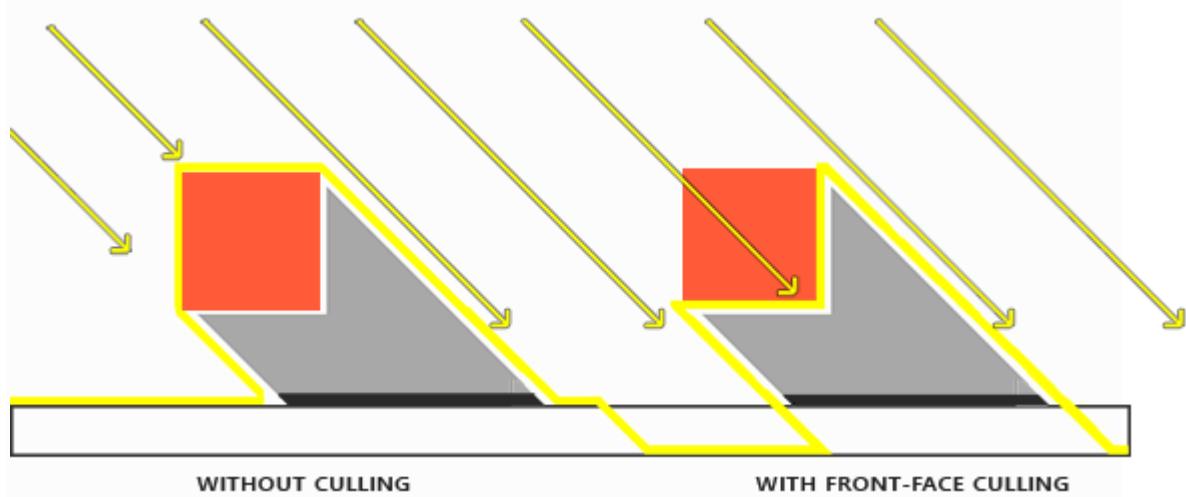
悬浮

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 **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();
```

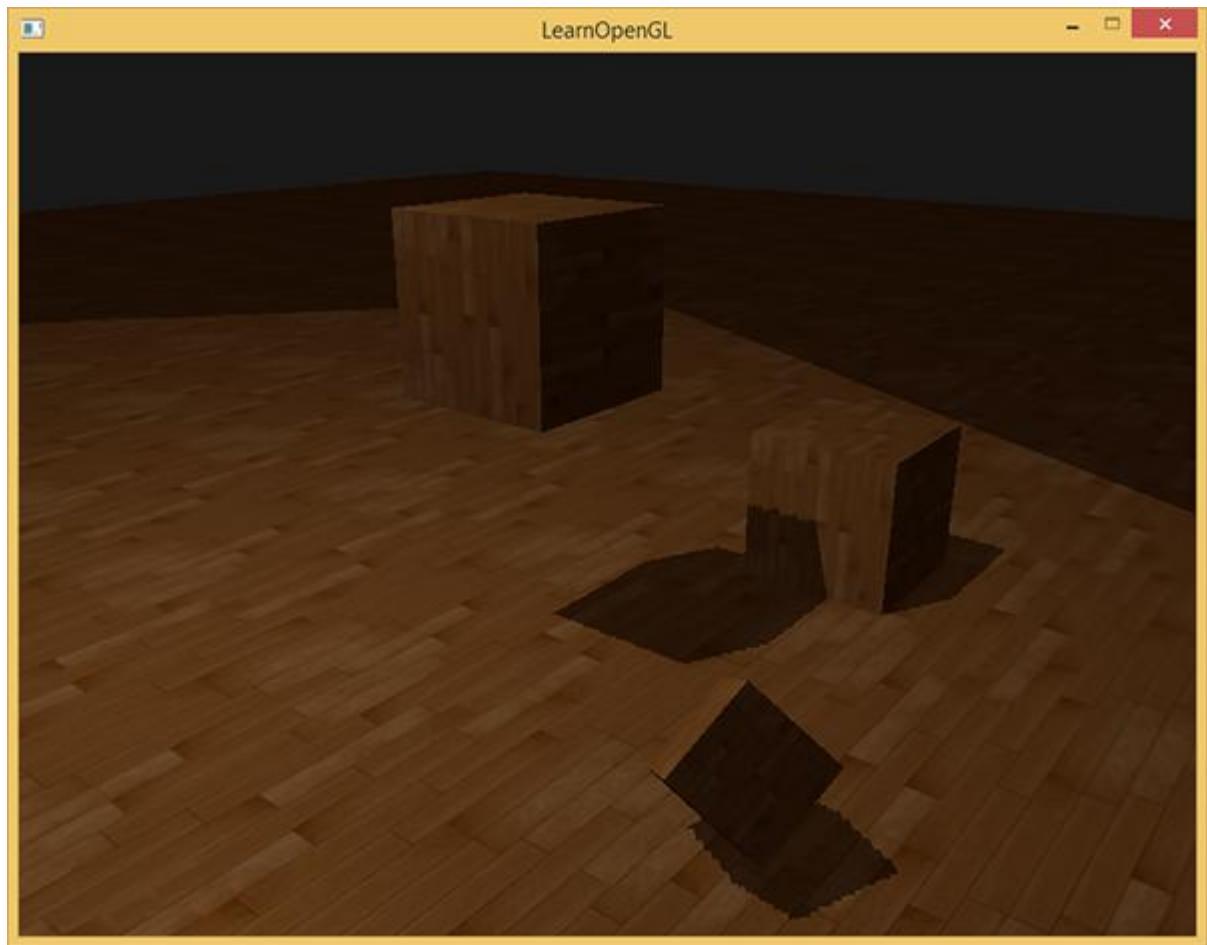
```
glCullFace(GL_BACK); // 不要忘记设回原先的 culling face
```

这十分有效地解决了 **peter panning** 的问题，但只针对实体物体，内部不会对外开口。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**。

采样超出

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。



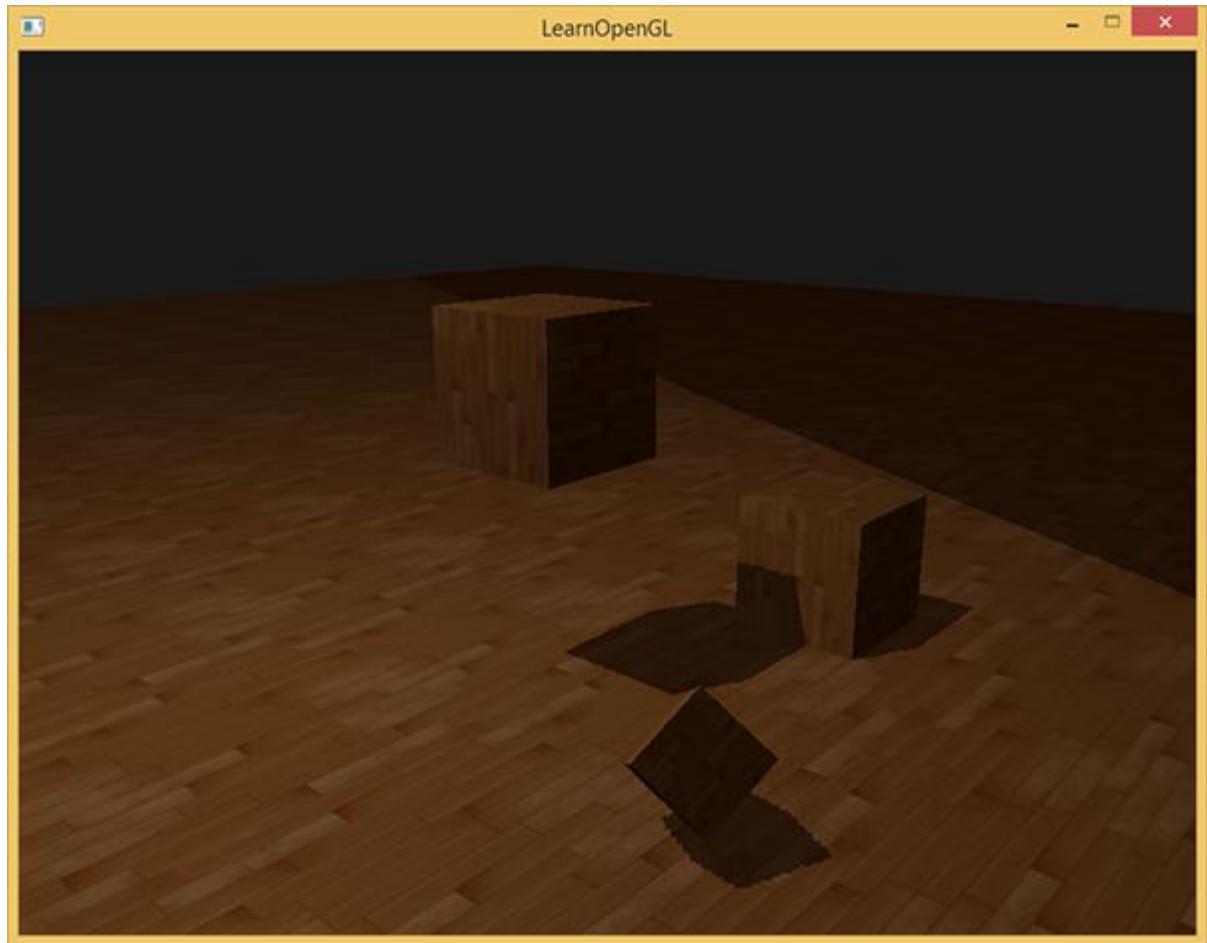
你可以在图中看到，光照有一个区域，超出该区域就成为了阴影；这个区域实际上代表着深度贴图的大小，这个贴图投影到了地板上。发生这种情况的原因是我们之前将深度贴图的环绕方式设置成了 `GL_REPEAT`。

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 };
```

```
glTexParameteri(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。

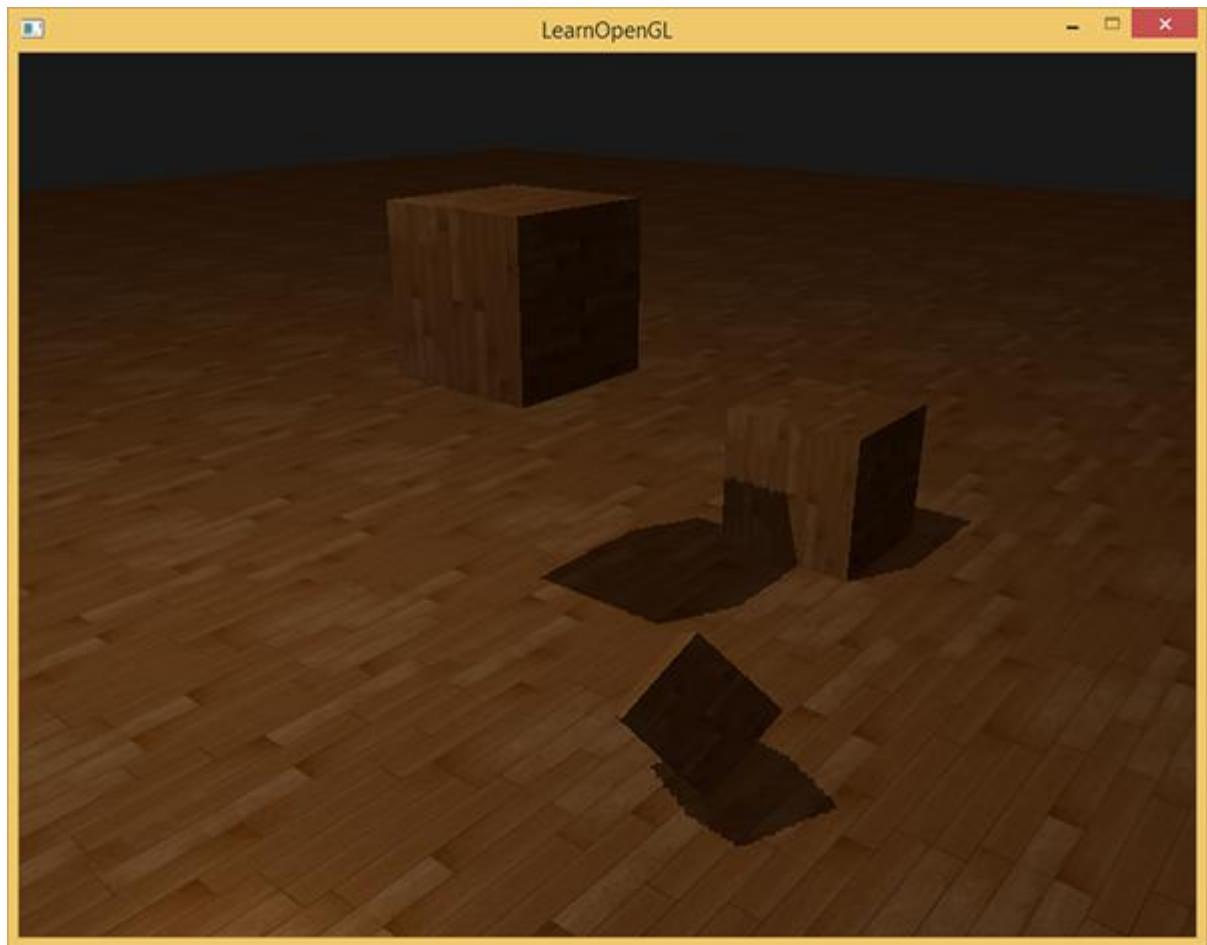
当一个点比光的远平面还要远时，它的投影坐标的 z 坐标大于 1.0。这种情况下，`GL_CLAMP_TO_BORDER` 环绕方式不起作用，因为我们把坐标的 z 元素和深度贴图的值进行了对比；它总是为大于 1.0 的 z 返回 `true`。

解决这个问题也很简单，我们简单的强制把 `shadow` 的值设为 0.0，不管投影向量的 z 坐标是否大于 1.0：

```
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，阴影映射对解析度的依赖很快变得很明显。



因为深度贴图有一个固定的解析度，多个片元对应于一个纹理像素。结果就是多个片元会从深度贴图的同一个深度值进行采样，这几个片元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解析度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再阴影中。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。

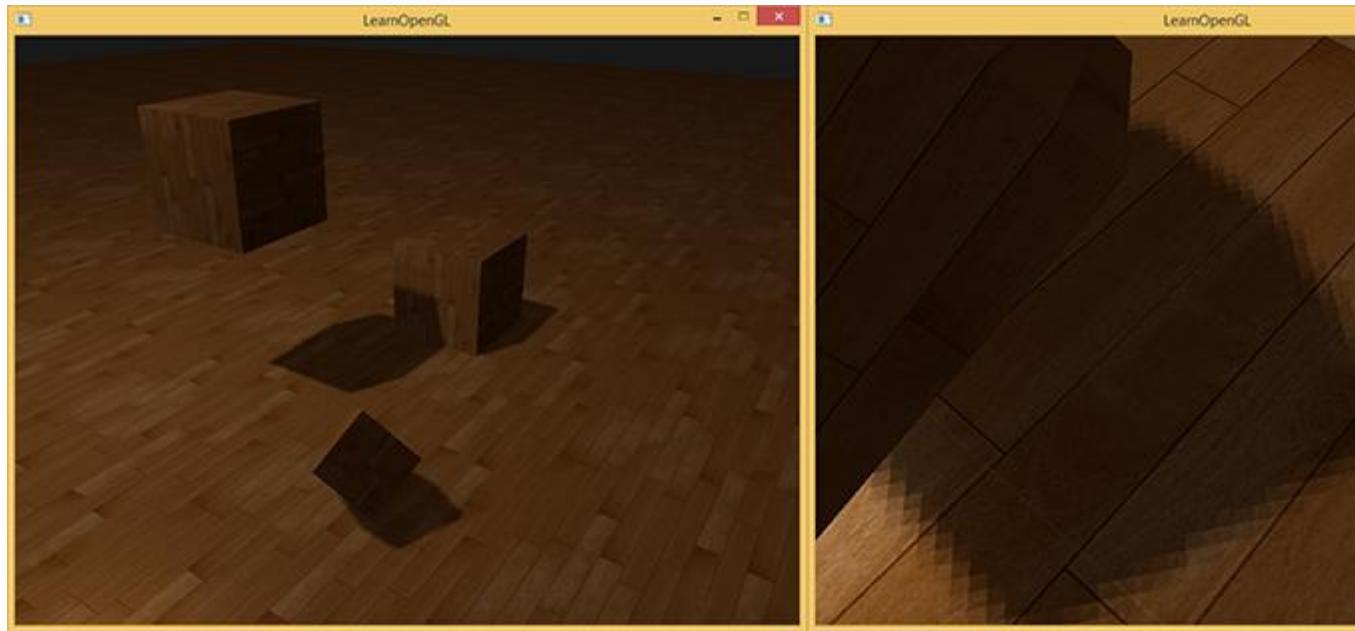
一个简单的 PCF 的实现是简单的从纹理像素四周对深度贴图采样，然后把结果平均起来：

```
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：



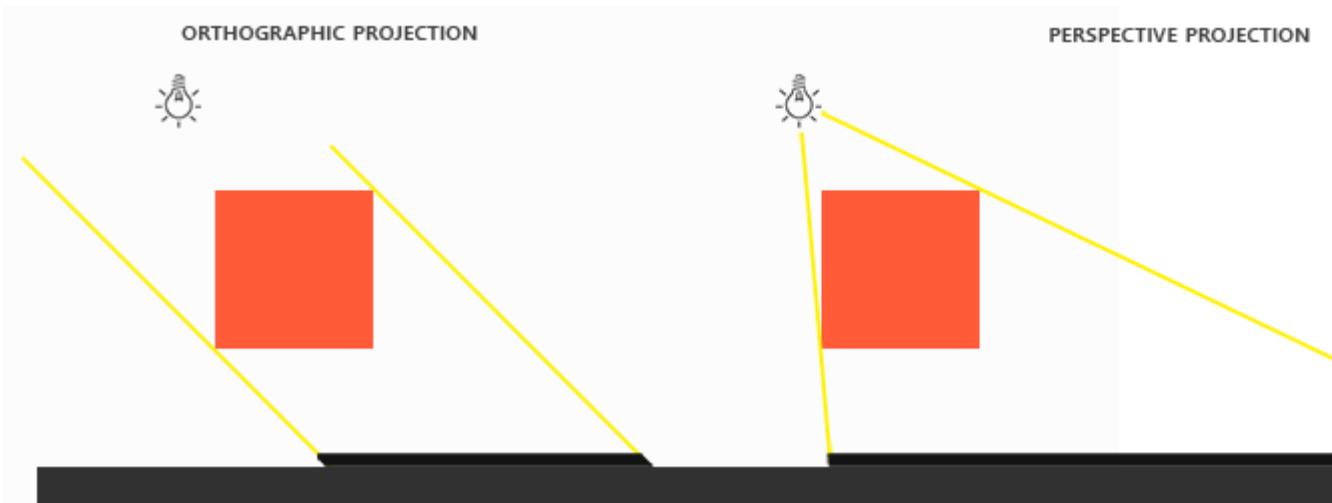
从稍微远一点的距离看去，阴影效果好多了，也不那么生硬了。如果你放大，仍会看到阴影贴图解析度的不真实感，但通常对于大多数应用来说效果已经很好了。

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，正交和投影矩阵之间有所不同。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接近于近平面。为了可以像使用正交投影一样合适的观察到深度值，你必须先讲过非线性深度值转变为线性的，我们在深度测试教程中已经讨论过。

```
#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: opengl-tutorial.org](#) 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。

Shadow Mapping 2

本文作者 JoeyDeVries，由 Django 翻译自 <http://learnopengl.com>

点光源阴影(Shadow Mapping)

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。

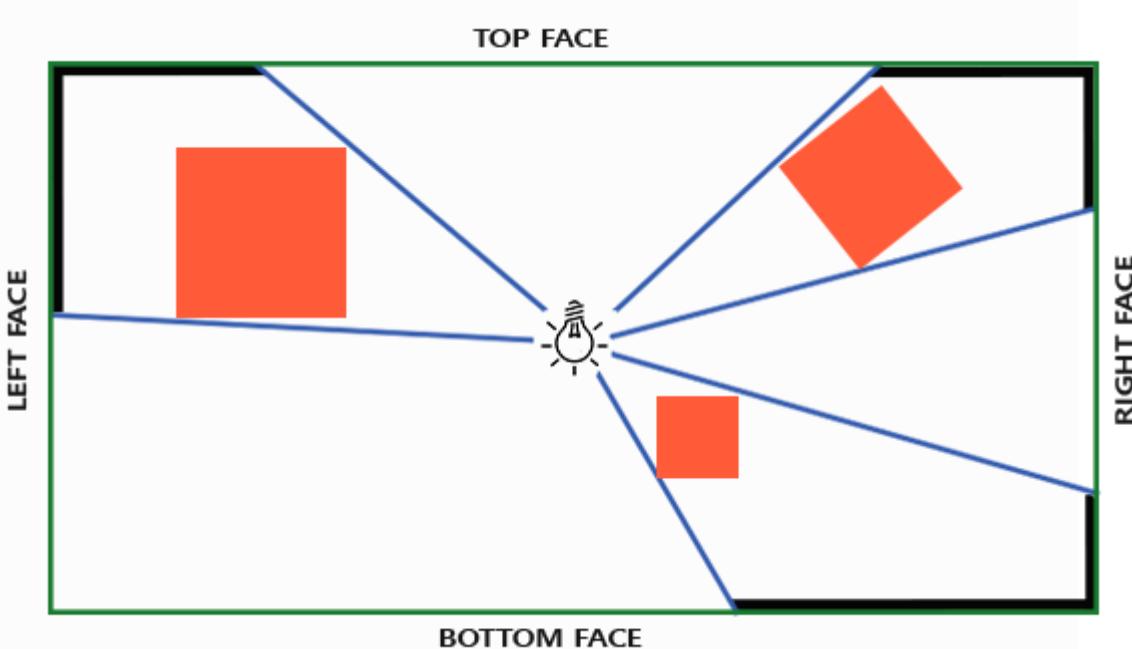
Important

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 `cubemap` 会怎样？因为 `cubemap` 可以储存 6 个面的环境数据，它可以将整个场景渲染到 `cubemap` 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 `cubemap` 被传递到光照像素着色器，它会用一个方向向量来采样 `cubemap`，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 `cubemap` 生成上稍微复杂一点。

生成深度 `cubemap`

为创建一个光周围的深度值的 `cubemap`，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 `cubemap` 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 cubemap。

首先，我们需要创建一个 cubemap：

```
GLuint depthCubemap;
```

```
glGenTextures(1, &depthCubemap);
```

然后生成 cubemap 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 `cubemap` 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 `cubemap` 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 `cubemap` 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 `cubemap` 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 `cubemap` 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 **cubemap** 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 **cubemap**，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(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 `cubemap` 的一个面，`cubemap` 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 `cubemap` 的的一个方向：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 `cubemap` 的面的一个方向。

这些变换矩阵发送到着色器渲染到 `cubemap` 里。

深度着色器

为了把值渲染到深度 `cubemap`，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 `cubemap` 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 cubemap 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 cubemap 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 `cubemap` 的 6 个面，我们每个面指定为一个输出面，把这个面的 `interger`（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 = gl_FragCoord.z;
```

```
}
```

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，`cubemap` 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 `cubemap`，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 `cubemap`，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 `cubemap` 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 `cubemap` 的森都。你可能已经从教程的 `cubemap` 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 `cubemap` 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会 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 `cubemap` 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。

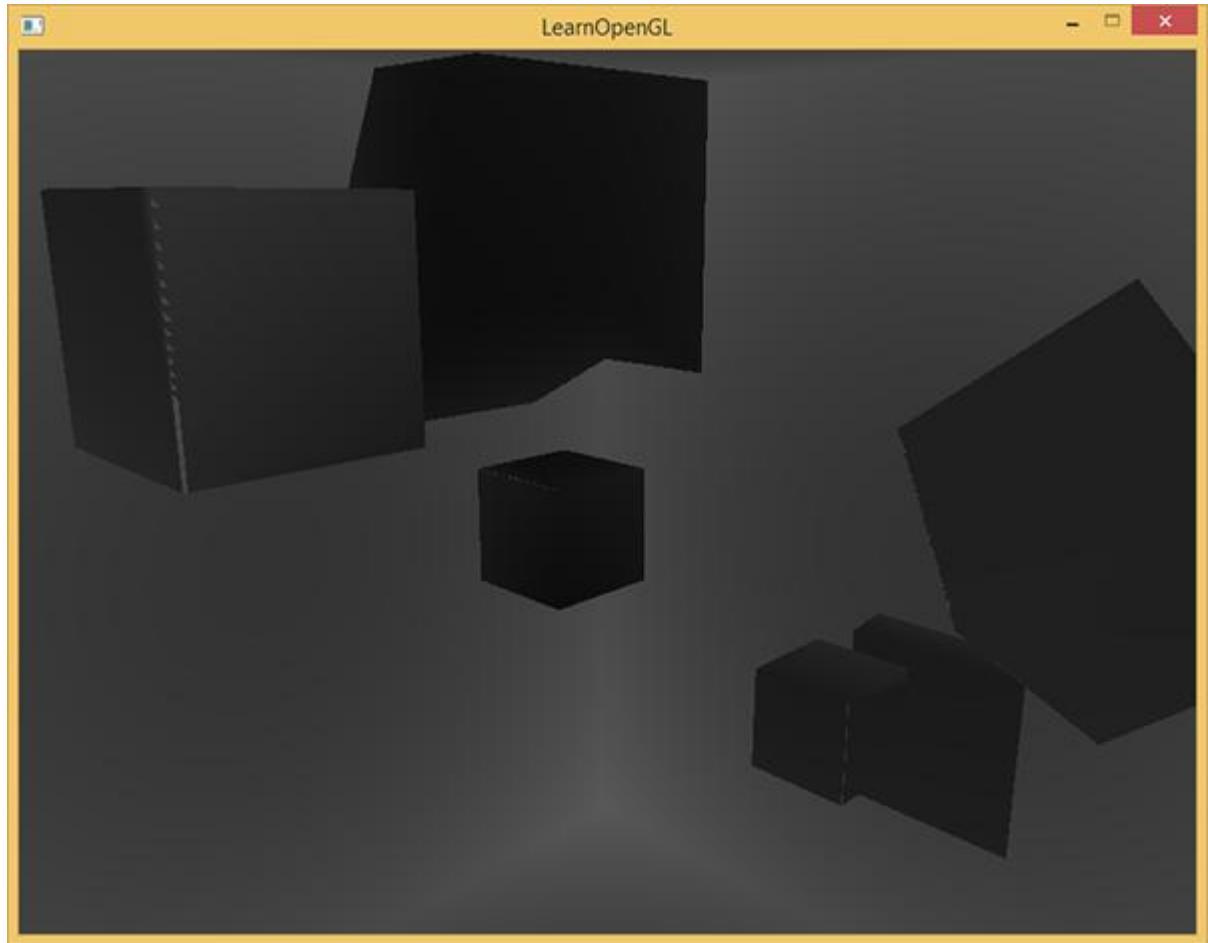
把 cubemap 深度缓冲显示出来

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 cubemap 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)

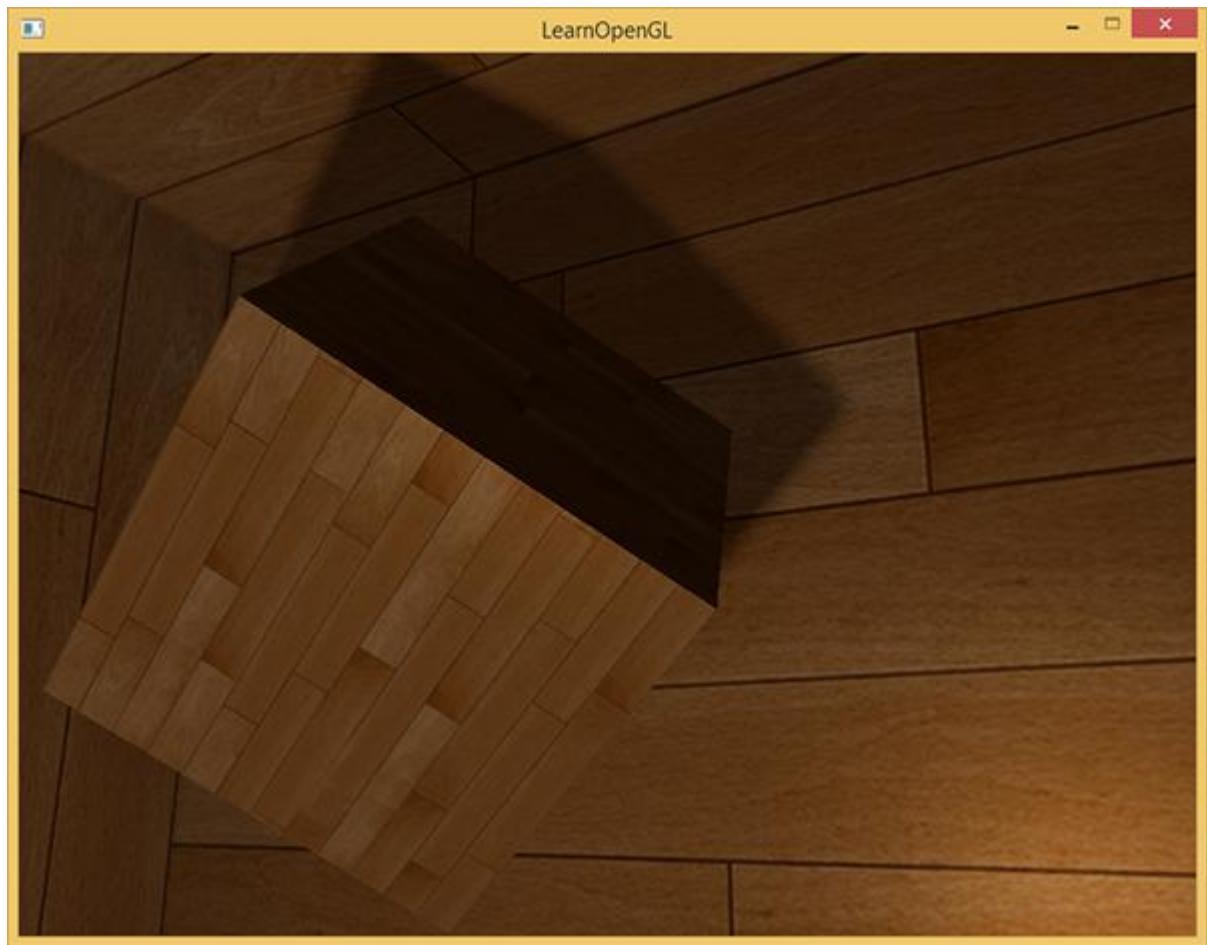
                shadow += 1.0;
        }
    }
}

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！

大多数这些样本都是多余的，它们在原始方向向量近处采样，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 `cubemap` 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)

```

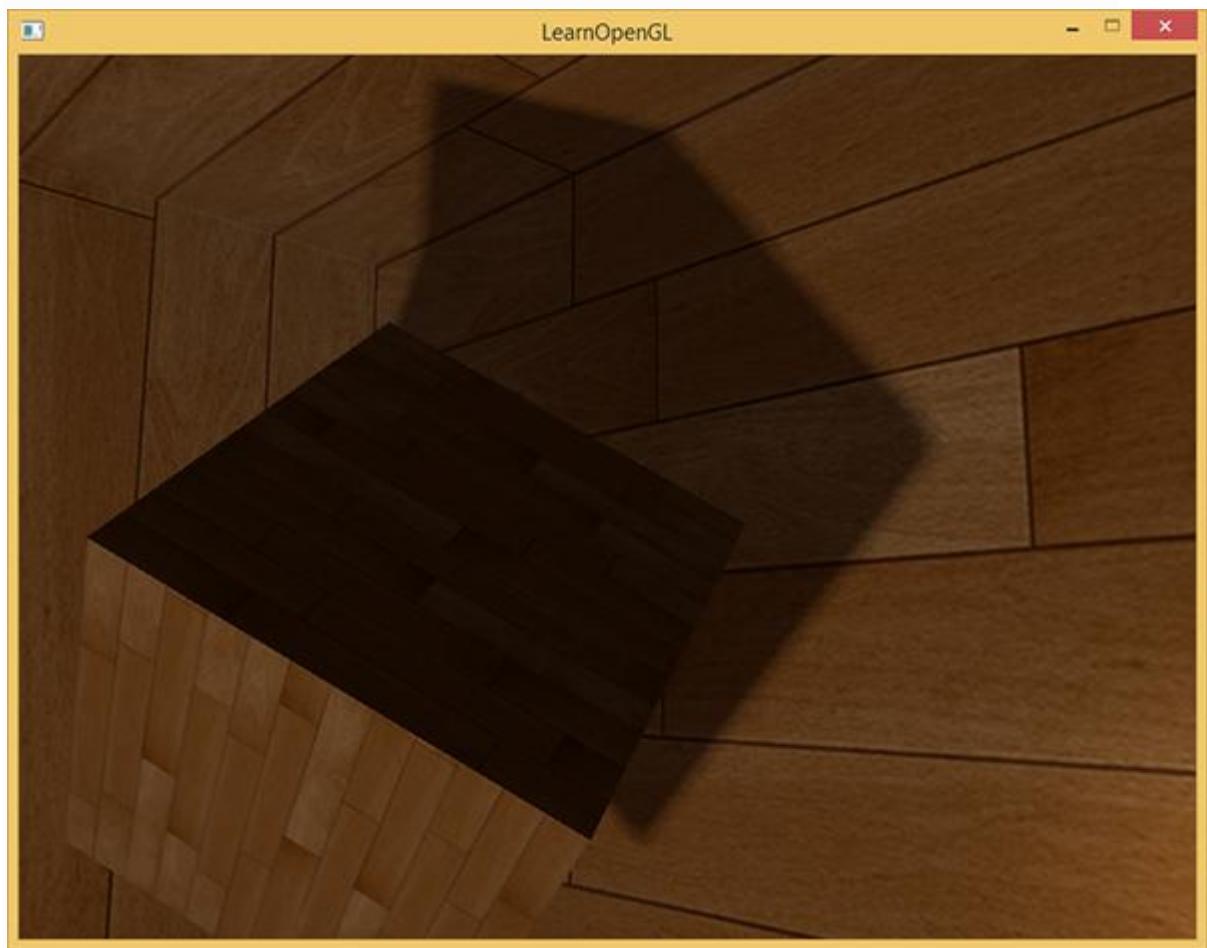
```
    shadow += 1.0;  
}  
  
shadow /= float(samples);
```

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 `cubemap` 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。

本文作者 JoeyDeVries，由 Django 翻译自 <http://learnopengl.com>

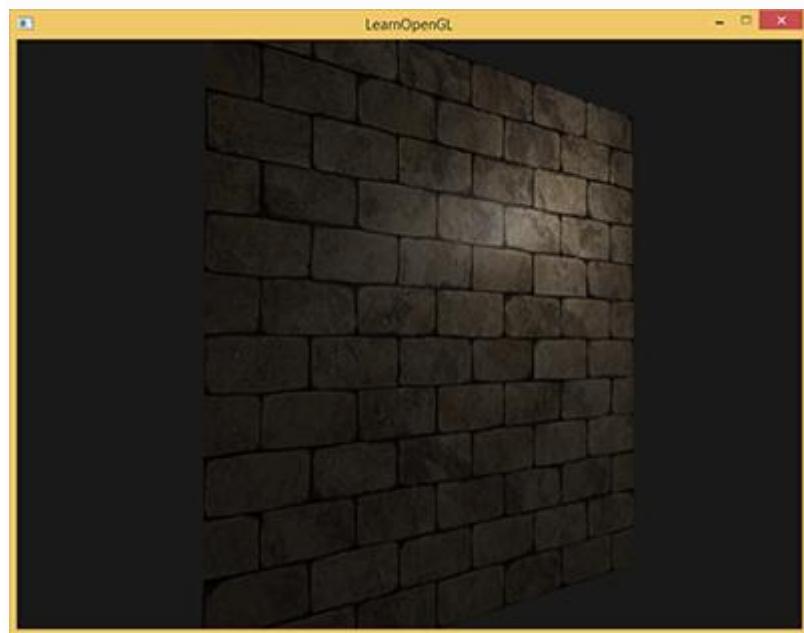
Normal Mapping

法线贴图（Normal Mapping）

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实。纹理确有助益，然而当你近看它们时，

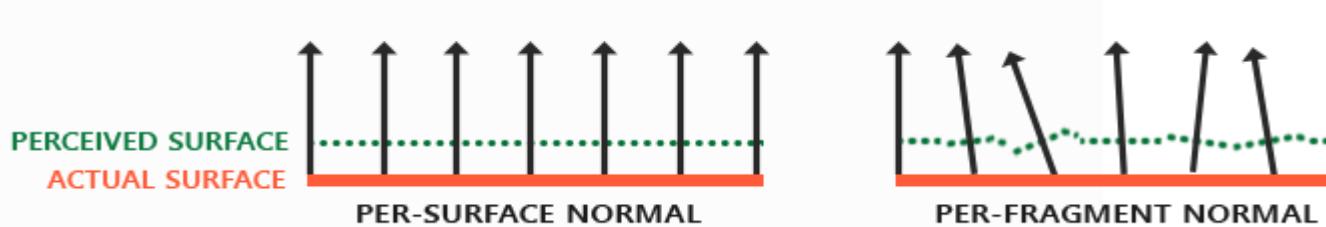
这个事实便隐藏不住了。现实中的物体表面并非是平坦的，而是表现出无数（凹凸不平的）细节。

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。

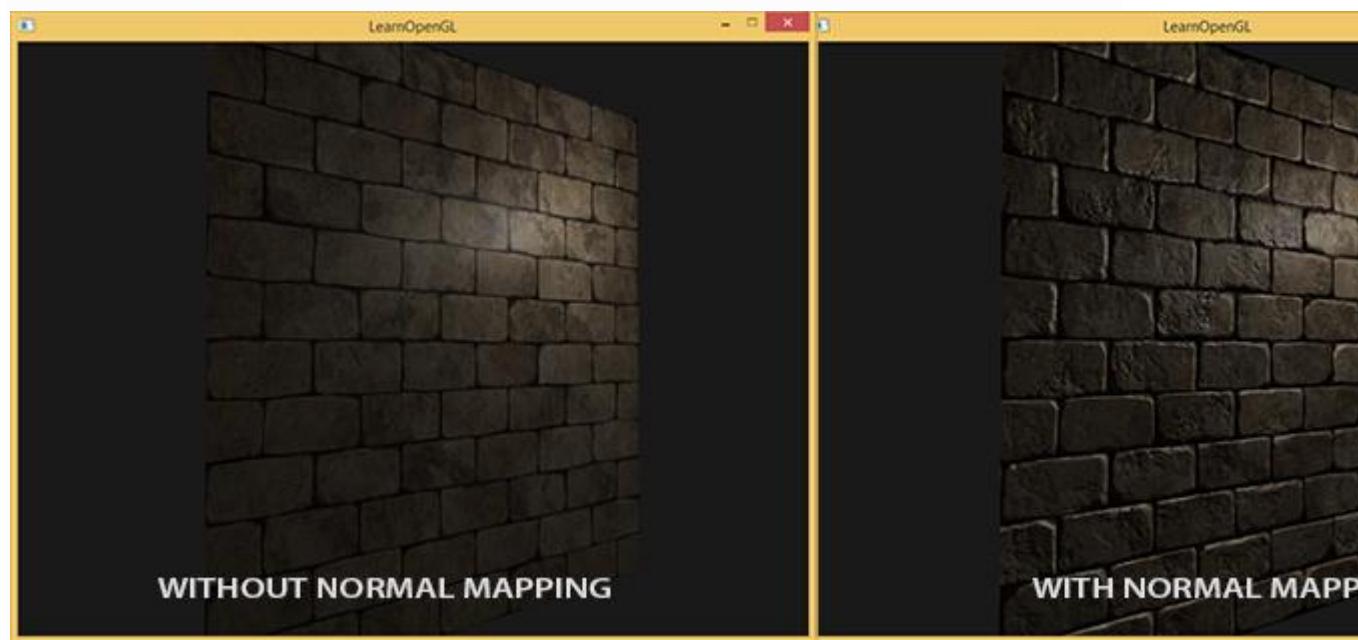


光照并没有呈现出任何裂痕和孔洞，完全忽略了砖块之间凹进去的线条；表面看起来完全就是平的。我们可以使用 **specular** 贴图根据深度或其他细节阻止部分表面被照的更亮，以此部分地解决问题，但这并不是一个好方案。我们需要的是某种可以告知光照系统给所有有关物体表面类似深度这样的细节的方式。

如果我们一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。

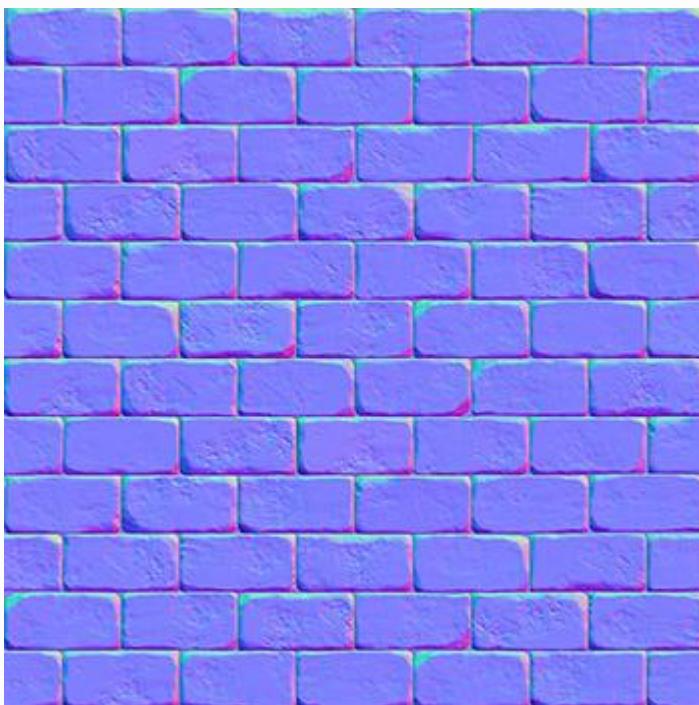
法线贴图

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：

```
1 vec3 rgb_normal = normal * 0.5 - 0.5; // transforms from [-1,1] to [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：



这会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：

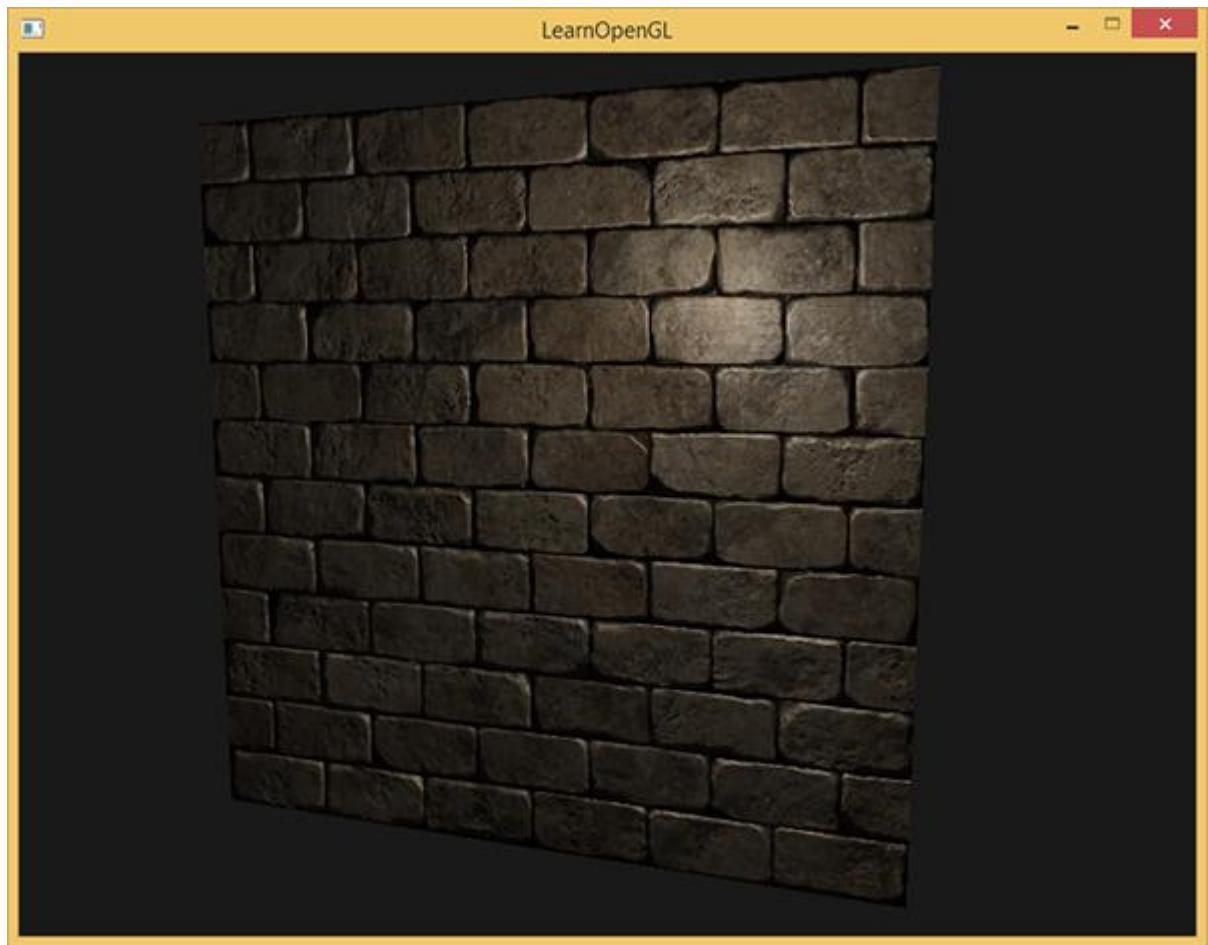
```
uniform sampler2D normalMap;

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
}
```

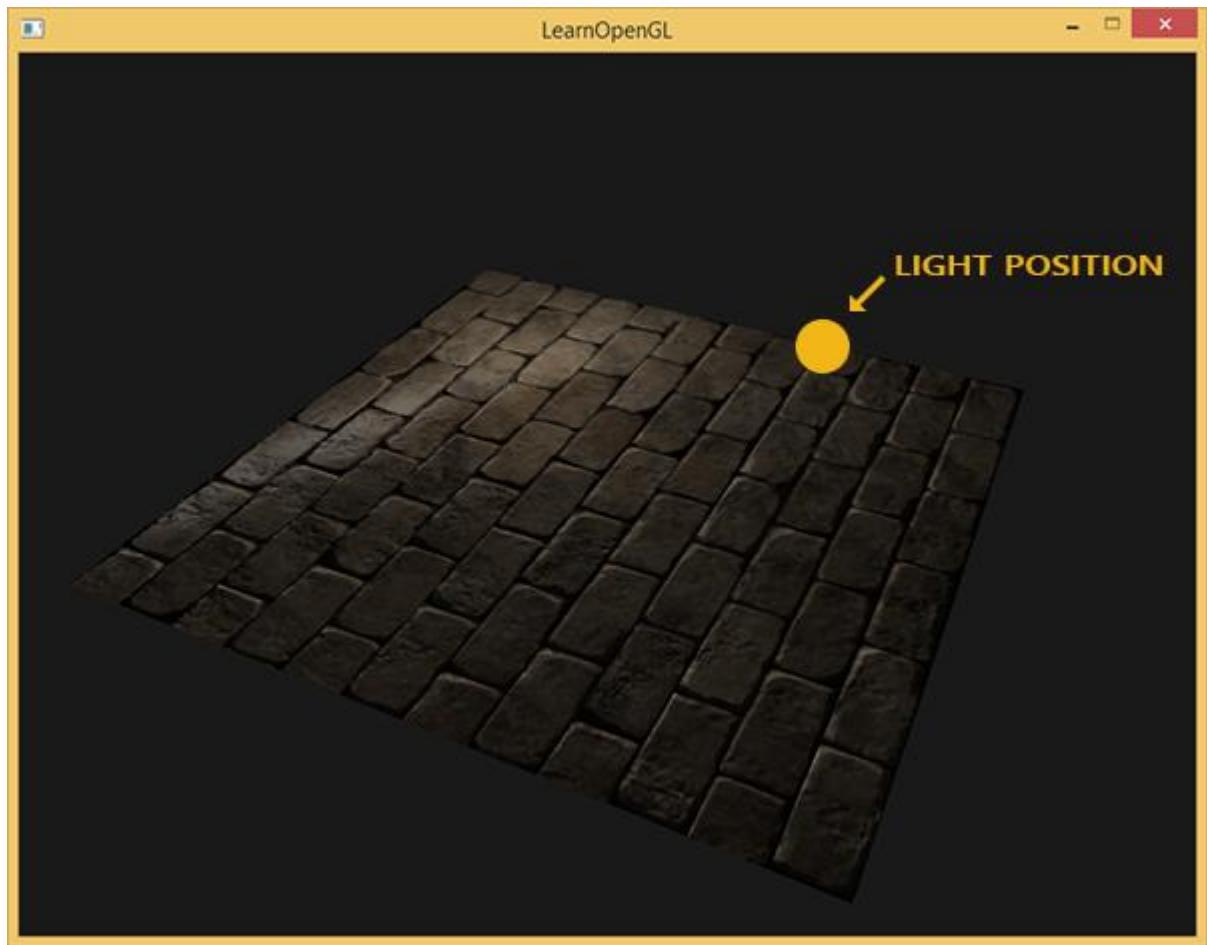
这里我们将被采样的法线颜色从 0 到 1 重新映射回-1 到 1，便能将 RGB 颜色重新处理成法线，然后使用采样出的法线向量应用于光照的计算。在例子中我们使用的是 Blinn-Phong 着色器。

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：

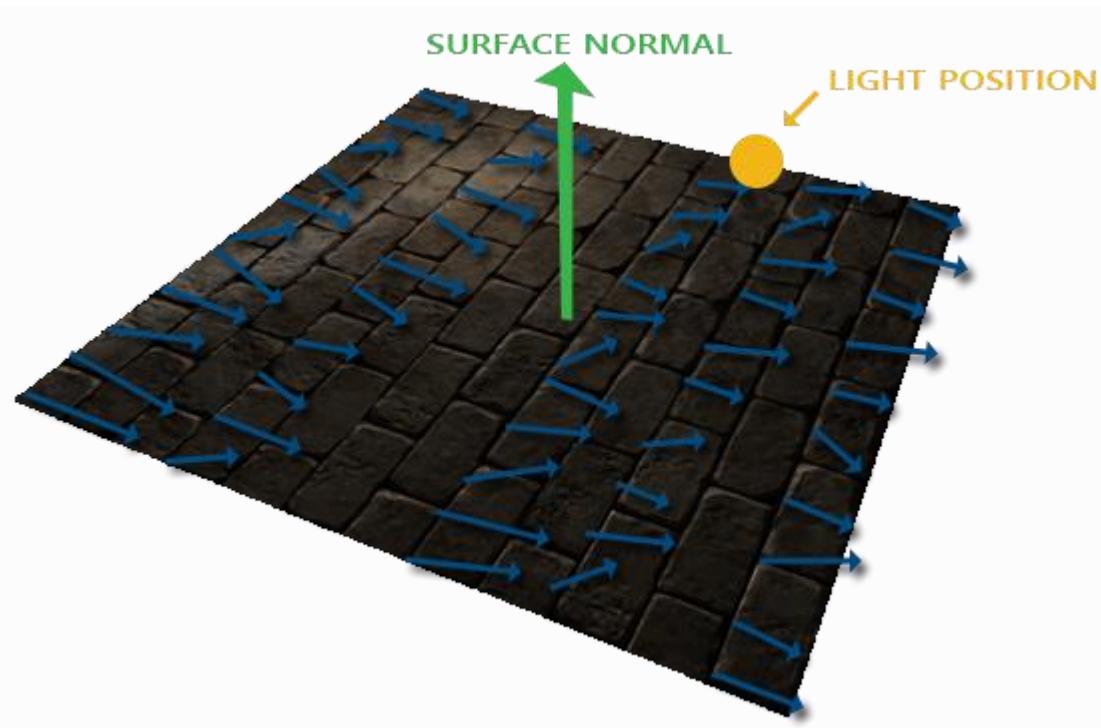


你可以在这里找到这个简单 **demo** 的源代码及其顶点和像素着色器。

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**）。

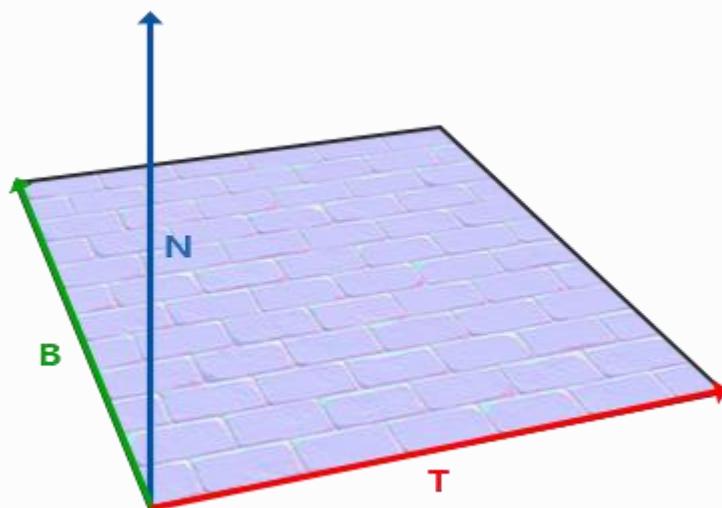
切线空间

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坐标，使它们转向到最终的贴图表面的方向。

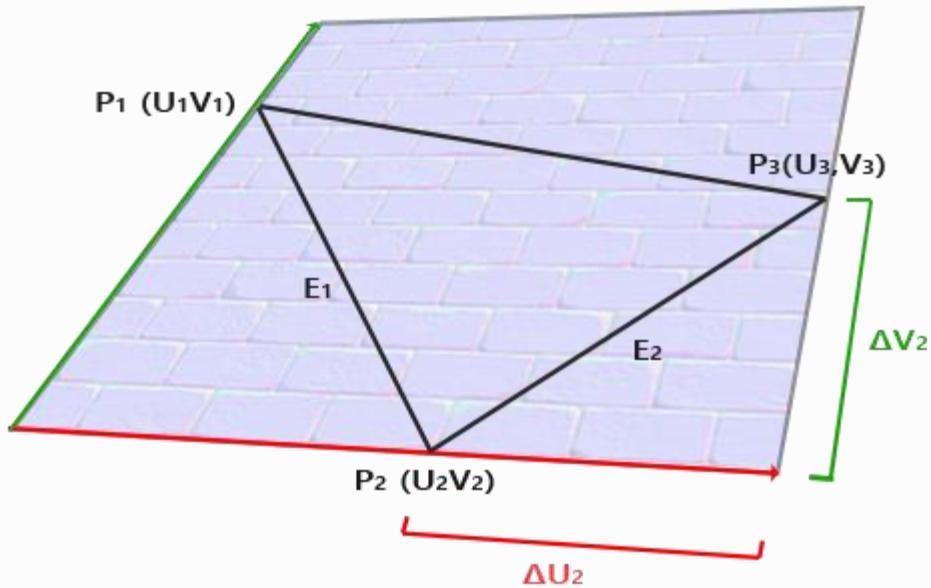
我们可以说，上个部分那个朝向正 y 的法线贴图错误的贴到了表面上。法线贴图被定义在切线空间中，所以一种解决问题的方式是计算出一种矩阵，把法线从切线空间变换到一个不同的空间，这样它们就能和表面法线方向对齐了：法线向量都会指向正 y 方向。切线空间的一大好处是我们可以为任何类型的表面计算出一个这样的矩阵，由此我们可以把切线空间的 z 方向和表面的法线方向对齐。

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。右和前向量是切线和副切线向量。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：



上图中我们可以看到边 E_2 纹理坐标的不同， E_2 是一个三角形的边，这个三角形的另外两条边是 ΔU_2 和 ΔV_2 ，它们与切线向量 T 和副切线向量 B 方向相同。这样我们可以把边和 E_1 和 E_2 用切线向量 T 和副切线向量 B 的线性组合表示出来（译注：注意 T 和 B 都是单位长度，在 TB 平面上所有点的 T 、 B 坐标都在 0 到 1 之间，因此可以进行这样的组合）：

$$E_1 = \Delta U_1 T + \Delta V_1 B$$

$$E_2 = \Delta U_2 T + \Delta V_2 B$$

我们也可以写成这样：

$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1(T_x, T_y, T_z) + \Delta V_1(B_x,$$

$$B_y, B_z)$$

E 是两个向量位置的差， U 和 V 是纹理坐标的差。然后我们得到两个未知数（切线 T 和副切线 B ）和两个等式。你可能想起你的代数课了，这是让我们去接 T 和 B 。

上面的方程允许我们把它们写成另一种格式：矩阵乘法

$$[E_{1x} E_{1y} E_{1z} E_{2x} E_{2y} E_{2z}] = [\Delta U_1 \Delta V_1 \Delta U_2 \Delta V_2] [T_x T_y T_z B_x B_y B_z]$$

尝试会以一下矩阵乘法，它们确实是同一种等式。把等式写成矩阵形式的好处是，解 T 和 B 会因此变得很容易。两边都乘以 $\Delta U \Delta V$ 的反数等于：

$$[\Delta U_1 \Delta V_1 \Delta U_2 \Delta V_2] - 1 [E_1x E_1y E_1z E_2x E_2y E_2z] = [T_x T_y T_z B_x B_y B_z]$$

这样我们就可以解出 T 和 B 了。这需要我们计算出 **delta** 纹理坐标矩阵的拟阵。我不打算讲解计算逆矩阵的细节，但大致是把它变化为，1 除以矩阵的行列式，再乘以它的共轭矩阵。

$$[T_x T_y T_z B_x B_y B_z] = 1 \Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1 [\Delta V_2 - \Delta V_1 - \Delta U_2 \Delta U_1] [E_1x E_1y E_1z E_2x E_2y E_2z]$$

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 T 和副切线 B 。

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：

```
1 2 3 4 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);
```

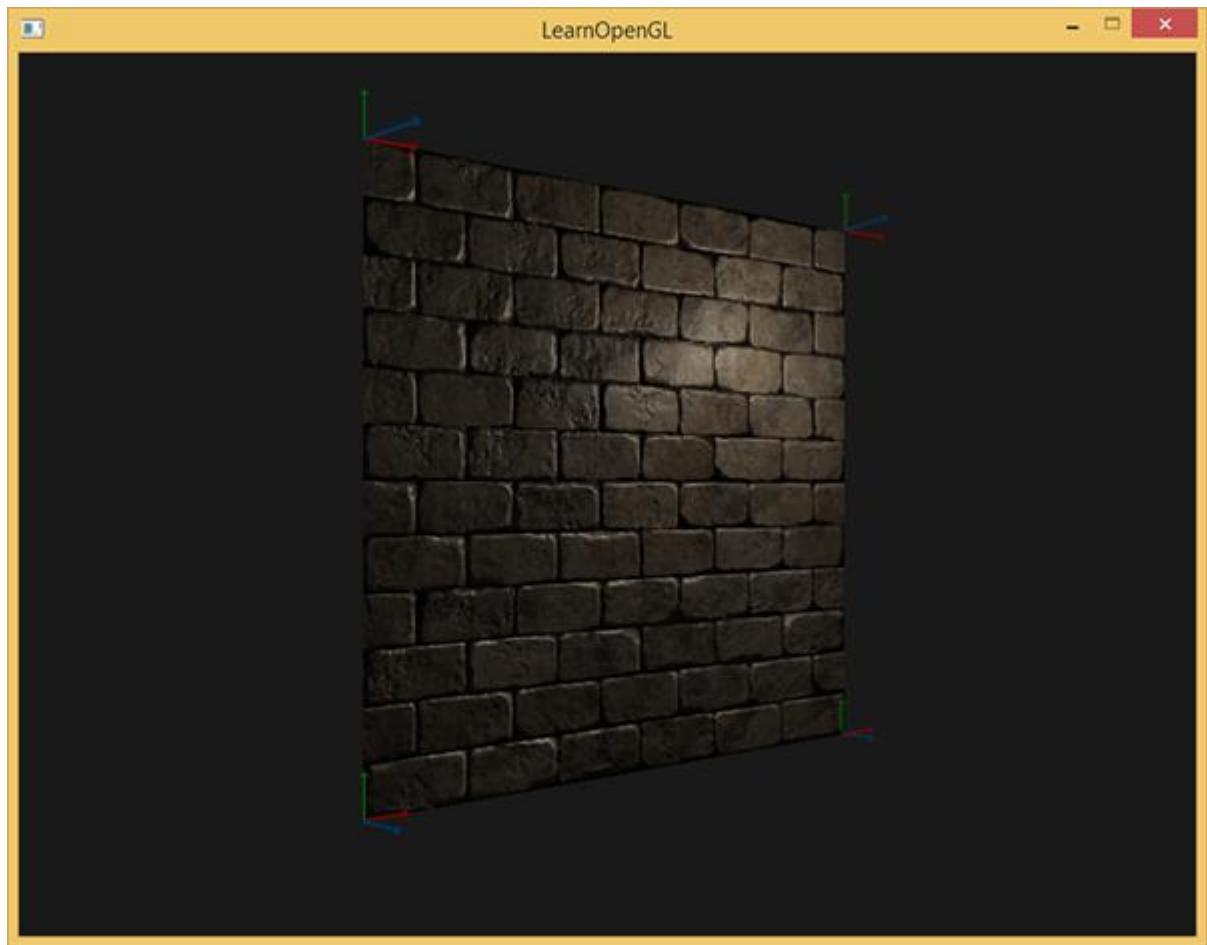
```
[...] // similar procedure for calculating tangent/bitangent for
```

```
plane's second triangle
```

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讲 **TBN** 向量乘以 **model** 矩阵，而是使用法线矩阵，但我们只关心向量的方向，不会平移也和缩放这个变换。

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庸 **T** 和 **N** 向量的叉乘，自己计算出副切线：**vec3 B = cross(T, N);** 现在我们有了 **TBN** 矩阵，如果来使用它呢？基本有两种方式可以使用，我们会把这两种方式都说明一下：

我们可以用 **TBN** 矩阵把所有向量从切线空间转到世界空间，传给像素着色器，然后把采样得到的法线用 **TBN** 矩阵从切线空间变换到世界空间；法线就处于和其他光照变量一样的空间中了。我们用 **TBN** 的逆矩阵把所有世界空间的向量转换到切线空间，使用这个矩阵将除法线以外的所有相关光照变量转换到切线空间中；这样法线也能和其他光照变量处于同一空间之中。我们来看看第一种情况。我们从法线贴图重采样得来的法线向量，是以切线空间表达的，尽管其他光照向量是以世界空间表达的。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。这个属性和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。

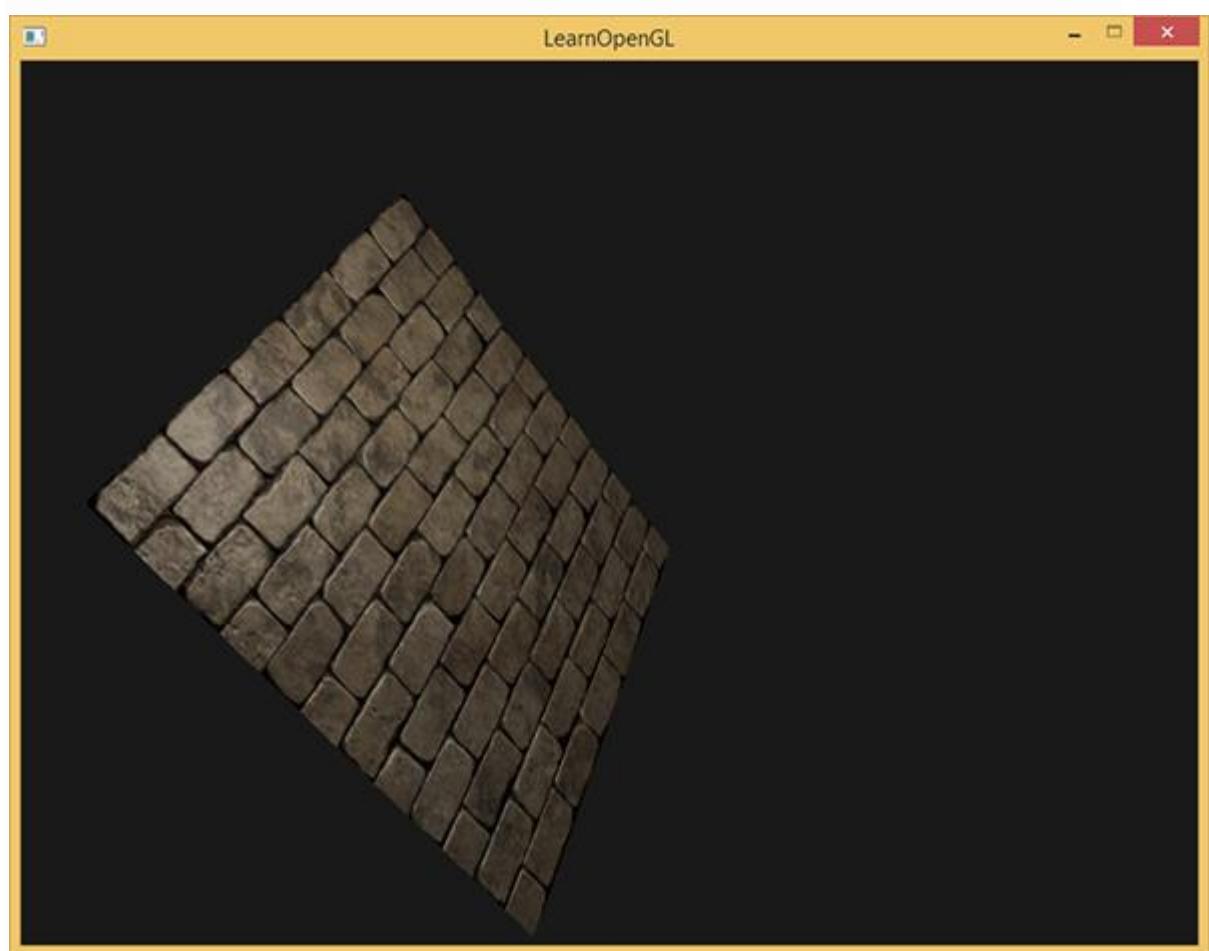
```
out VS_OUT {  
    vec3 FragPos;  
    vec2 TexCoords;  
    vec3 TangentLightPos;  
    vec3 TangentViewPos;  
    vec3 TangentFragPos;  
} vs_out;  
  
uniform vec3 lightPos;  
uniform vec3 viewPos;  
  
[...]  
  
void main()  
{  
    [...]  
    mat3 TBN = transpose(mat3(T, B, N));  
    vs_out.TangentLightPos = 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, 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](#)。

复杂的物体

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"  
)
```

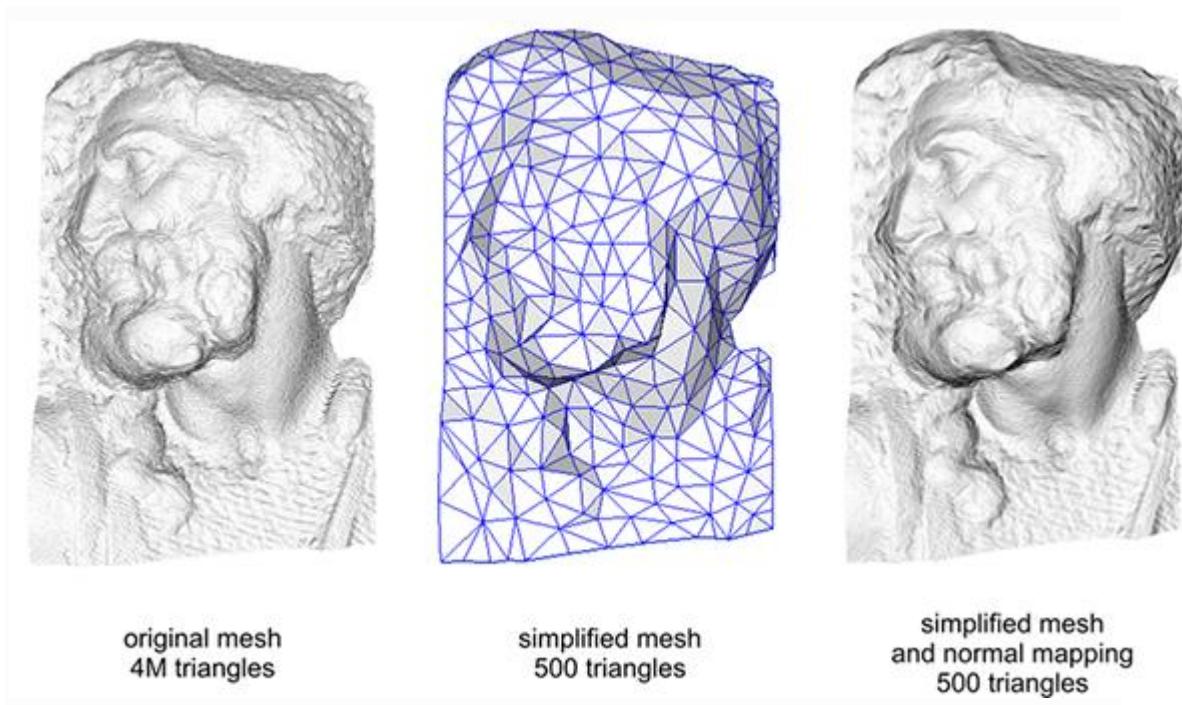
当然，对于每个模型的类型和文件格式来说都是不同的。同样了解 `aiProcess_CalcTangentSpace` 并不能总是很好的工作也很重要。计算切线是需要根据纹理坐标的，有些模型制作者使用一些纹理小技巧比如镜像一个模型上的纹理表面时也镜像了另一半的纹理坐标；这样当不考虑这个镜像的特别操作的时候（Assimp 就不考虑）结果就不对了。

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，当发下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(tangent, 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

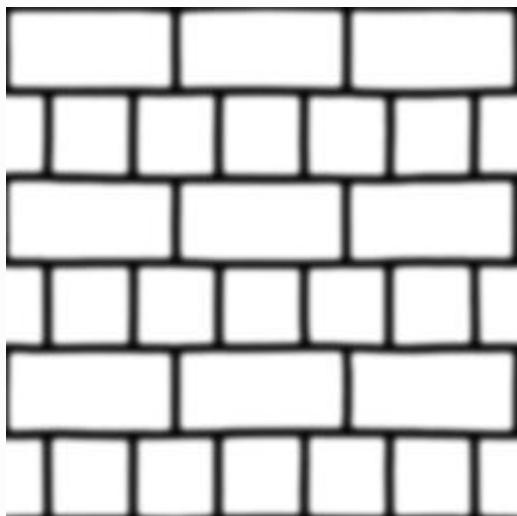
- [Tutorial 26: Normal Mapping](#): ogldev 的法线贴图教程。
- [How Normal Mapping Works](#): TheBennyBox 的讲述法线贴图如何工作的视频。
- [Normal Mapping Mathematics](#): TheBennyBox 关于法线贴图的数学原理的教程。
- [Tutorial 13: Normal Mapping](#): opengl-tutorial.org 提供的法线贴图教程。
 - 本文作者 JoeyDeVries, 由 Django 翻译自 <http://learnopengl.com>
 -
-
-

Parallax Mapping 视差贴图

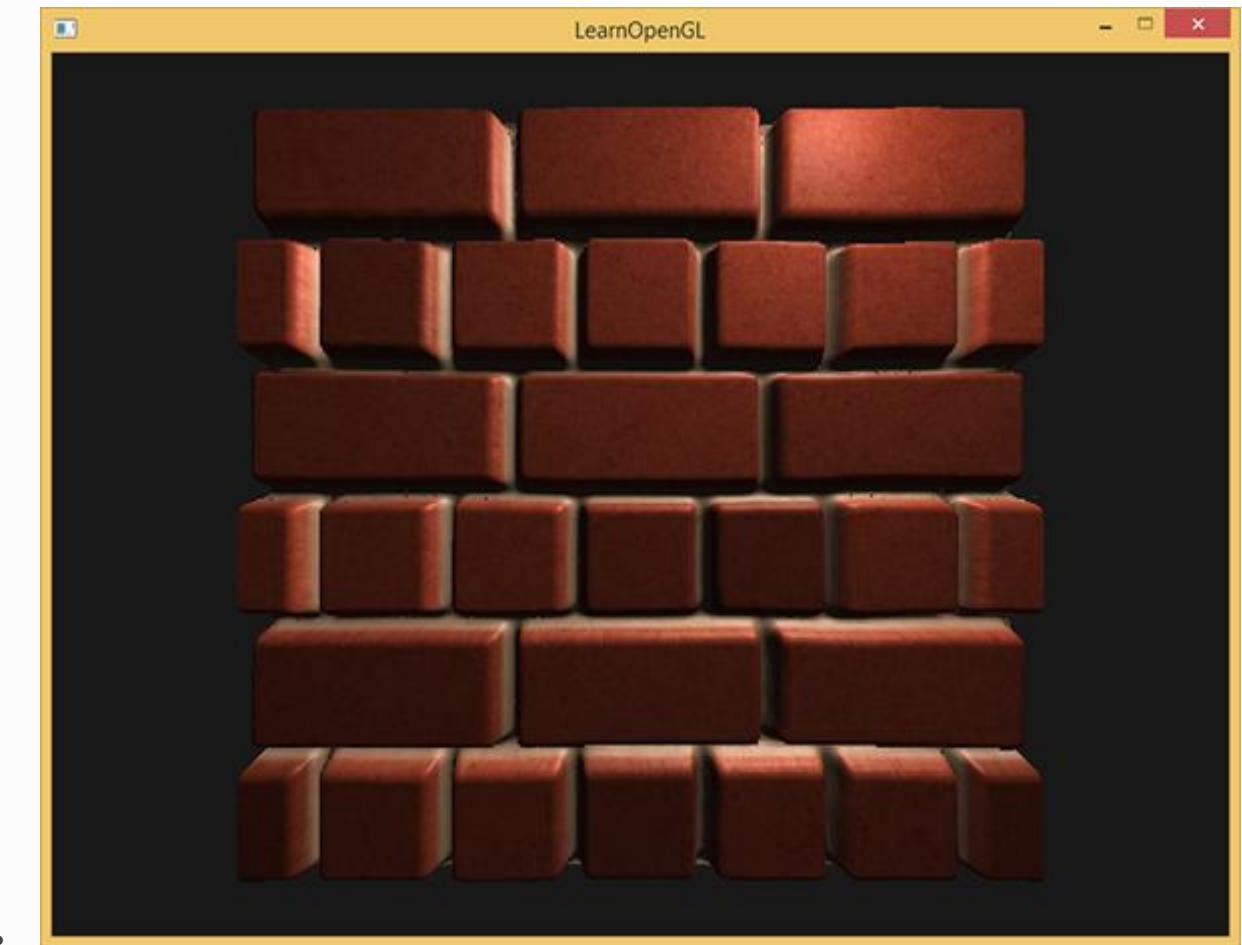
视差贴图(Parallax Mapping)

- 视差贴图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（译注：displacement mapping 也叫置换贴图）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，更具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告诉贴图如下所示：

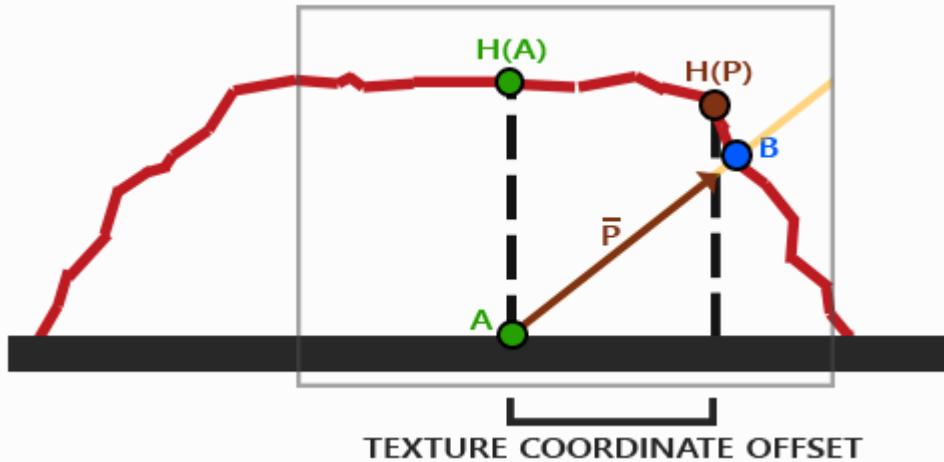


- 整个平面上的每个顶点都根据从高度贴图采样出来的高度值进行位移，根据材质的几何属性平坦的平面变换成凹凸不平的表面。例如一个平坦的平面利用上面的高度贴图进行置换能得到以下结果：

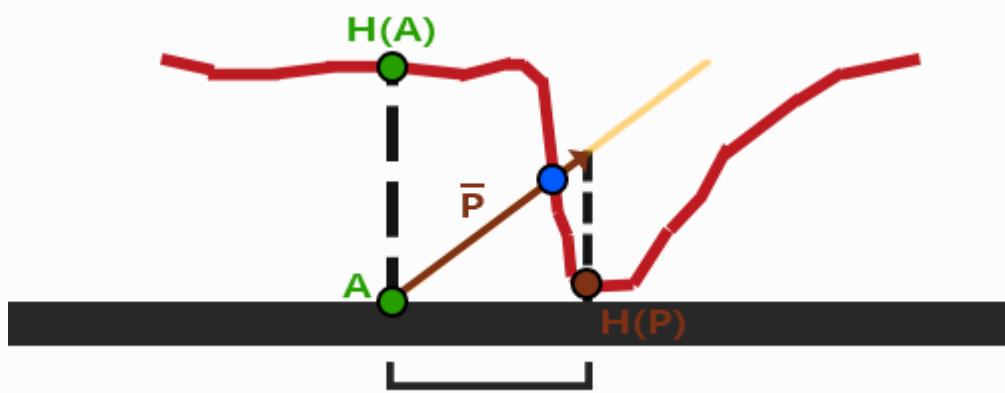


- 置换顶点有一个问题就是平面必须由很多顶点组成才能获得具有真实感的效果，否则看起来效果并不会很好。一个平坦的表面上有 1000 个顶点计算量太大了。我们能否不用这么多的顶点就能取得相似的效果呢？事实上，上面的表面就是用 6 个顶点渲染出来的（两个三角形）。上面的那个表面使用视差贴图技术渲染，位移贴图技术不需要额外的顶点数据来表达深度，它像法线贴图一样采用一种聪明的手段欺骗用户的眼睛。
- 视差贴图背后的思想是修改纹理坐标使一个 **fragment** 的表面看起来比实际的更高或者更低，所有这些都根据观察方向和高度贴图。为了理解它如何工作，看看下面砖块表面的图片：
- 这里粗糙的红线代表高度贴图中的数值的立体表达，向量 **V** 代表观察方向。如果平面进行实际位移，观察者会在点 **B** 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 **A** 与平面接触。视差贴图的目的是，在 **A** 位置上的 **fragment** 不再使用点 **A** 的纹理坐标而是使用点 **B** 的。随后我们用点 **B** 的纹理坐标采样，观察者就像看到了点 **B** 一样。

- 这个技巧就是描述如何从点 A 得到点 B 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 V 进行缩放的方式解决这个问题，缩放的大小是 A 处 **fragment** 的高度。所以我们将 V 的长度缩放为高度贴图在点 A 处 $H(A)$ 采样得来的值。下图展示了经缩放得到的向量 \bar{P} :



- 我们随后选出 \bar{P} 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 \bar{P} 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 B 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 \bar{P} 最终不会和 B 接近，就像下图这样：

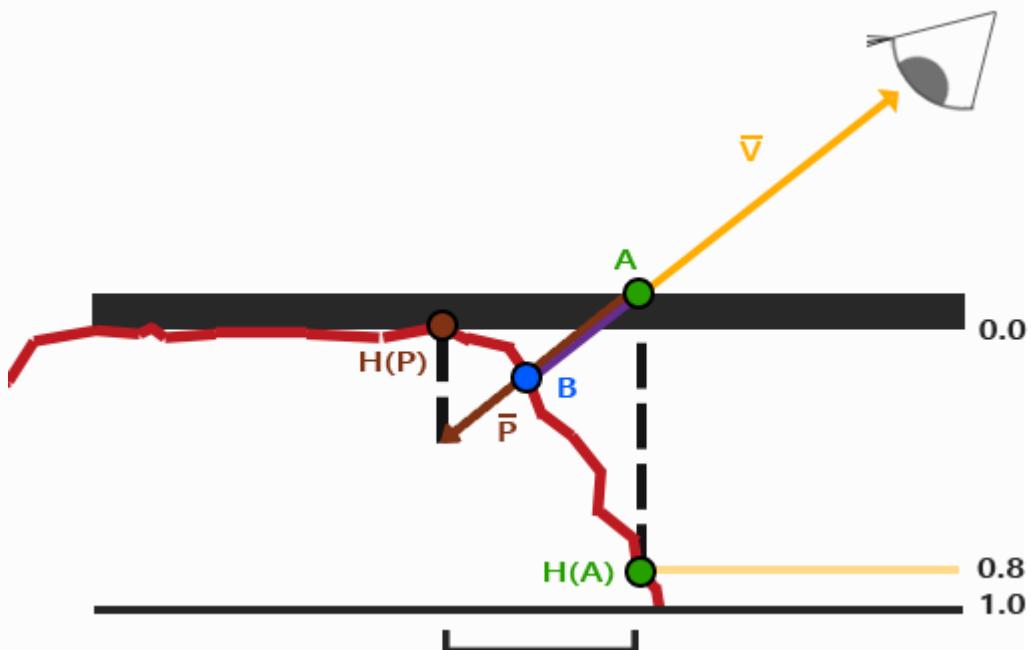


- 视差贴图的另一个问题是，当表面被任意旋转以后很难指出从 \bar{P} 获取哪一个坐标。我们在视差贴图中使用了另一个坐标空间，这个空间 \bar{P} 向量的 x 和 y 元素总是与纹理表面对齐。如果你看了法线贴图教程，你也许猜到了，我们实现它的方法，是的，我们还是在切线空间中实现视差贴图。

- 将 fragment 到观察者的向量 V 转换到切线空间中，经变换的 P 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 P 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 A 和 B ，但是这次我们用向量 V 减去点 A 的纹理坐标得到 P 。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 `V` 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：

```

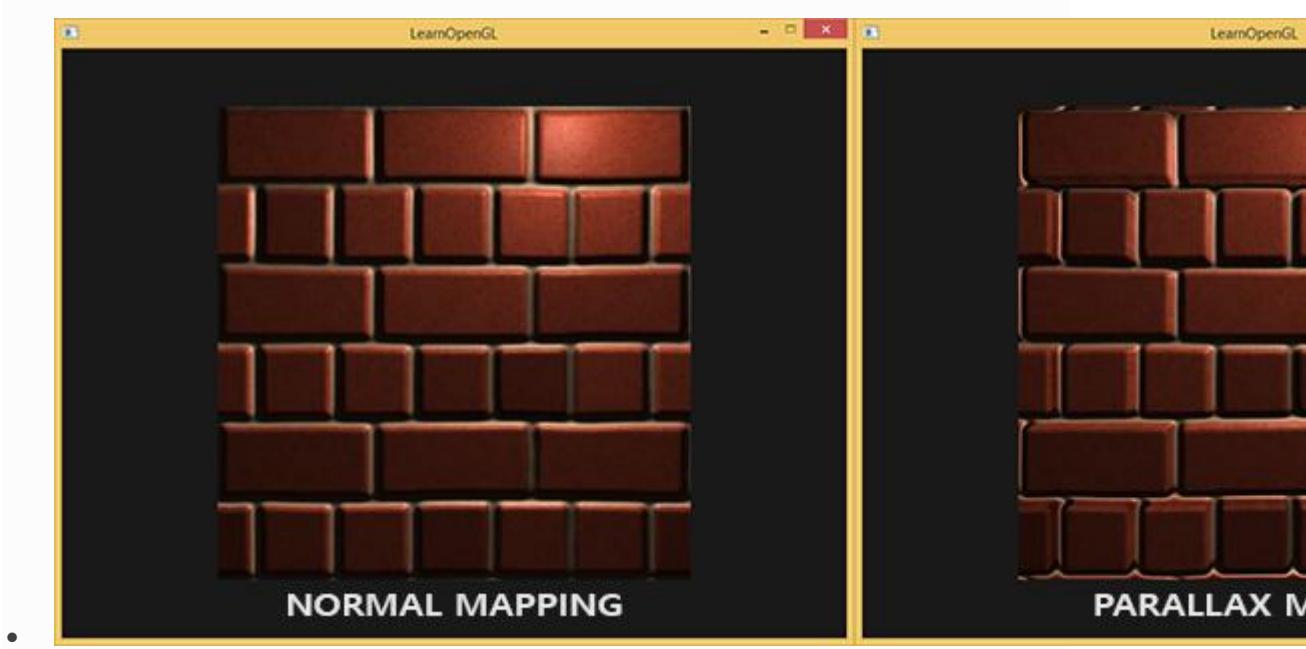
• vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)

• {

•     float height = texture(depthMap, texCoords).r;
```

```
•     vec3 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出当前 `fragment` 高度 $H(A)$ 。然后计算出 P , 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。我们同时引入额一个 `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 P 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 P 向量。所以基本上我们增加了 P 的大小, 当以一个角度朝向一个表面相比朝向顶部时它对纹理坐标会进行更大程度的缩放; 这回在角上获得更大的真实度。
- 有些人更喜欢在等式中不使用 `viewDir.z`, 因为普通的视差贴图会在角上产生不想要的结果; 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:



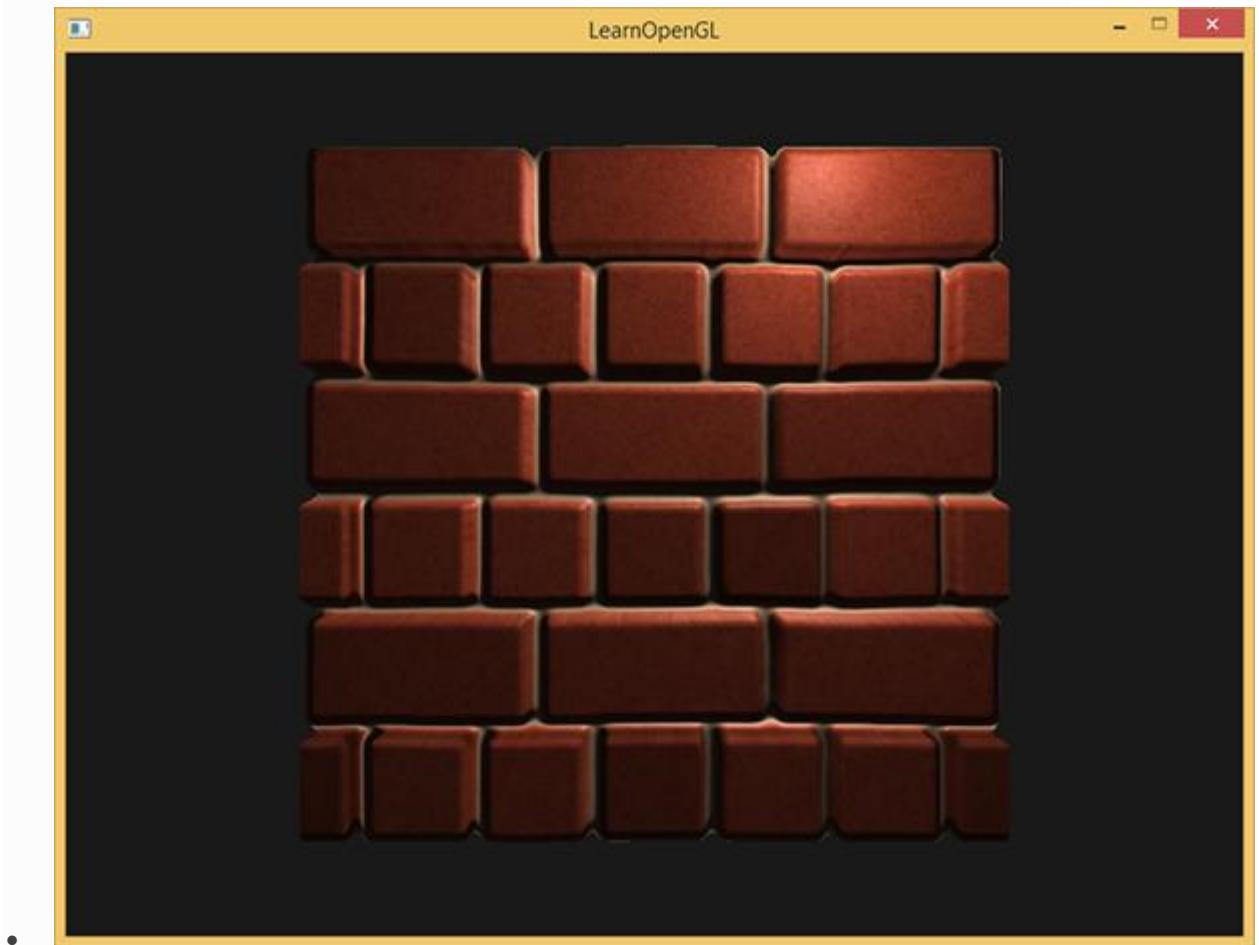
- 这里你会看到只用法线贴图和与视差贴图相结合的法线贴图的不同之处。因为视差贴图尝试模拟深度，它实际上能够根据你观察它们的方向使砖块叠加到其他砖块上。
- 在视差贴图的那个平面里你仍然能看到在边上有古怪的失真。原因是在平面的边缘上，纹理坐标超出了 0 到 1 的范围进行采样，根据纹理的环绕方式导致了不真实的结果。解决的方法是当它超出默认纹理坐标范围进行采样的时候就丢弃这个 fragment:

- ```

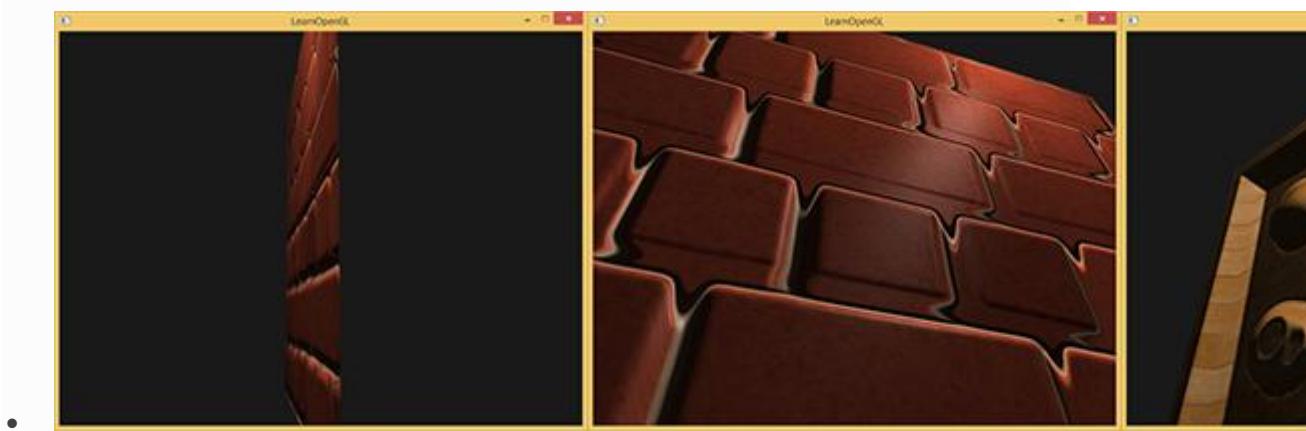
• texCoords = ParallaxMapping(fs_in.TexCoords, viewDir);

• 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：



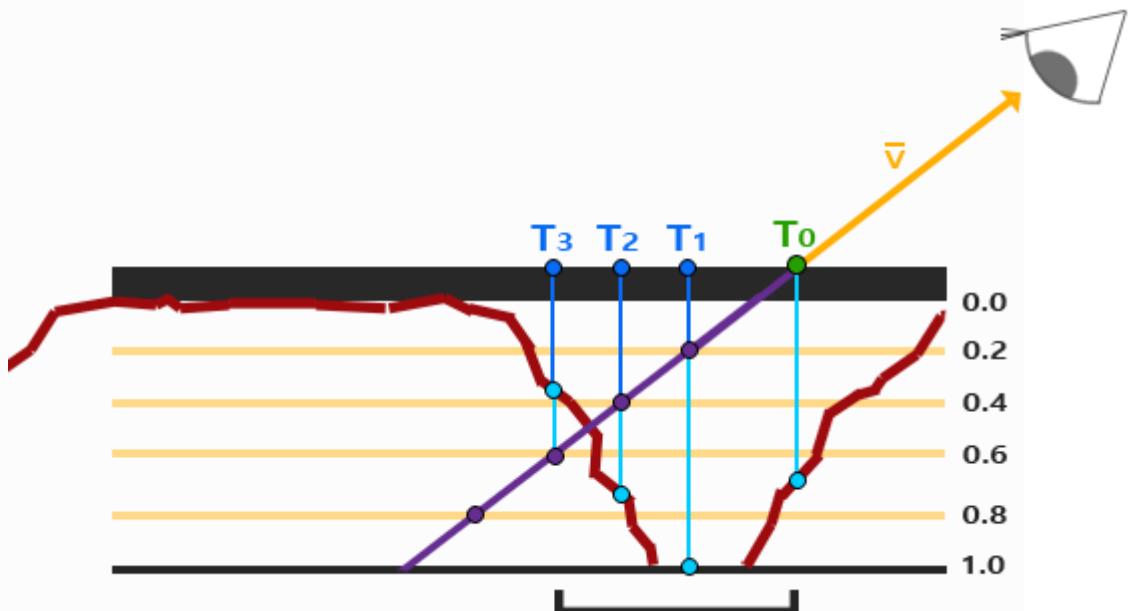
- 你可以在这里找到[源代码](#), 以及[顶点](#)和[像素](#)着色器。
- 看起来不错, 运行起来也很快, 因为我们只要给视差贴图提供一个额外的纹理样本就能工作。当从一个角度看过去的时候, 会有一些问题产生 (和法线贴图相似), 陡峭的地方会产生不正确的结果, 从下图你可以看到:



- 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 **B** 会得到怎样的结果？

## • 陡峭视差映射（Steep Parallax Mapping）

- 陡峭视差映射是视差映射的扩展，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 **P** 到 **B**。它能得到更好的结果，它将总深度范围分布到同一个深度/高度的多个层中。从每个层中我们沿着 **P** 方向移动采样纹理坐标，直到我们找到了一个采样得到的低于当前层的深度值的深度值。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 **P** 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高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 **P** 是可用的位移几何位置。我们可以用从向量 **P3** 的纹理坐标偏移 **T3** 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 `P` 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  $\mathbf{P}$  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。

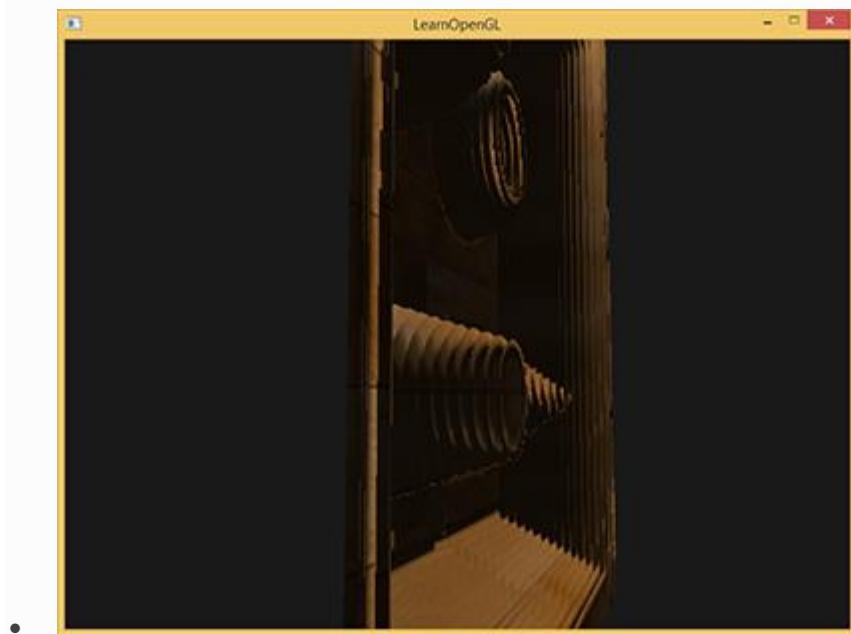
- 有 10 个样本砖墙从一个角度看上去就已经很好了，但是当有一个强前面展示的木制表面一样陡峭的表面时，陡峭的视差映射的威力就显示出来了：



- 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  $B$  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，所以我们将下面讨论一下。

## • Parallax Occlusion Mapping

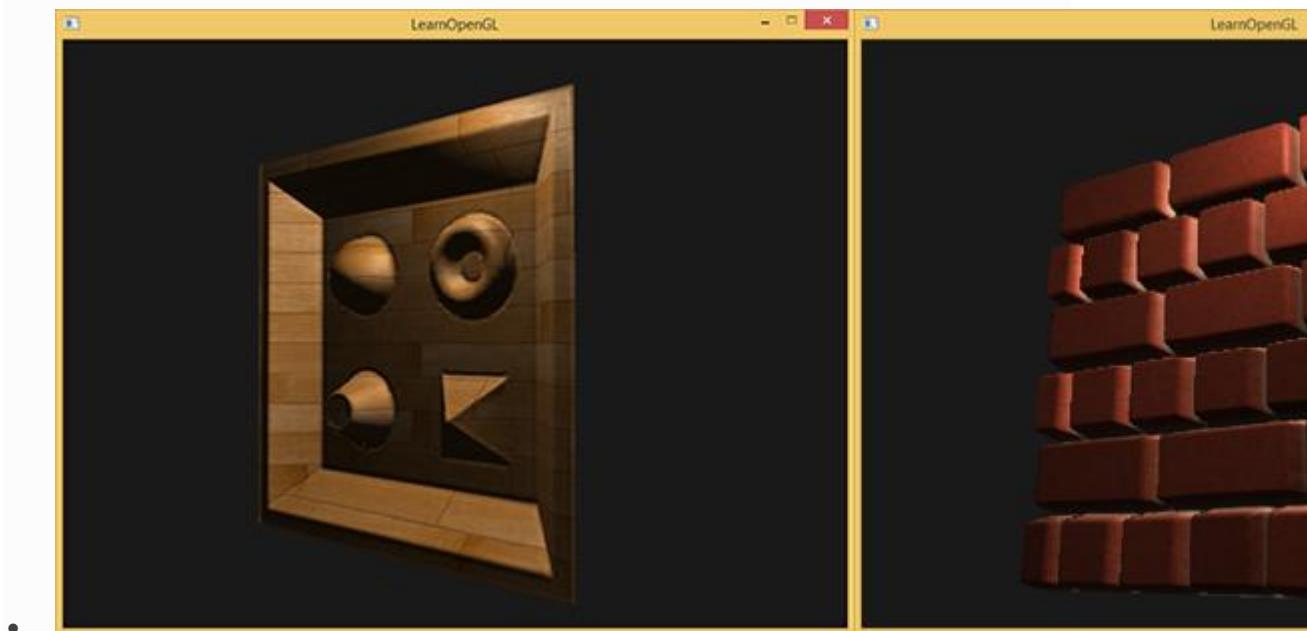
- Parallax Occlusion Mapping 和陡峭视差映射的原则相同，但不是用触碰的第一个深度层的纹理坐标，而是在触碰之前和之后，在深度层之间进行线性插值。我们根据表面的高度距离哪个深度层的深度层值的距离来确定线性插值的大小。看看下面的图 pain 就能了解它是如何工作的：
- 你可以看到大部分和陡峭视差映射一样，不一样的地方是有个额外的步骤，两个深度层的纹理坐标围绕着交叉点的线性插值。这也是近似的，但是比陡峭视差映射更精确。
- Parallax Occlusion Mapping 的代码基于陡峭视差映射，所以并不难：

```
• [...] // 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。
- Parallax Occlusion Mapping 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能被注意到了，对于提升物体的细节可以祈祷难以置信的效果。

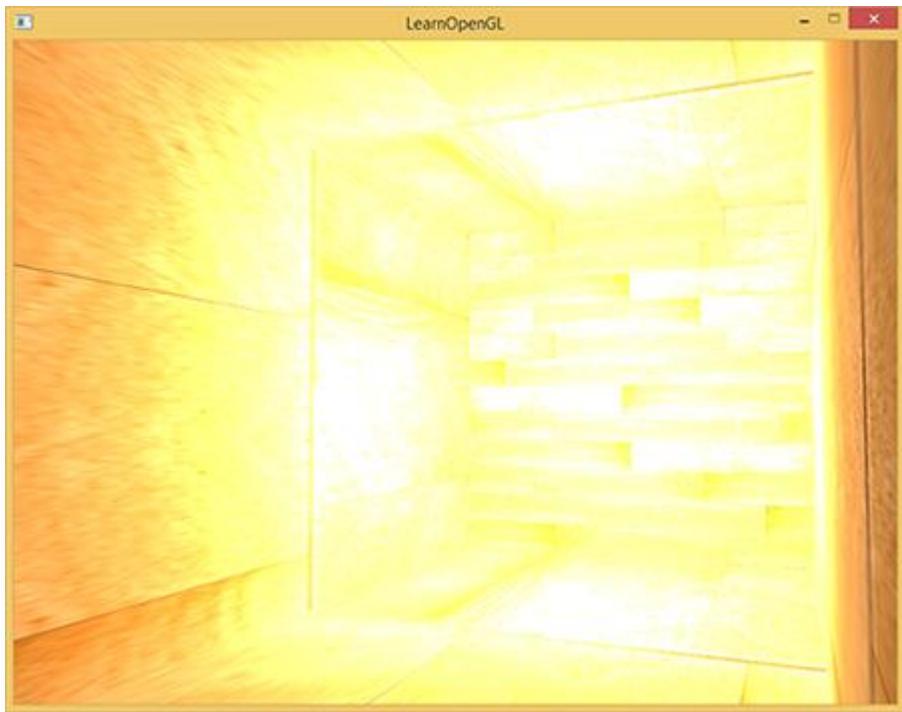
- 附加资源
- [Parallax Occlusion Mapping in GLSL](#): sunandblackcat.com 上的视差贴图教程。
- [How Parallax Displacement Mapping Works](#): TheBennyBox 的关于视差贴图原理的视频教程。

本文作者 JoeyDeVries, 由 Meow J 翻译自 <http://learnopengl.com>

- 
- **HDR**

## HDR

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，其中有很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** 值转换成在[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(Floating Point Framebuffers)

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 `GLRGB16F` 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, hdrFBO);

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 范围中. 我们需要应用到色调映射.

## 色调映射(Tone Mapping)

色调映射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** 色调映射算法平均得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，给我们对于无损场景光照精确的控制.

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于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, 因为它们可以由色调映射重新获取; 多个光照的叠加不会导致亮度被约束的区域; 光照可以被设定为他们原来的亮度而不是被 LDR 值限定. 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他.

## 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.

本文作者 JoeyDeVries, 由 Django 翻译自 <http://learnopengl.com>

•

- **Bloom 泛光**

## 泛光(Bloom)

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 **bloom** 来实现。**bloom** 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）：

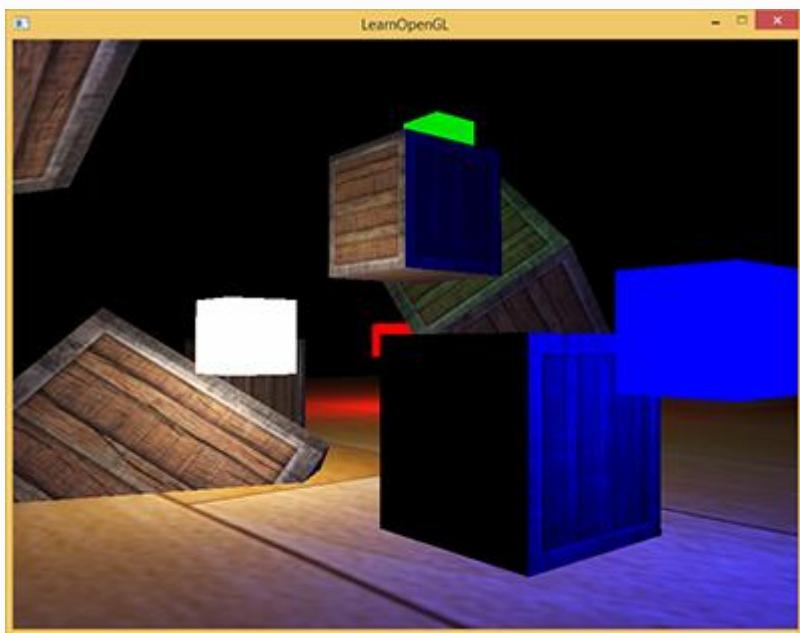


**Bloom** 是我们能够注意到一个明亮的物体真的有种明亮的感觉。**bloom** 可以极大提升场景中的光照效果，并提供了极大的效果提升，尽管做到这一切只需一点改变。

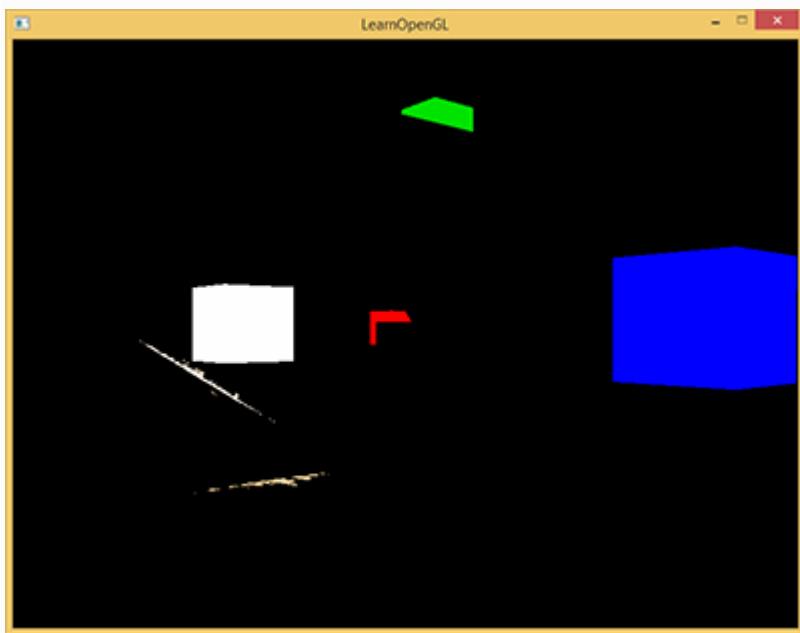
**Bloom** 和 **HDR** 结合使用效果很好。常见的一个误解是 **HDR** 和 **bloom** 是一样的，很多人认为两种技术是可以互换的。但是它们是两种不同的技术，用于各自不同的目的上。可以使用默认的 8 位精确度的帧缓冲，也可以在不使用 **bloom** 效果的时候，使用 **HDR**。只不过在有了 **HDR** 之后再实现 **bloom** 就更简单了。

为实现 **bloom**，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。

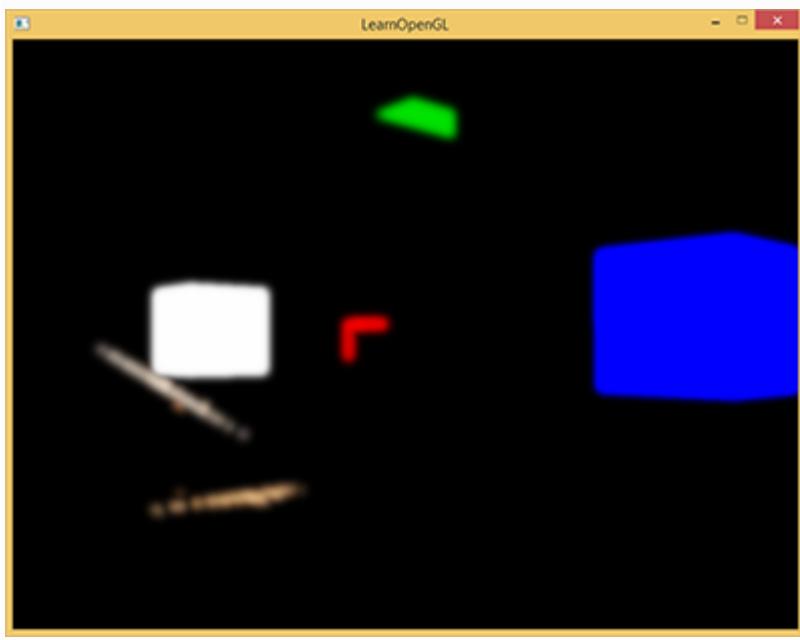
我们来一步一步解释这个处理过程。我们在场景中渲染一个带有 4 个立方体形式不同颜色的明亮的光源。带有颜色的发光立方体的亮度在 1.5 到 15.0 之间。如果我们将其渲染至 **HDR** 颜色缓冲，场景看起来会是这样的：



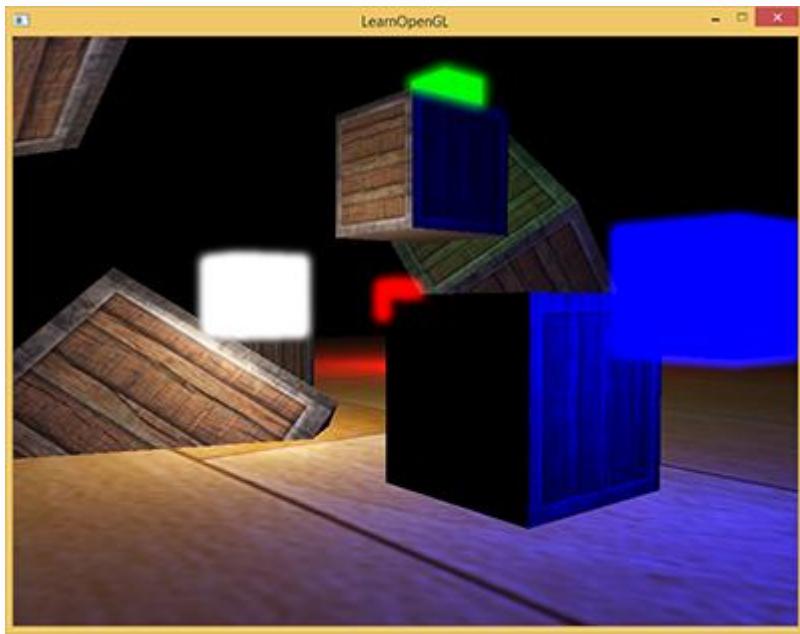
我们得到这个 **HDR** 颜色缓冲纹理，提取所有超出一定亮度的 **fragment**。这样我们就会获得一个只有 **fragment** 超过了一定阈限的颜色区域：



我们将这个超过一定亮度阈限的纹理进行模糊。**bloom** 效果的强度很大程度上被模糊过滤器的范围和强度所决定。

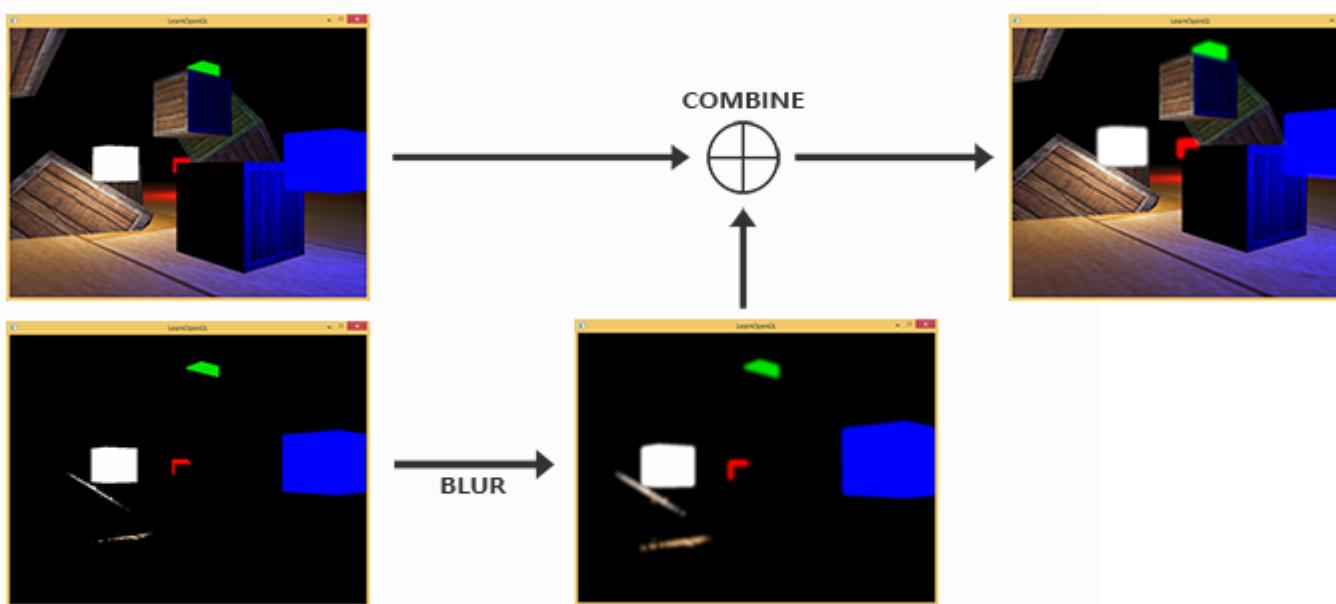


最终的被模糊化的纹理就是我们用来获得发出光晕效果的东西。这个已模糊的纹理要添加到原来的 **HDR** 场景纹理的上部。因为模糊过滤器的应用明亮区域发出光晕，所以明亮区域在长和宽上都有所扩展。



**bloom** 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。简单的改改模糊过滤器就会极大的改变 **bloom** 效果的品质。

下面这几步就是 **bloom** 后处理特效的过程，它总结了实现 **bloom** 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，这样我们就能定义多个像素着色器了；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中的时候，一个着色器使用一个布局 `location` 修饰符，那么 `fragment` 就会用相应的颜色缓冲就会被用来渲染。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

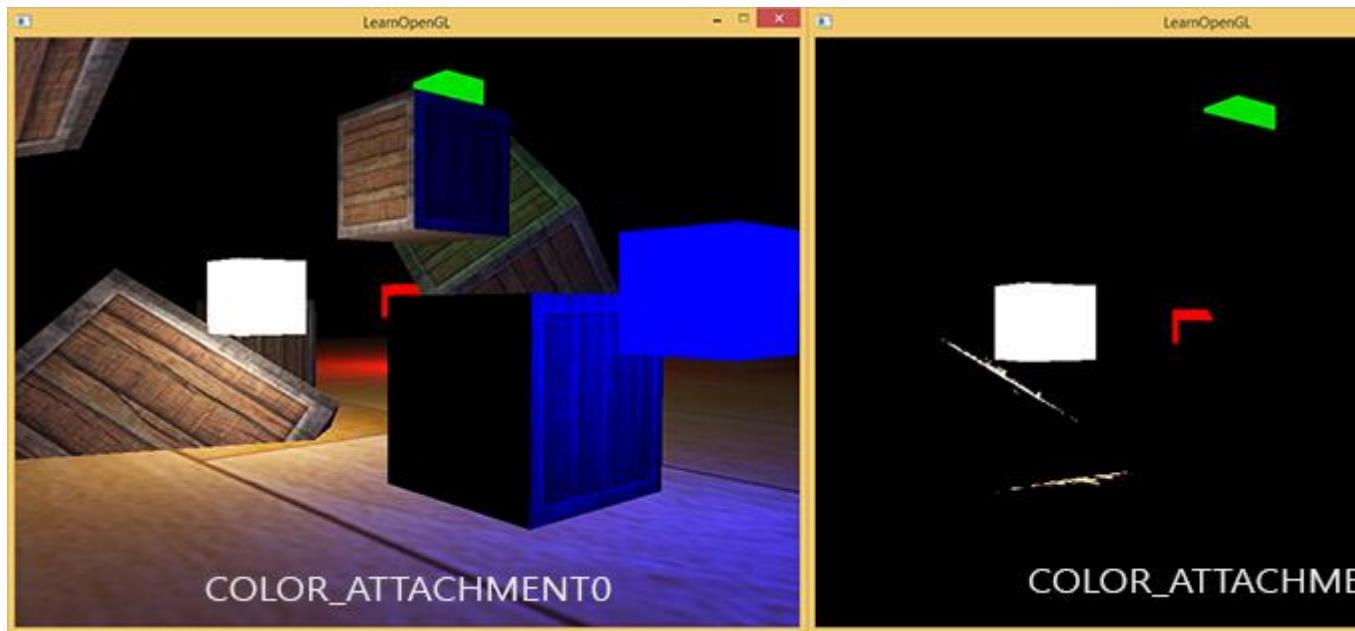
float brightness = dot(FragColor.rgb, vec3(0.2126, 0.7152,
0.0722));

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 `bloom` 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。

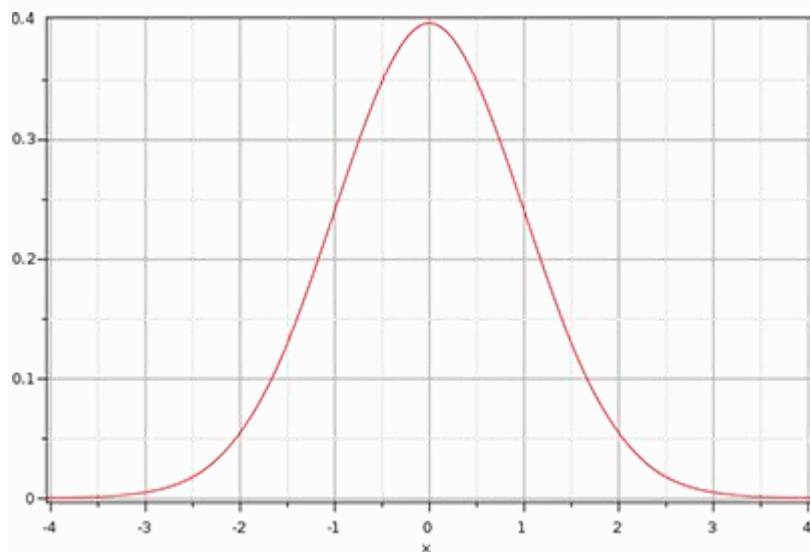
有了两个颜色缓冲，我们就有了一个正常场景的图像和一个提取出的亮区的图像；这些都在一个渲染步骤中完成。



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。

## 高斯模糊

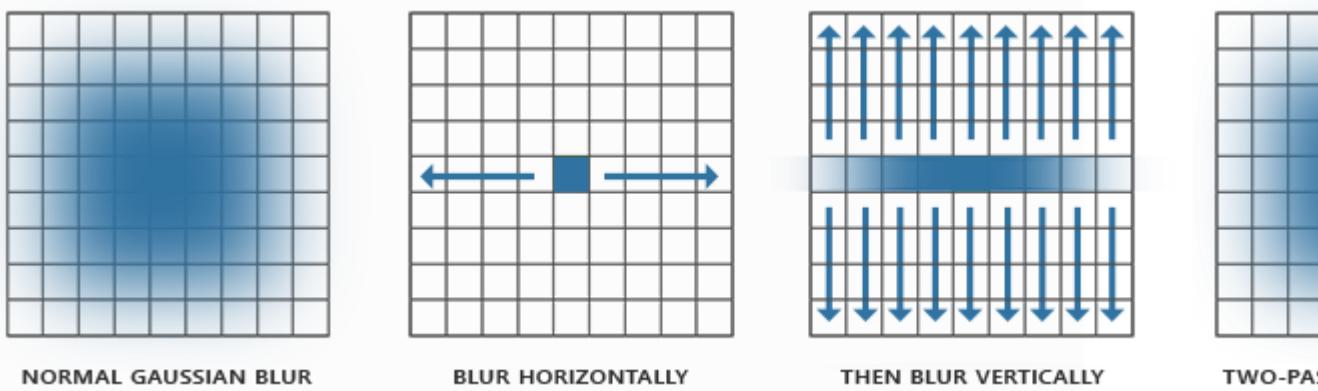
在后处理教程那里，我们采用的模糊是一个图像中所有周围像素的均值，它的确为我们提供了一个简易实现的模糊，但是效果并不好。高斯模糊基于高斯曲线，高斯曲线通常被描述为一个钟形曲线，中间的值达到最大化，随着距离的增加，两边的值不断减少。高斯曲线在数学上有不同的形式，但是通常是这样的形状：



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  $32 \times 32$  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  $32 \times 32$  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 pingpongFBO[2];

GLuint pingpongBuffer[2];

glGenFramebuffers(2, pingpongFBO);

```

```
glGenTextures(2, pingpongColorbuffers);

for (GLuint i = 0; i < 2; i++)

{
 glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[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，然后对其进行模糊处理 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, pingpongFBO[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;

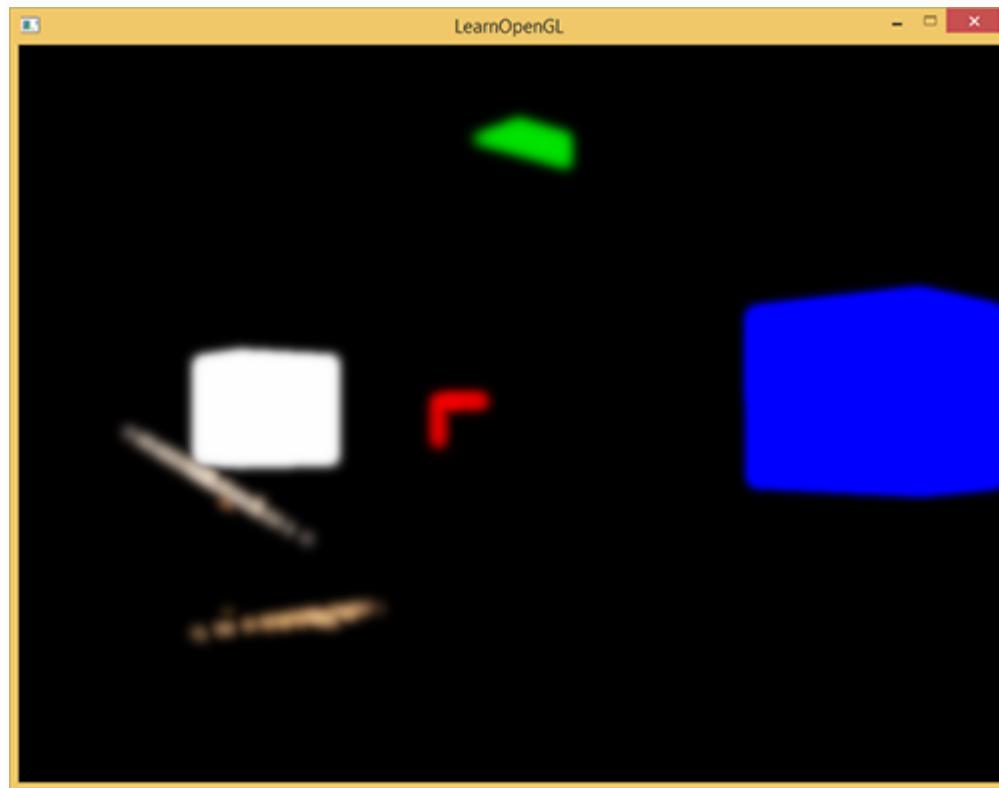
}

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。



bloom 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 bloom 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);

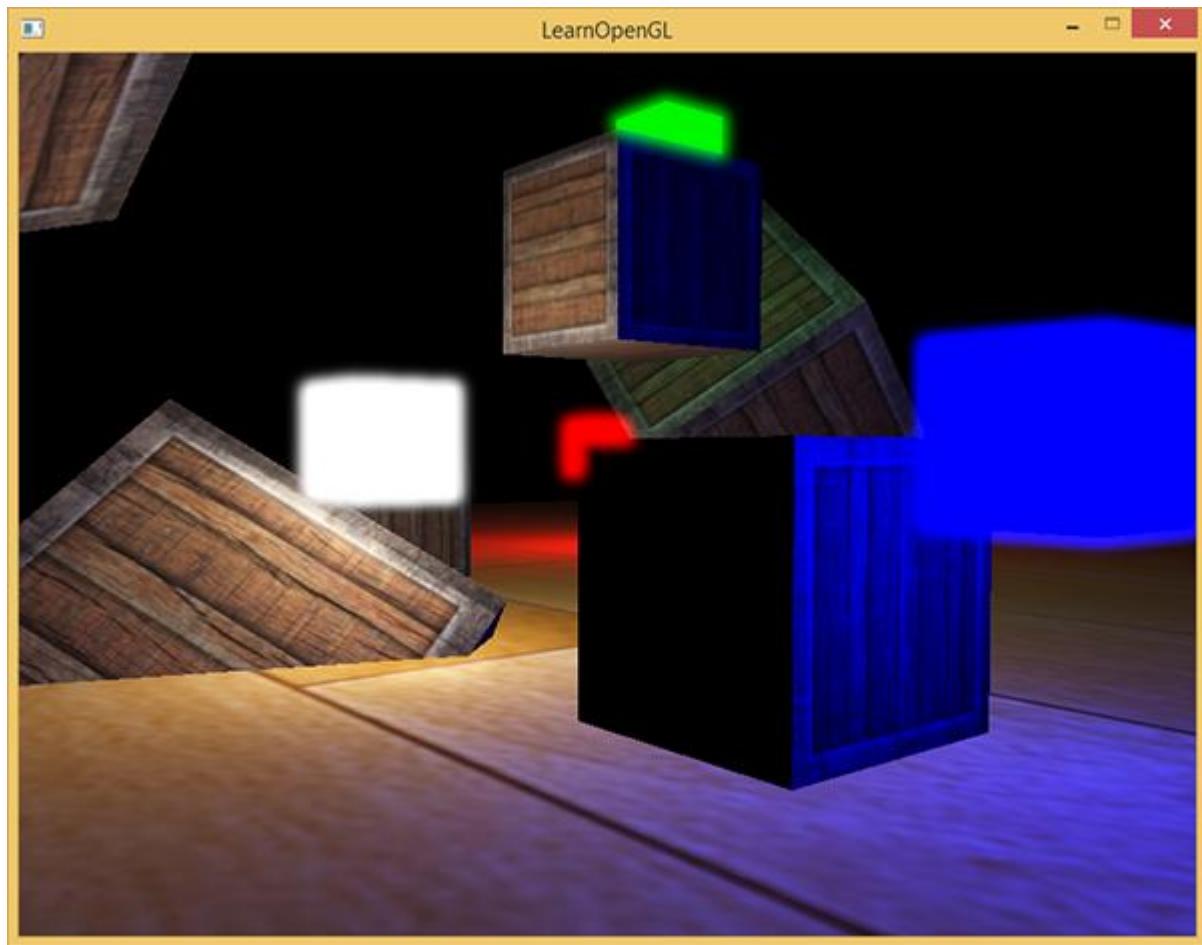
 // also gamma correct while we're at it
 result = pow(result, vec3(1.0 / gamma));

 FragColor = vec4(result, 1.0f);
}

```

要注意的是我们要在应用色调映射之前添加 **bloom** 效果。这样添加的亮区的 **bloom**，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 **bloom** 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 **bloom** 效果的质量正相关，提升模糊效果就能够提升 **bloom** 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 **bloom** 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。

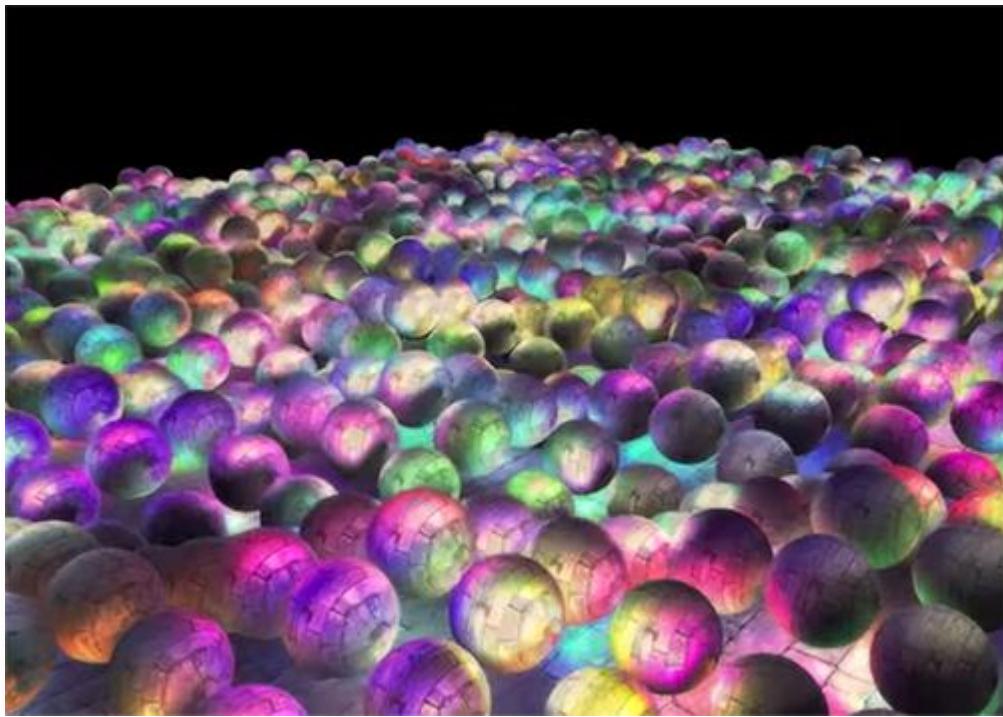
- [Bloom Post Process Effect](#): 来自 Epic Games 关于通过对权重的多个高斯曲线结合来提升 bloom 效果的文章。
- [How to do good bloom for HDR rendering](#): Kalogirou 的文章描述了如何使用更好的高斯模糊算法来提升 bloom 效果。

## 延迟着色法(Deferred Shading)

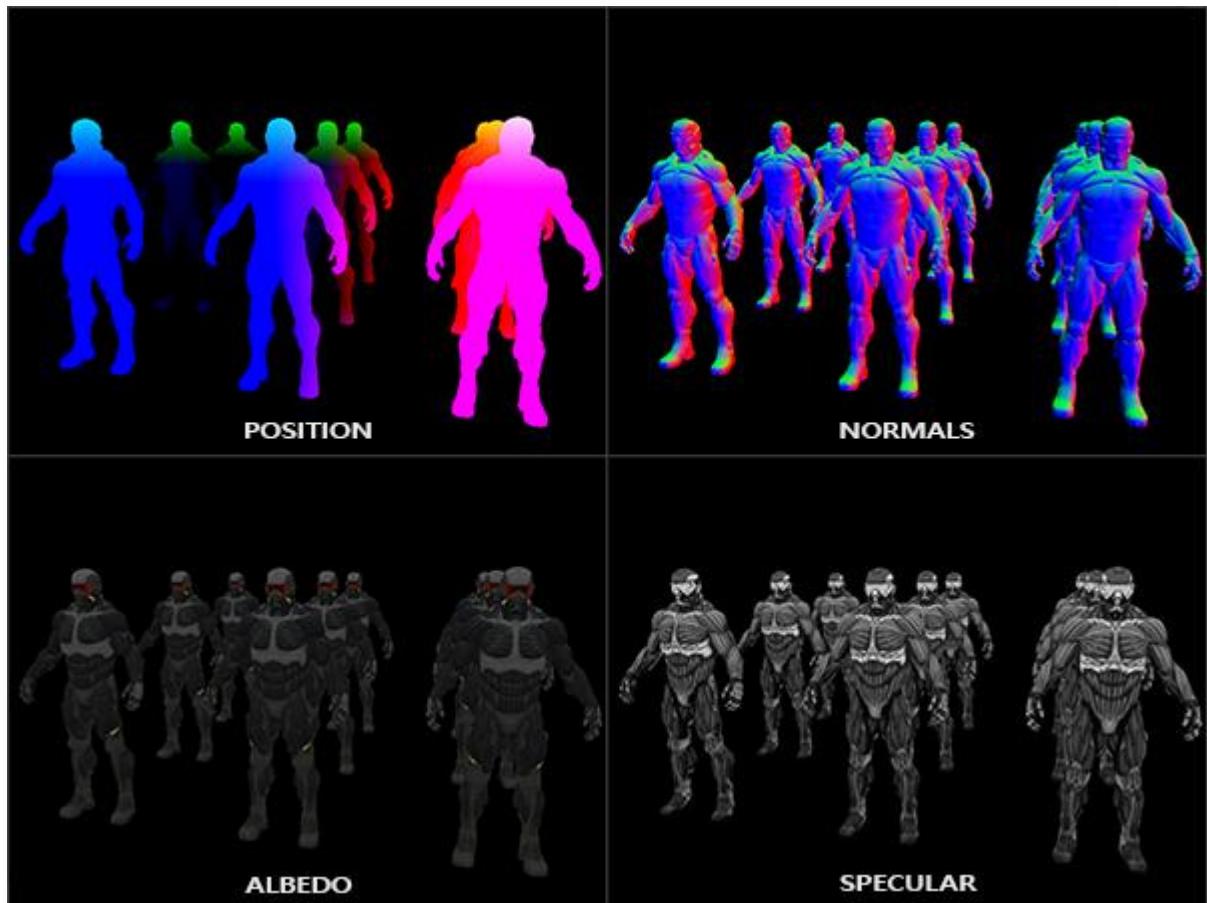
| 原文 | <a href="#"><u>Deferred Shading</u></a> |
|----|-----------------------------------------|
| 作者 | JoeyDeVries                             |
| 翻译 | Meow J                                  |
| 校对 | 未校对                                     |

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](#))。

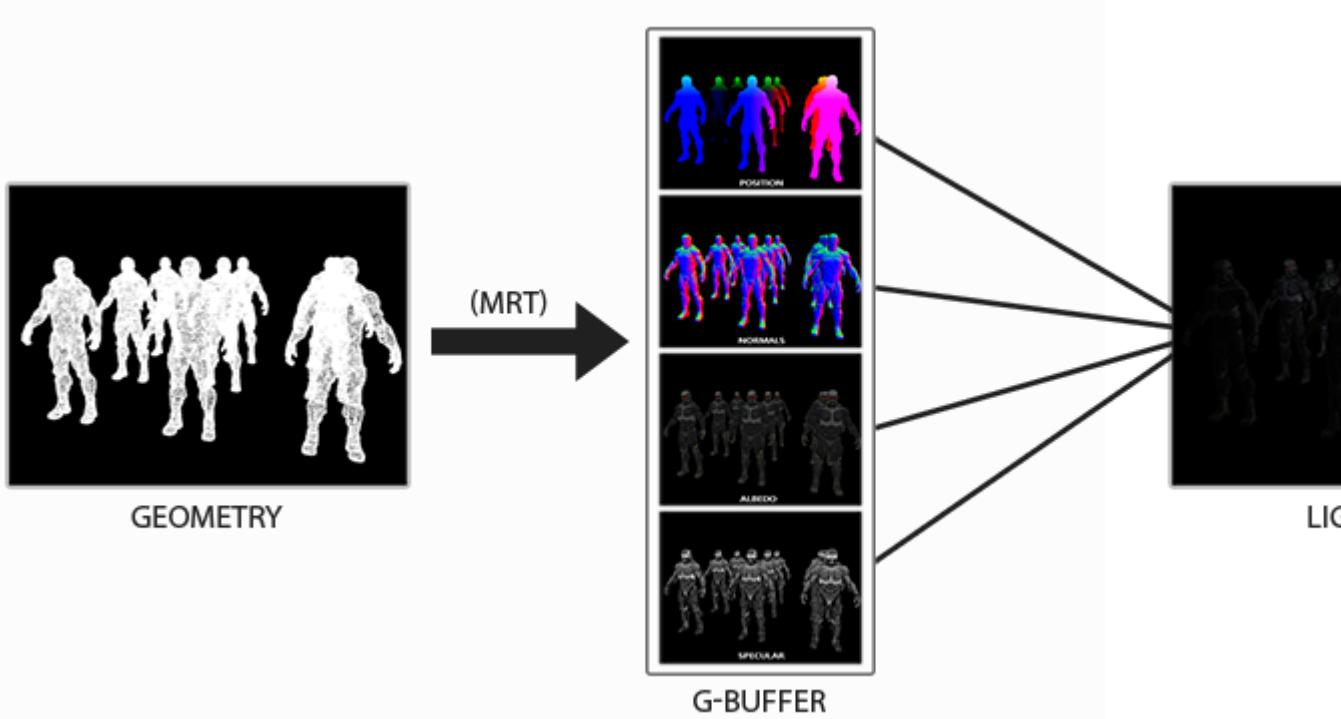


延迟着色法基于我们**延迟(Defer)**或**推迟(Postpone)**大部分计算量非常大的渲染(像是光照)到后期进行处理的想法。它包含两个处理阶段(**Pass**)：在第一个几何处理阶段(**Geometry Pass**)中，我们先渲染场景一次，之后获取对象的各种几何信息，并储存在一系列叫做**G 缓冲(G-buffer)**的纹理中；想想位置向量(**Position Vector**)、颜色向量(**Color Vector**)、法向量(**Normal Vector**)和/或镜面值(**Specular Value**)。场景中这些储存在**G 缓冲**中的几何信息将会在之后用来做(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。

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是位置，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` 关联的哪个颜色缓冲。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;
}

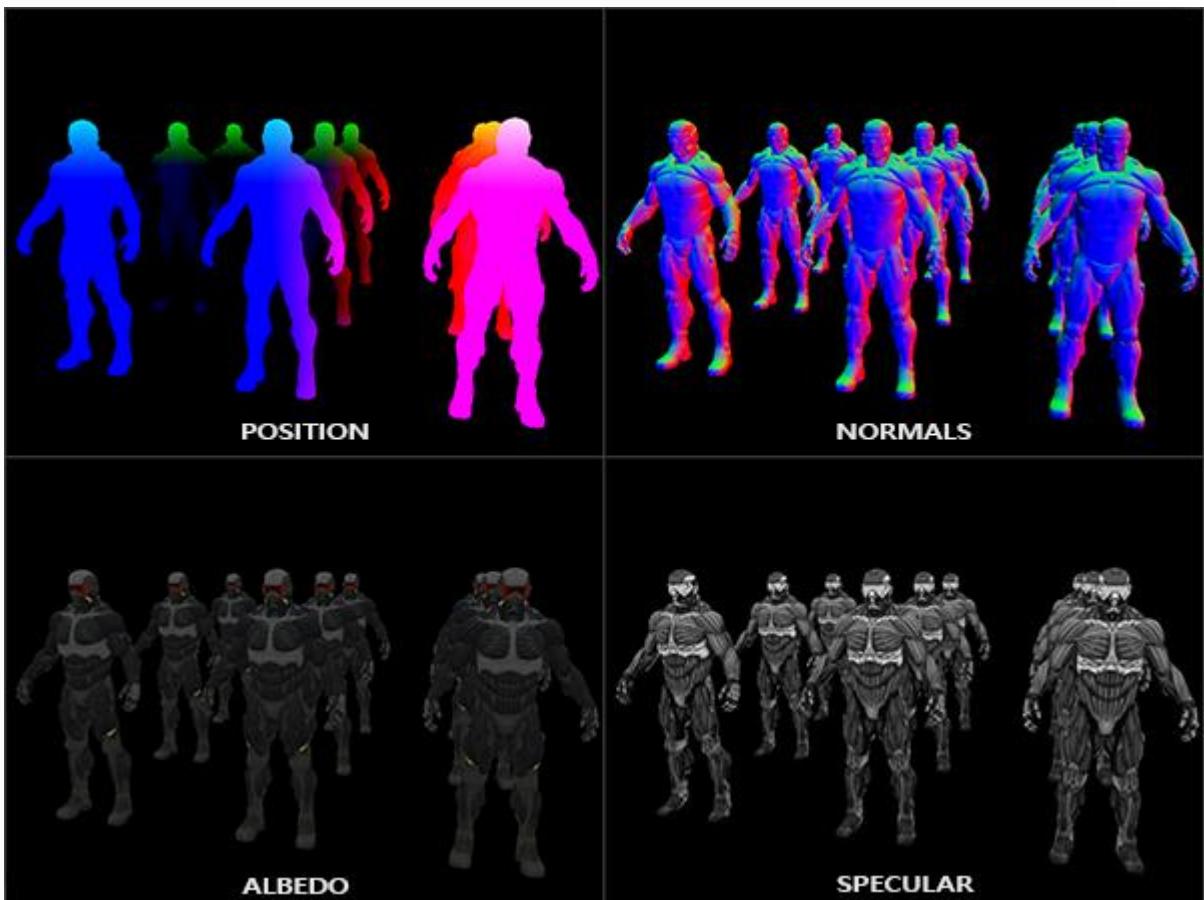
```

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。

### Attention

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他们显示出来，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照片段着色器。

```
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;

}

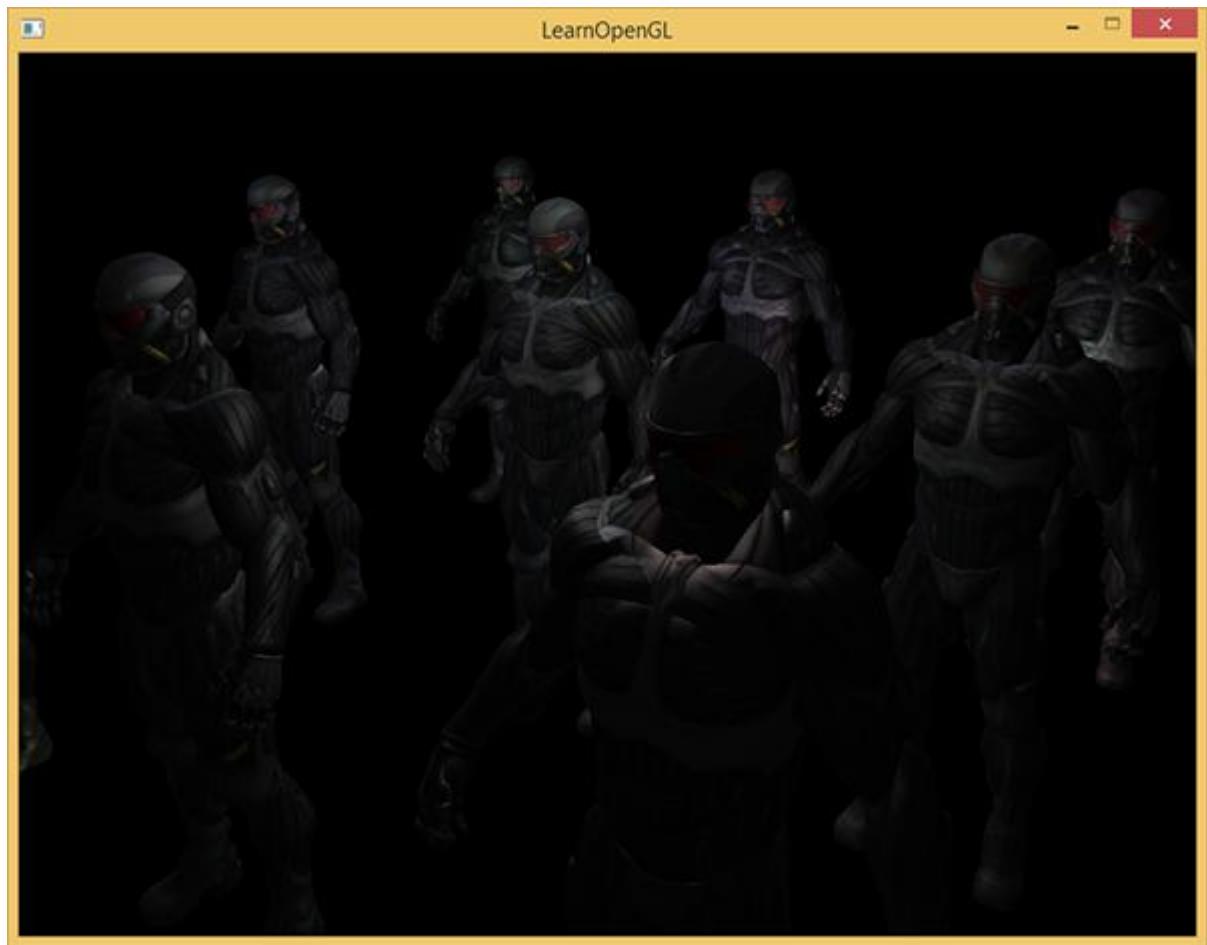
FragColor = vec4(lighting, 1.0);
}

```

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，延迟渲染它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。很大一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：

$$F_{light} = \frac{I}{K_c + K_l * d + K_q * d^2}$$

我们现在想要在  $F_{light}$  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  $5/256$  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)。

### Important

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  $5/256$  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：

$$\frac{5}{256} = \frac{I_{max}}{Attenuation}$$

在这里,  $I_{max}$ 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:

$$\frac{5}{256} * Attenuation = I_{max}$$

$$5 * Attenuation = I_{max} * 256$$

$$Attenuation = I_{max} * \frac{256}{5}$$

$$K_c + K_l * d + K_q * d^2 = I_{max} * \frac{256}{5}$$

$$K_q * d^2 + K_l * d + K_c - I_{max} * \frac{256}{5} = 0$$

最后的方程形成了 $ax^2 + bx + c = 0$ 的形式, 我们可以用求根公式来解这个二次方程:

$$x = \frac{-K_l + \sqrt{K_l^2 - 4 * K_q * (K_c - I_{max} * \frac{256}{5})}}{2 * K_q}$$

它给我们了一个通用公式从而允许我们计算  $x$  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

 [...]

 }
}

```

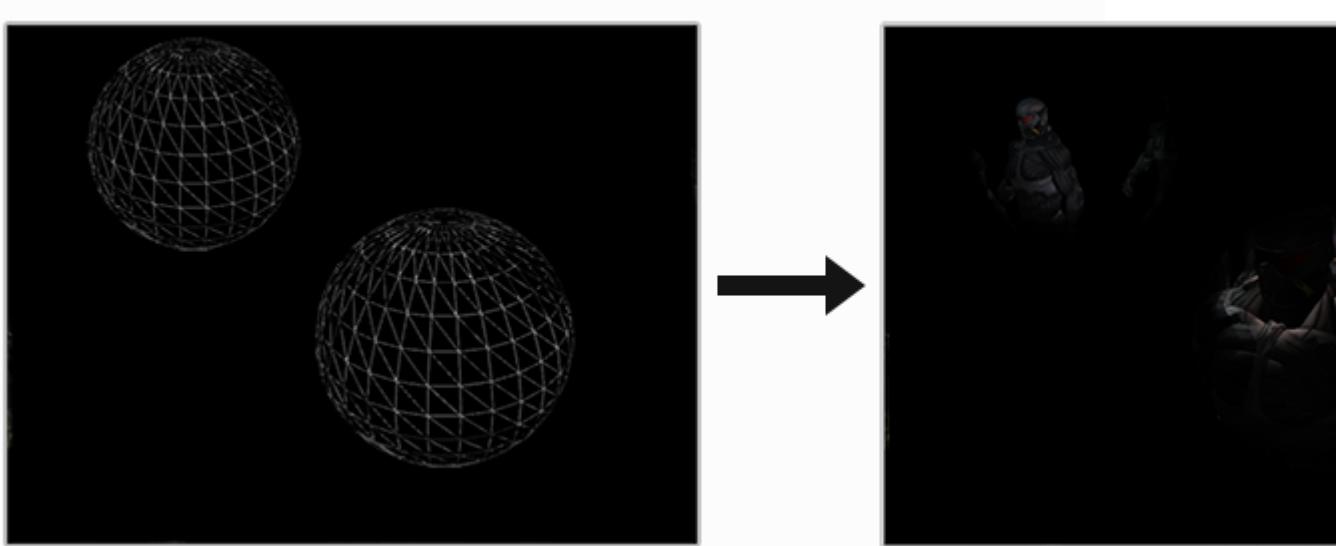
这次的结果和之前一模一样，但是这次物体只对所在光体积的光源计算光照。

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 **切片式延迟着色法(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

## SSAO

| 原文 | <a href="#">SSAO</a> |
|----|----------------------|
| 作者 | JoeyDeVries          |
| 翻译 | Meow J               |
| 校对 | 未校对                  |

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？

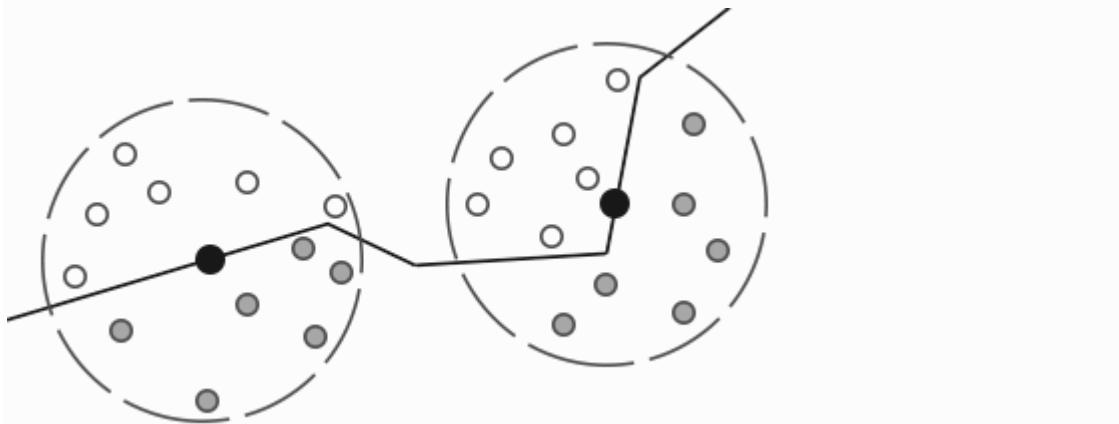
下面这幅图展示了在使用和不使用 SSAO 时场景的不同。特别注意对比褶皱部分，你会发现(环境)光被遮蔽了许多：



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。

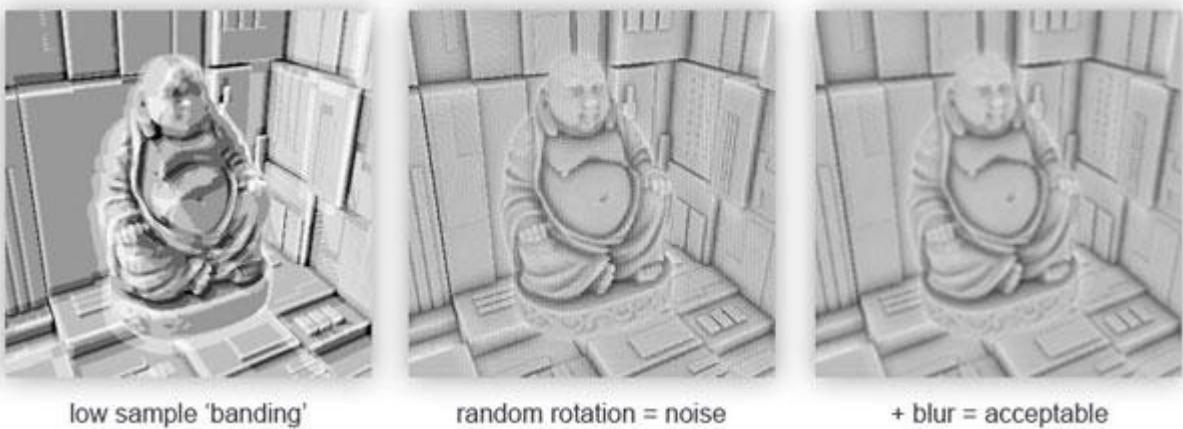
环境光遮蔽这一技术会带来很大的性能开销，因为它还需要考虑周围的几何体。我们可以对空间中每一点发射大量光线来确定其遮蔽量，但是这在实时运算中会很快变成大问题。在 2007 年，Crytek 公司发布了一款叫做**屏幕空间环境光遮蔽 (Screen-Space Ambient Occlusion, SSAO)**的技术，并用在了他们的看家作孤岛危机上。这一技术使用了屏幕空间场景的深度而不是真实的几何体数据来确定遮蔽量。这一做法相对于真正的环境光遮蔽不但速度快，而且还能获得很好的效果，使得它成为近似实时环境光遮蔽的标准。

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。

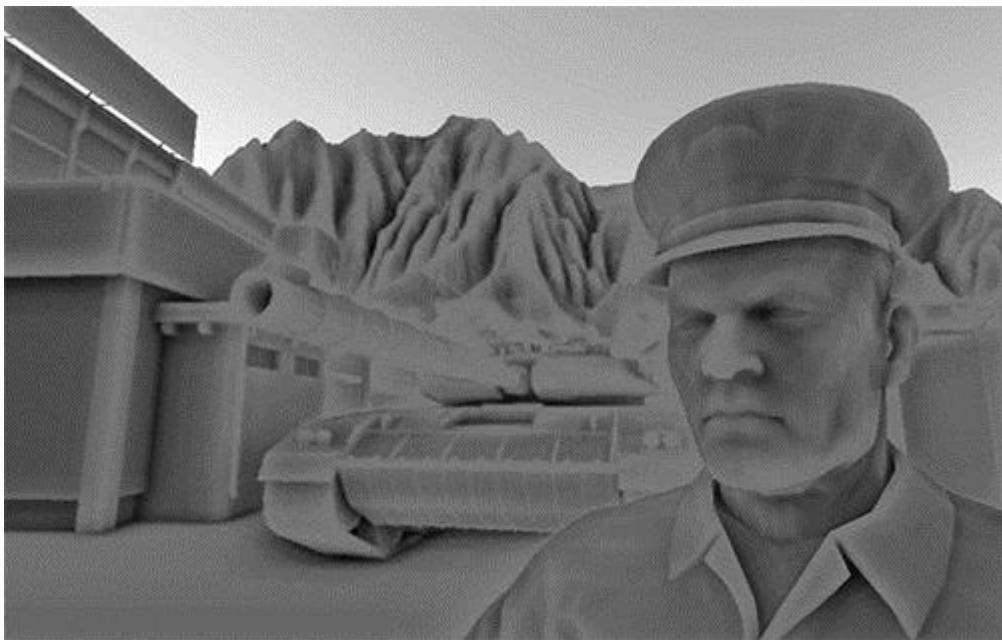
很明显，渲染效果的质量和精度与我们采样的样本数量有直接关系。如果样本数量太低，渲染的精度会急剧减少，我们会得到一种叫做波纹(**Banding**)的效果；如果它太高了，反而会影响性能。我们可以通过引入随机性到采样核心(**Sample Kernel**)的采样中从而减少样本的数目。通过随机旋转采样核心，我们能在有限样本数量中得到高质量的结果。然而这仍然会有一定的麻烦，因为随机性引入了一个很明显的噪声图案，我们将需要通过模糊结果来修复这一问题。下面这幅图片([John Chapman](#) 的佛像)展示了波纹效果还有随机性造成的效果：



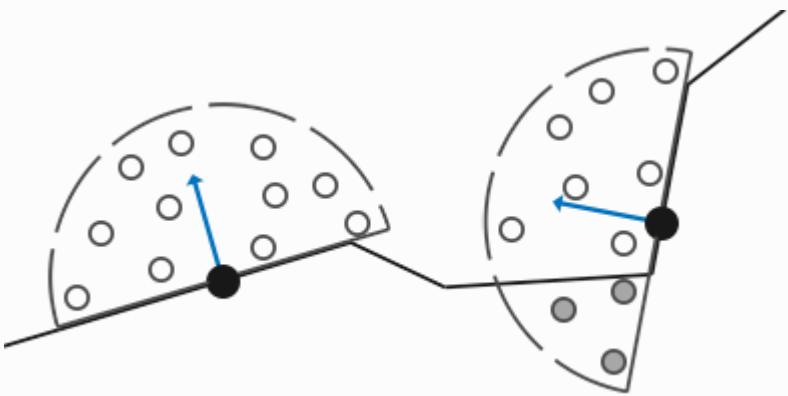
你可以看到，尽管我们在低样本数的情况下得到了很明显的波纹效果，引入随机性之后这些波纹效果就完全消失了。

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。

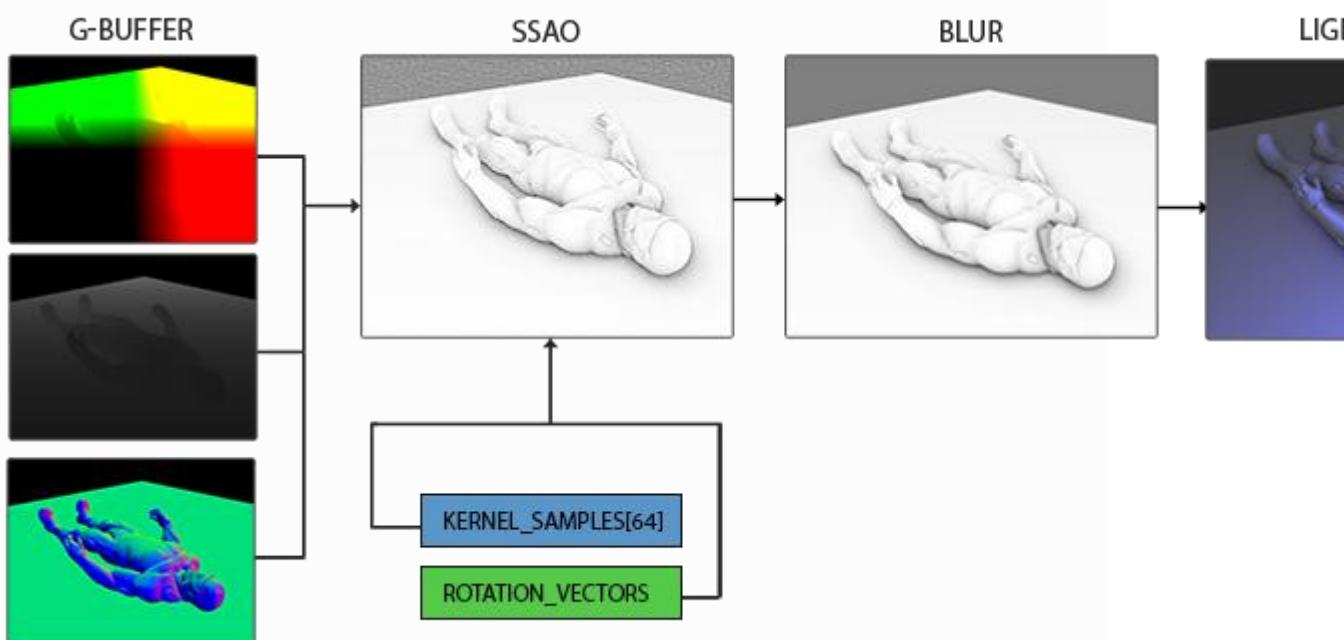


通过在法向半球体(**Normal-oriented Hemisphere**)周围采样，我们将不会考虑到片段底部的几何体。它消除了环境光遮蔽灰蒙蒙的感觉，从而产生更真实的结果。这个 SSAO 教程将会基于法向半球法和 John Chapman 出色的 [SSAO 教程](#)。

## 样本缓冲(Sample buffers)

**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 - 线性深度纹理 - 采样核心 - 用来旋转采样核心的逐片段随机旋转矢量

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。

### Important

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)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。

## Important

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);

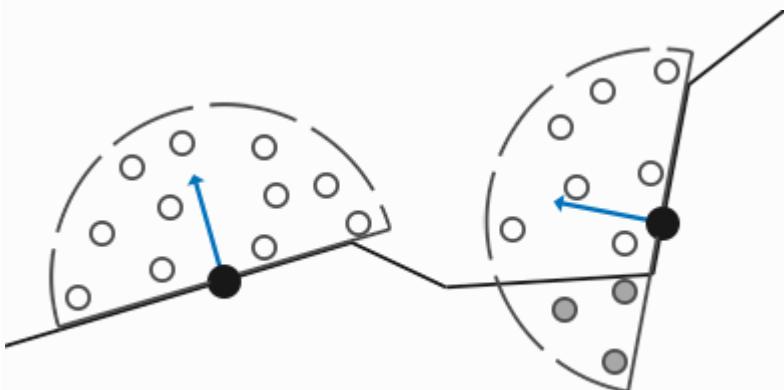
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
 GL_CLAMP_TO_EDGE);
```

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(Normal-oriented Hemisphere)

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)
```

```
{\n glm::vec3 sample(\n randomFloats(generator) * 2.0 - 1.0,\n randomFloats(generator) * 2.0 - 1.0,\n randomFloats(generator)\n);\n\n sample = glm::normalize(sample);\n\n sample *= randomFloats(generator);\n\n GLfloat scale = GLfloat(i) / 64.0;\n\n ssaoKernel.push_back(sample);\n}\n
```

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]\n\n scale = lerp(0.1f, 1.0f, scale * scale);\n\n sample *= scale;\n\n ssaoKernel.push_back(sample);\n}\n
```

lerp 被定义为：

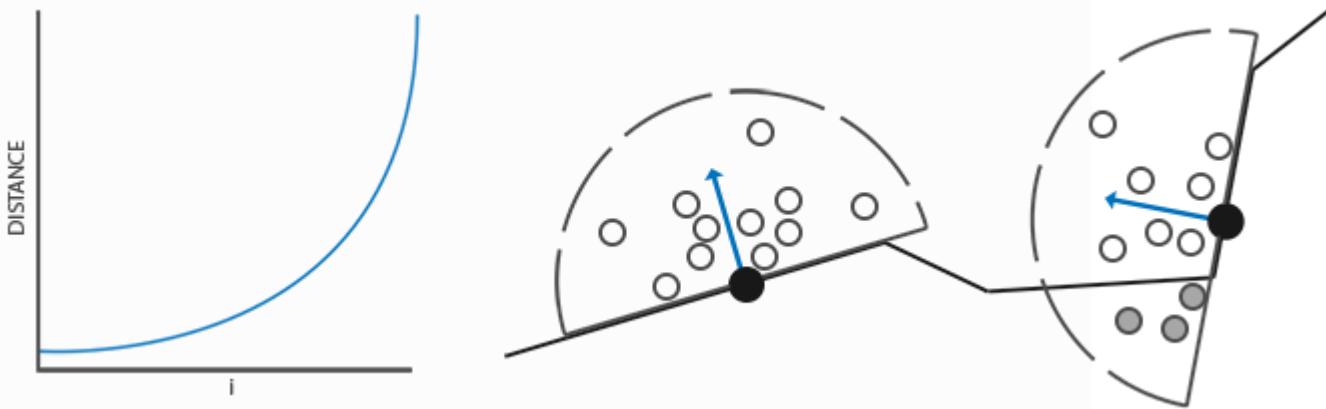
```

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  $4 \times 4$  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);
}
```

由于采样核心实验者正 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它们的默认值为 `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)：



without range check



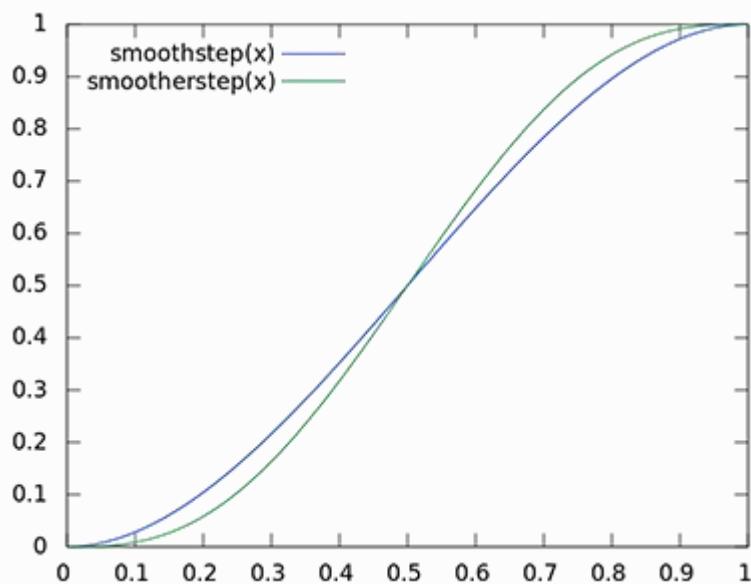
with range check

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。

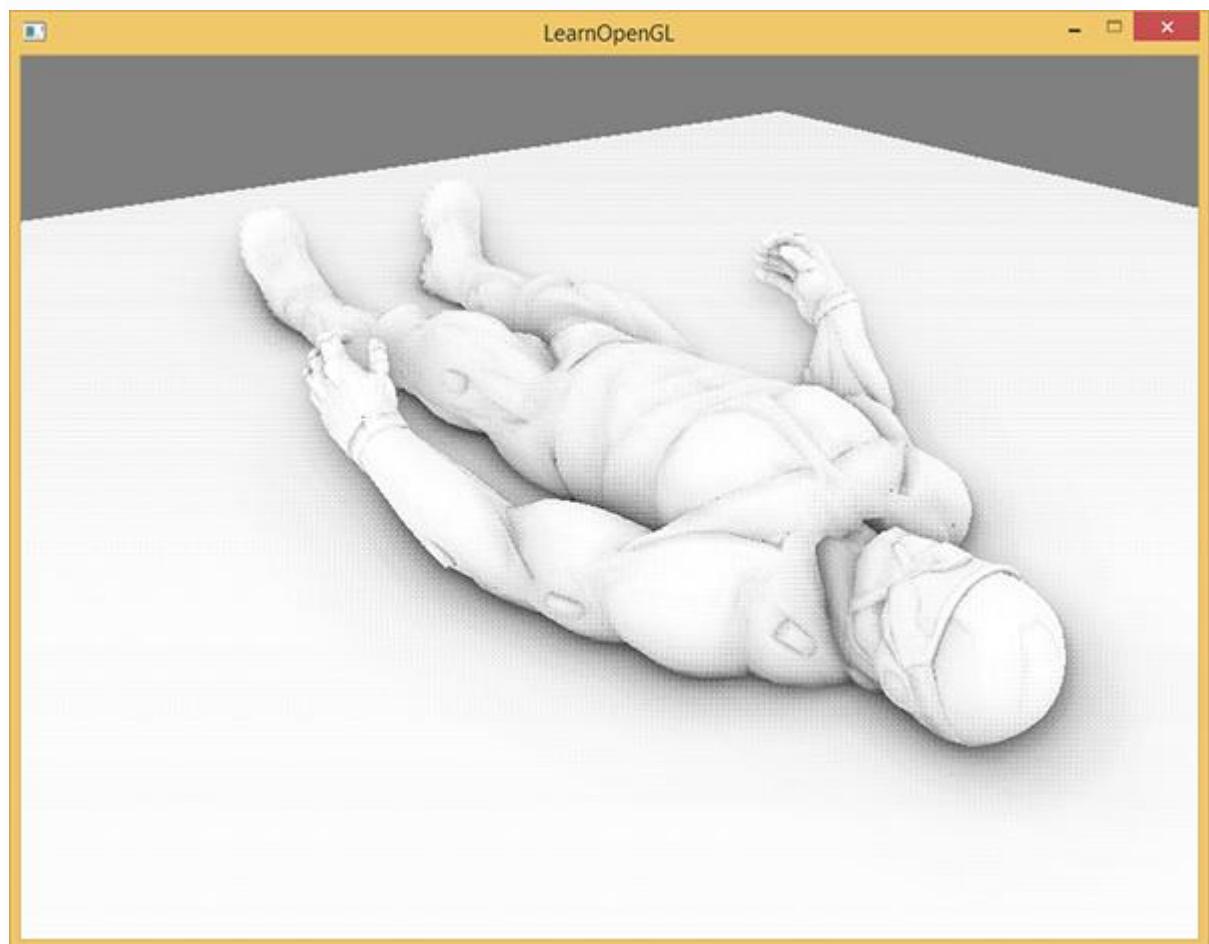
最后一步，我们需要将遮蔽贡献根据核心的大小标准化，并输出结果。注意我们用 1.0 减去了遮蔽因子，以便直接使用遮蔽因子去缩放环境光照分量。

```
}
```

```
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 ssaoBlurFBO, ssaoColorBufferBlur;

glGenFramebuffers(1, &ssaoBlurFBO);

glBindFramebuffer(GL_FRAMEBUFFER, ssaoBlurFBO);

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;

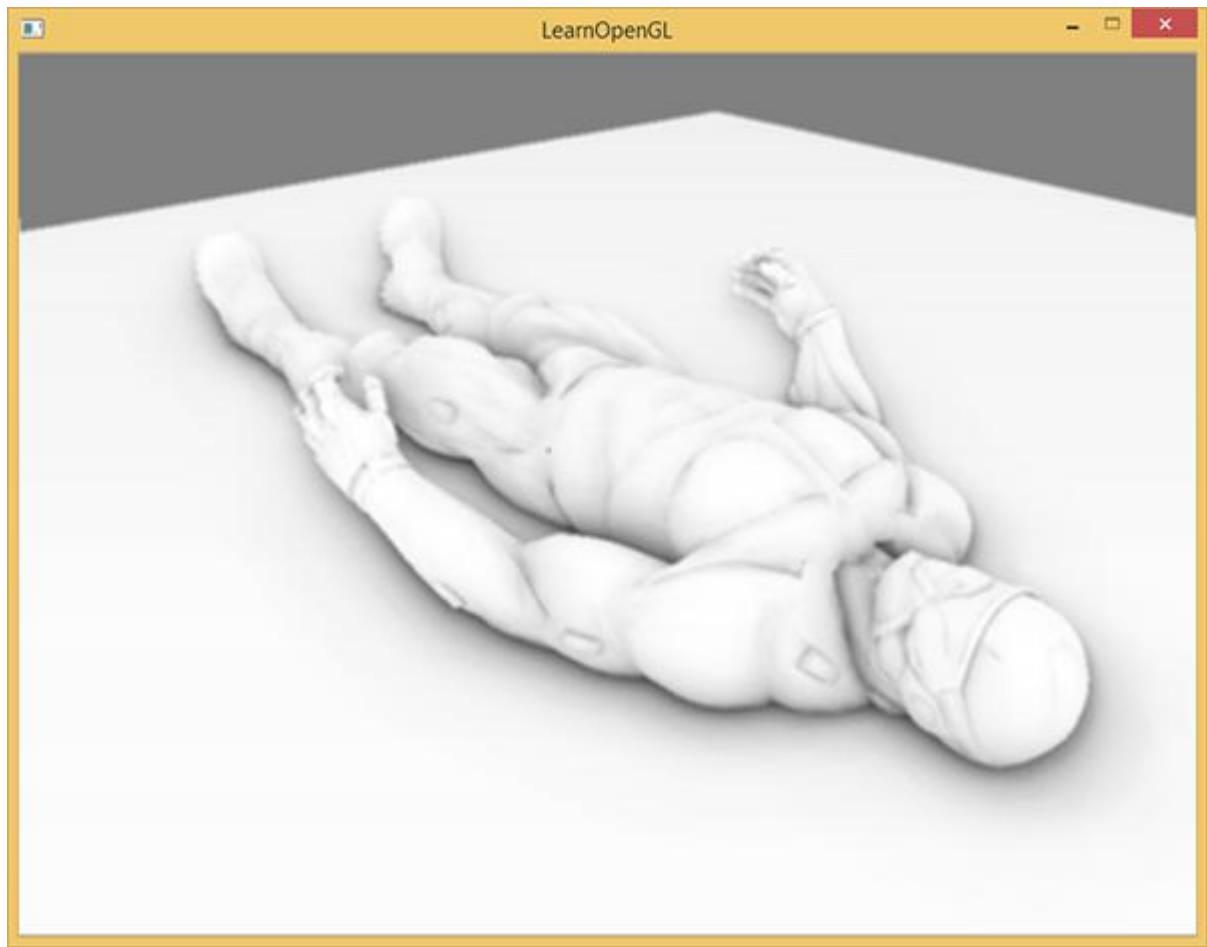
 }

 }

 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;
```

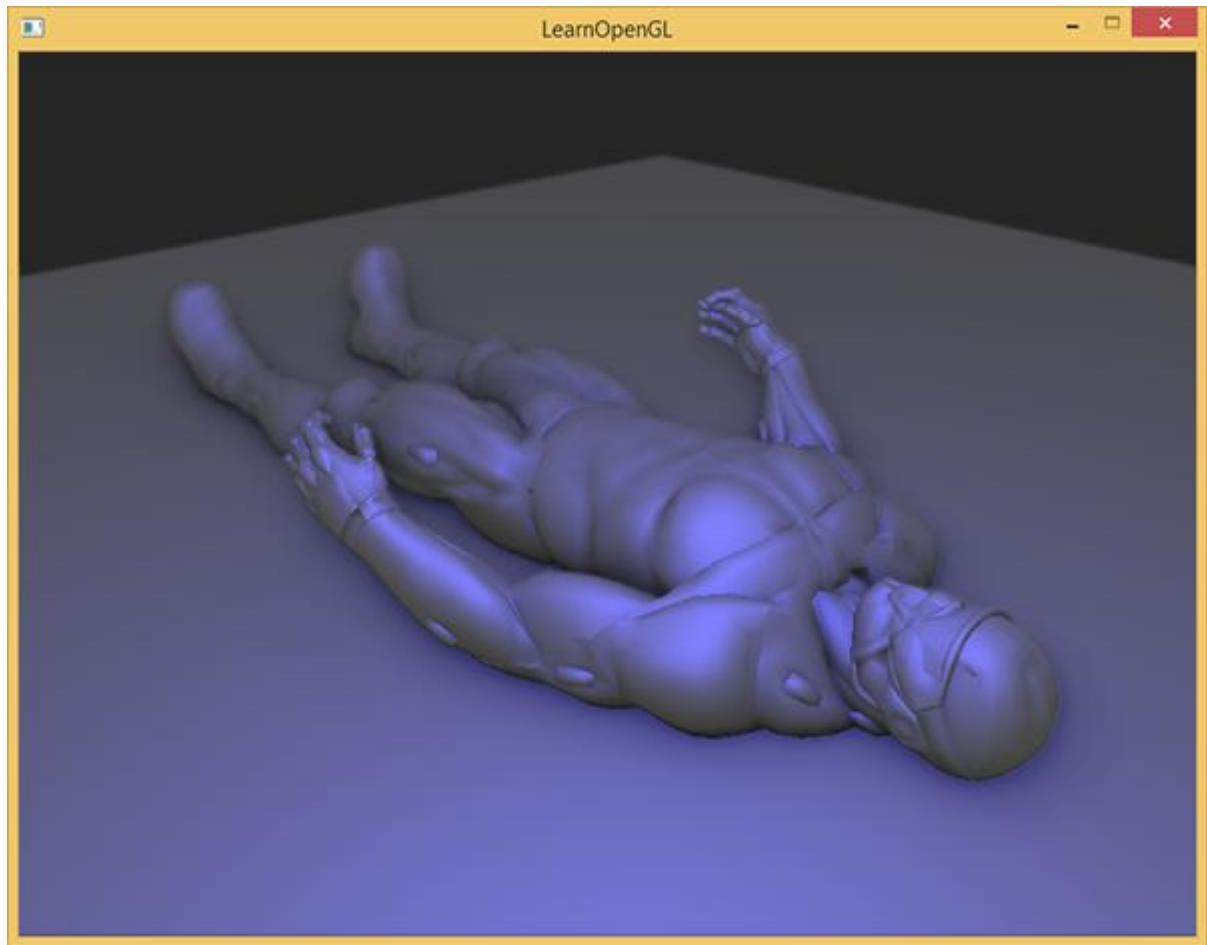
```
specular *= attenuation;
```

```
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

## 文字渲染

| 原文 | <a href="#">Text Rendering</a> |
|----|--------------------------------|
| 作者 | JoeyDeVries                    |
| 翻译 | <a href="#">Geequlim</a>       |
| 校对 | gjy_1992                       |

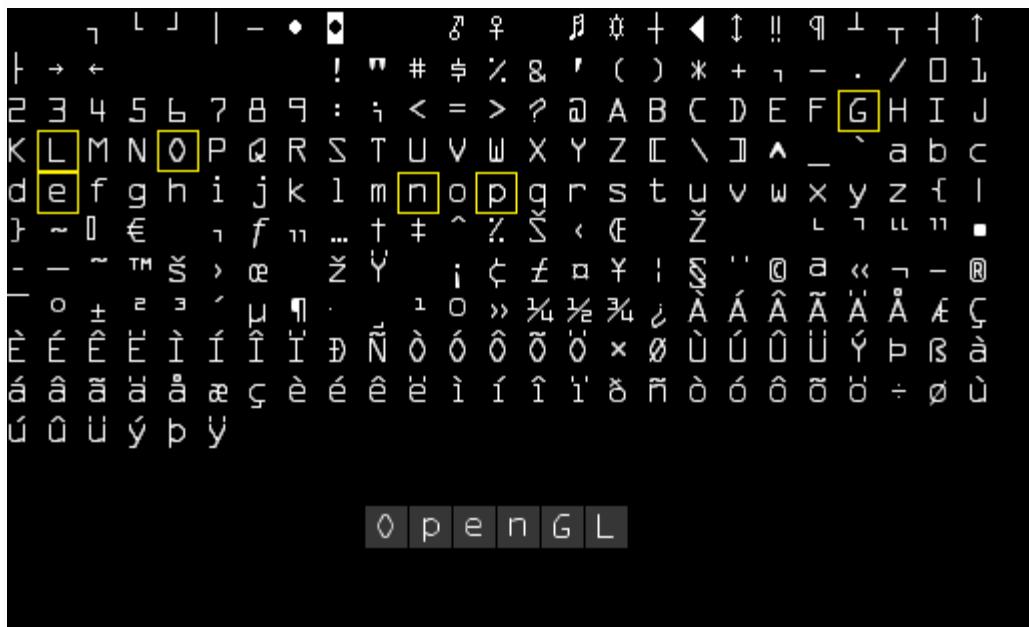
当你在图形计算领域冒险到了一定阶段以后你可能会想使用 OpenGL 来绘制文字。然而，可能与你想象的并不一样，使用像 OpenGL 这样的底层库来把文字渲染到屏幕上并不是一件简单的事情。如果你只需要绘制 128 种字符，那么事情可能会简单一些。但是当我们要绘制的字符有着不同的宽、高和边距；如果你使用的语言中不止包含 128 个字符；当你要绘制音乐符、数学符号；以及考虑把如何处理文本自动转行等等情况考虑进来的时候...事情马上就会变得复杂得多，你甚至觉得这些工作并不属于像 OpenGL 这样的底层图形库该讨论的范畴。

由于 OpenGL 本身并没有定义如何渲染文字到屏幕，也没有用于表示文字的基本图形，我们必须自己定义一套全新的方式才能让 OpenGL 来绘制文字。目前一些技术包括：通过 `GL_LINES` 来绘制字形、创建文字的 3D 网格、将带有文字的纹理渲染到一个 2D 方块中。

开发者最常用的一种方式是将字符纹理绘制到矩形方块上。绘制这些纹理方块其实并不是很复杂，然而检索要绘制的文字的纹理却变成了一项挑战性的工作。本教程将探索多种文字渲染的实现方法，并且着重对更加现代而且灵活的渲染技术（使用 FreeType 库）进行讲解。

## 经典文字渲染：使用位图字体

在早期渲染文字时，选择你应用程序的字体（或者创建你自己的字体）来绘制文字是通过将所有用到的文字加载在一张大纹理图中来实现的。这张纹理贴图我们把它叫做位图字体(Bitmap Font)，它包含了所有我们想要使用的字符。这些字符被称为字形(**glyph**)。每个字形根据他们的编号被放到位图字体中的确切位置，在渲染这些字形的时候根据这些排列规则将他们取出并贴到指定的 2D 方块中。



上图展示了我们如何从一张位图字体的纹理中通过对字形的合理取样（通过小心地选择字形的纹理坐标）来实现绘制文字“OpenGL”到 2D 方块中的原理。通过对 OpenGL 启用混合并让位图字体的纹理背景保持透明，这样就能实现使用 OpenGL 绘制你想要文字到屏幕的功能。上图的这张位图字体是使用 [Codehead 的位图字体生成器](#) 生成的。

使用这种途径来绘制文字有许多优势也有很多缺点。首先，它相对来说很容易实现，并且因为位图字体被预渲染好，使得这种方法效率很高。然而，这种途径并不够灵活。当你想要使用不同的字体时，你不得不重新生成位图字体，以及你的程序会被限制在一个固定的分辨率：如果你对这些文字进行放大的话你会看到文字的像素边缘。此外，这种方式仅局限于用来绘制很少的字符，如果你想让它来扩展支持 **Unicode** 文字的话就很不现实了。

这种绘制文字的方式曾经得益于它的高速和可移植性而非常流行，然而现在已经存在更加灵活的方式。其中一个是即将展开讨论的使用 **FreeType** 库来加载 **TrueType** 字体的方式。

## 现代文字渲染：使用 **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，被用于 **Mac OS X**、**Java**、**PlayStation** 主机、**Linux**、**Android** 等。**FreeType** 的真正吸引力在于它能够加载 **TrueType** 字体。

**TrueType** 字体不采用像素或其他不可缩放的方式来定义，而是一些通过数学公式(曲线的组合)。这些字形，类似于矢量图像，可以根据你需要的字体大小来生成像素图像。通过使用 **TrueType** 字体可以轻易呈现不同大小的字符符号并且没有任何质量损失。

**FreeType** 可以在他们的[官方网站](#)中下载到。你可以选择自己编译他们提供的源代码或者使用他们已编译好的针对你的平台的链接库。请确认你将 **freetype.lib** 添加到你项目的链接库中，同时你还要添加它的头文件目录到项目的搜索目录中。

然后请确认包含合适的头文件：

```
#include <ft2build.h>
```

```
#include FT_FREETYPE_H
```

### Attention

由于 **FreeType** 开发得比较早（至少在我写这篇文章以前就已经开发好了），你不能将它们的头文件放到一个新的目录下，它们应该保存在你包含目录的根目录

下。通过使用像 `#include <FreeType/ft2build.h>` 这样的方式导入 FreeType 可能会出现各种头文件冲突的问题。

FreeType 要做的事就是加载 TrueType 字体并为每一个字形生成位图和几个度量值。我们可以取出它生成的位图作为字形的纹理，将这些度量值用作字形纹理的位置、偏移等描述。

要加载一个字体，我们需要做的是初始化 FreeType 并且将这个字体加载为 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 函数在出现错误的情况下返回一个非零整数值。

一旦我们加载字体面完成，我们还要定义文字大小，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出的高度中动态计算出字形的宽度。

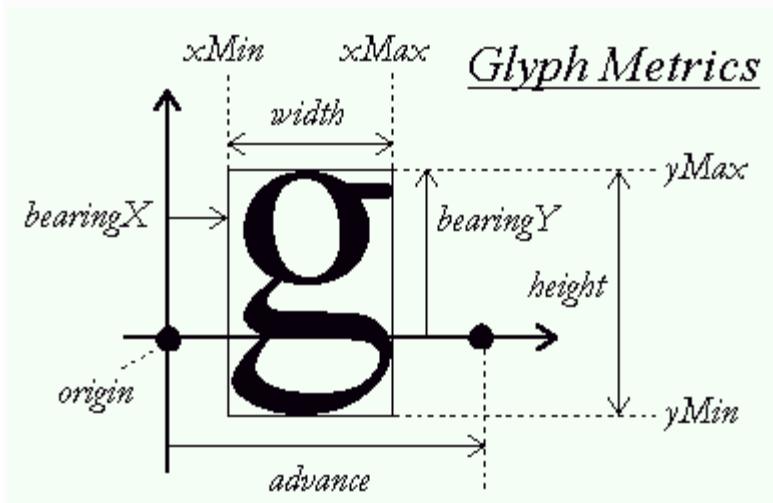
一个字体面中包含了所有字形的集合。我们可以通过调用 `FT_Load_Char` 函数来激活当前要表示的字形。这里我们选在加载字母字形'X'：

```
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` 设为一个加载标识，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` 来取得这个位图。

使用 FreeType 加载的字形位图并不像我们使用位图字体那样持有相同的尺寸大小。使用 FreeType 生产的字形位图的大小是恰好能包含这个字形的尺寸。例如生产用于表示'.'的位图的尺寸要比表示'X'的小得多。因此，FreeType 在加载字形的时候还生产了几个度量值来描述生成的字形位图的大小和位置。下图展示了 FreeType 的所有度量值的涵义。



每一个字形都放在一个水平的基线(Baseline)上，上图中被描黑的水平箭头表示该字形的基线。这条基线类似于拼音四格线中的第二根水平线，一些字形被放在基线上(如'x'或'a')，而另一些则会超过基线以下(如'g'或'p')。FreeType 的这些度量值中包含了字形在相对于基线上的偏移量用来描述字形相对于此基线的位置，字形的大小，以及与下一个字符之间的距离。下面列出了我们在渲染字形时所需要的度量值的属性：

| 属性       | 获取方式                                         |
|----------|----------------------------------------------|
| width    | <code>face-&gt;glyph-&gt;bitmap.width</code> |
| height   | <code>face-&gt;glyph-&gt;bitmap.rows</code>  |
| bearingX | <code>face-&gt;glyph-&gt;bitmap_left</code>  |
| bearingY | <code>face-&gt;glyph-&gt;bitmap_top</code>   |

| 属性      | 获取方式                   |
|---------|------------------------|
| advance | face->glyph->advance.x |

在我们想要渲染字符到屏幕的时候就能根据这些度量值来生成对应字形的纹理了，然而每次渲染都这样做显然不是高效的。我们应该将这些生成的数据储存在应用程序中，在渲染过程中再去取。方便起见，我们需要定义一个用来储存这些属性的结构体，并创建一个字符表来存储这些字形属性。

```
struct Character {
 GLuint TextureID; // 字形纹理 ID
 glm::ivec2 Size; // 字形大大小
 glm::ivec2 Bearing; // 字形基于基线和起点的位置
 GLuint Advance; // 起点到下一个字形起点的距离
};

std::map<GLchar, Character> Characters;
```

本教程本着让一切简单的目的，我们只生成表示 128 个 ASCII 字符的字符表。并为每一个字符储存纹理和一些度量值。这样，所有需要的字符就被存下来备用 了。

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); // 禁用 byte-alignment 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;
}

// 生成字形纹理
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);

// 将字符存储到字符表中备用

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` 循环中我们将所有 ASCII 中的 128 个字符设置合适的字形。为每一个字符创建纹理并存储它们的部分度量值。有趣的是我们这里将纹理的格式设置为 **GL\_RED**。通过字形生成的位图是 8 位灰度图，他的每一个颜色表示为一个字节。因此我们需要将每一位都作为纹理的颜色值。我们通过创建每一个字节表示一个颜色的红色分量(颜色分量的第一个字节)来创建字形纹理。我们想用一个字节来表示纹理颜色，这需要提前通知 OpenGL

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

OpenGL 要求所有的纹理颜色必须是 4 字节队列，这样纹理的内存大小就一定是 4 字节的整数倍。通常这并不会出现什么问题，因为通常的纹理都有 4 或者 4 的整数倍的储存大小，但是现在我们只用一位来表示每一个像素颜色，此时的纹理可能有任意内存长度。通过将纹理解压参数设为 1，这样才能确保不会有内存对齐的解析问题。

当你处理完字形后同样不要忘记清理 FreeType 的资源。

```
FT_Done_Face(face);
```

```
FT_Done_FreeType(ft);
```

## 着色器

我们需要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。顶点着色器将会将位置坐标与投影矩阵相乘，并将纹理坐标转发给片段着色器：

```
#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，另一个是文字的颜色，我们可以同调整它来改变最终输出的字体颜色。我们首先从位图纹理中采样，由于纹理数据中仅存储着红色分量，我们就通过 `r` 分量来作为取样颜色的 `alpha` 值。结果值是一个字形背景为纯透明，而字符部分为不透明的白色的颜色。我们将此颜色与字体颜色 `uniform` 值相乘就得到了要输出的字符颜色了。

当然我们必须开启混合才能让这一切行之有效：

```

glEnable(GL_BLEND);

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

```

至于投影矩阵我们将使用一个正交投影矩阵。对于文字渲染我们通常都不需要进行透视，使用正交投影也能保证我们所有的顶点坐标设置有效：

```
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可以通过设置 `0.0f~600.0f` 的纵坐标来表示顶点在窗口中的垂直位置。这意味着现在点 `(0.0,0.0)` 表示左下角而不再是窗口正中间。

最后要做的事是创建一个 VBO 和 VAO 用来渲染方块。现在我们分配足够的内存来初始化 VBO 然后在我们渲染字符的时候再来更新 VBO 的内存。

```
GLuint VAO, VBO;

glGenVertexArrays(1, &VAO);

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 方块需要 6 个顶点，每个顶点又是由一个 4 维向量（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  $6 \times 4$  个 float 的大小。由于我们会在绘制字符时更新这断内存，所以我们将内存类型设置为 `GL_DYNAMIC_DRAW`

## 渲染一行文字

要渲染一个字符，我们从之前创建的字符表中取出一个字符结构体，根据字符的度量值来计算方块的大小。根据方块的大小我们就能计算出 6 个描述方块的顶点，并使用 `glBufferSubData` 函数将其更新到 VBO 所在内内存中。

我们创建一个函数用来渲染一行文字：

```
void RenderText(Shader &s, std::string text, GLfloat x, GLfloat y,
 GLfloat scale, glm::vec3 color)
{
 // 激活合适的渲染状态
 s.Use();

 glUniform3f(glGetUniformLocation(s.Program, "textColor"),
 color.x, color.y, color.z);

 glEnableTexture(GL_TEXTURE0);

 glBindVertexArray(VAO);

 // 对文本中的所有字符迭代
 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;

 // 当前字符的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 }
};

// 在方块上绘制字形纹理

glBindTexture(GL_TEXTURE_2D, ch.textureID);

// 更新当前字符的VBO

glBindBuffer(GL_ARRAY_BUFFER, VBO);

glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices),
vertices);

glBindBuffer(GL_ARRAY_BUFFER, 0);

// 绘制方块

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; //(2^6 = 64)
}

glBindVertexArray(0);

glBindTexture(GL_TEXTURE_2D, 0);
```

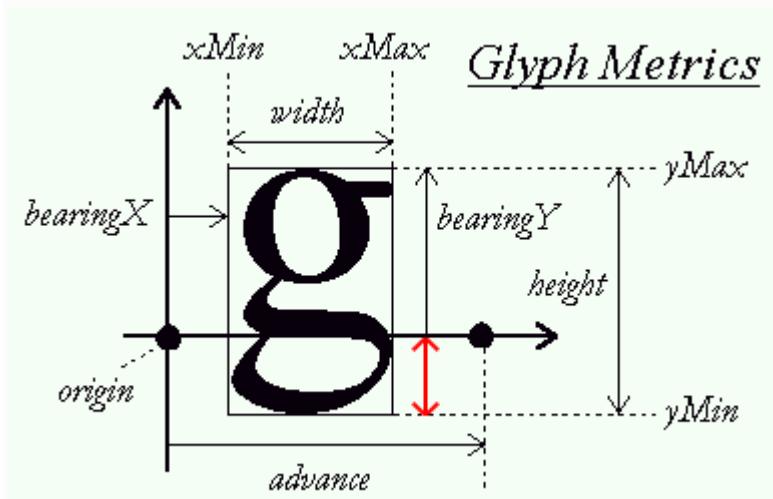
}

这个函数的内容注释的比较详细了：我们首先计算出方块的起点坐标和它的大小，并为该方块生成一个 6 个顶点；注意我们在缩放的同时会将部分度量值也进行缩放。接着我们更新方块的 VBO、绑定字形纹理来渲染它。

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(诸如'p'或'q')需要被渲染到基线一下，因此字形方块也应该讲 y 位置往下调整。调整的量就是便是字形度量值中字形的高度和 BearingY 的差：



要计算这段偏移量的距离，我们需要指出是字形在基线以下的部分最底断到基线的距离。在上图中这段距离用红色双向箭头标出。如你所见，在字形度量值中，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。这段距离有可能是 0.0f(如'x'字符)，或是超出基线底部的距离的长度(如'g'或'j')。

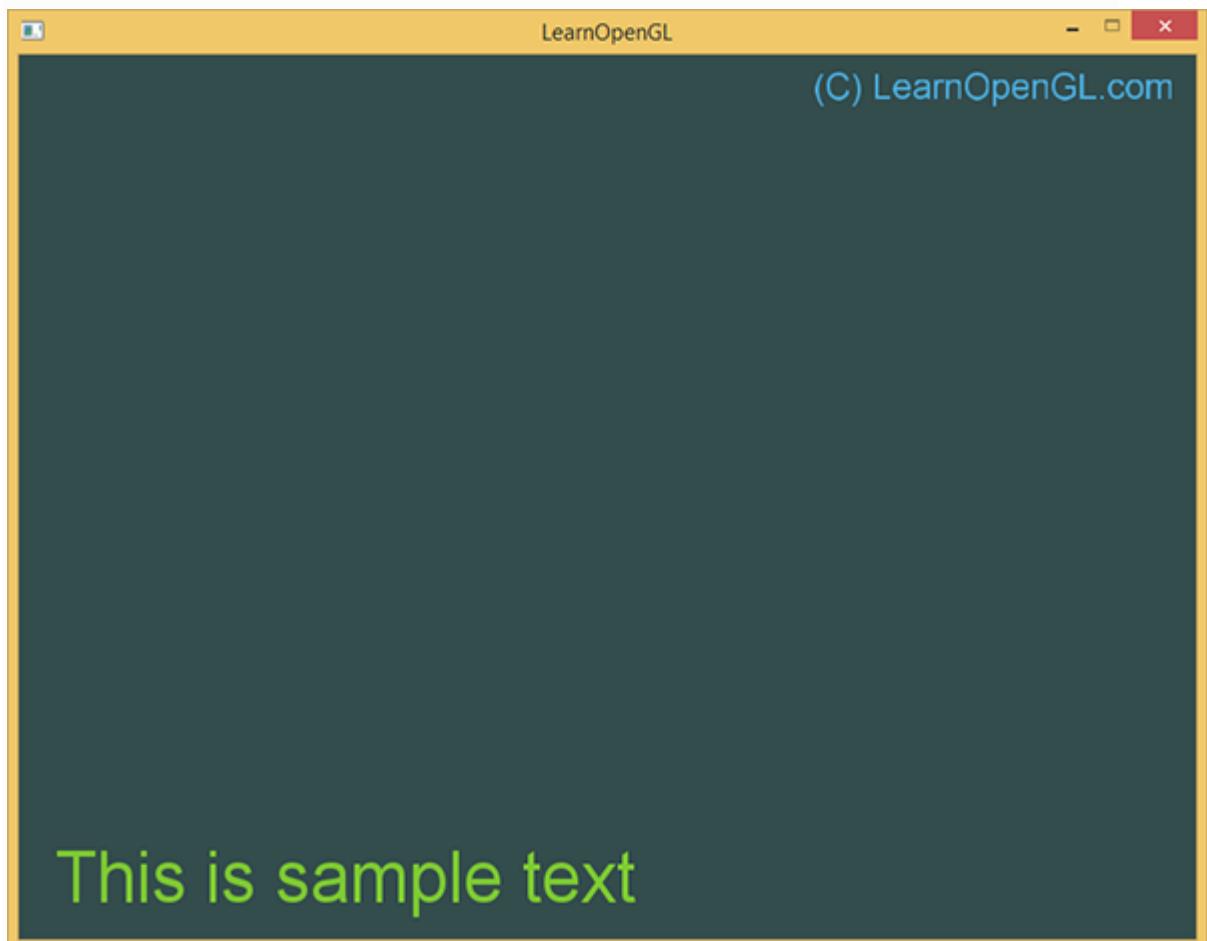
如果你每件事都做对了，那么你现在已经可以使用下面的句子成功地渲染字符串了：

```
RenderText(shader, "This is sample text", 25.0f, 25.0f, 1.0f,
```

```
glm::vec3(0.5, 0.8f, 0.2f));
```

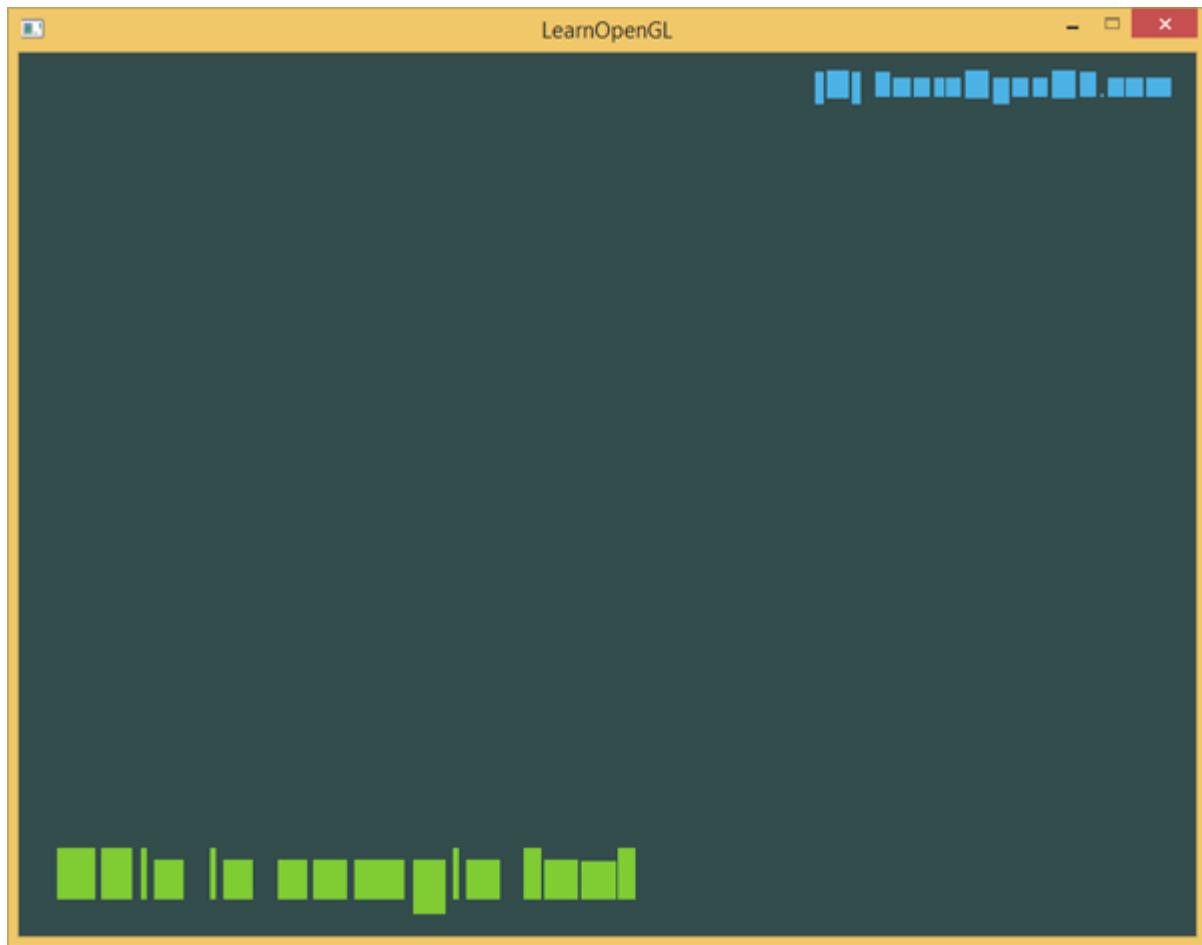
```
RenderText(shader, "(C) LearnOpenGL.com", 540.0f, 570.0f, 0.5f,
glm::vec3(0.3, 0.7f, 0.9f));
```

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](#)。

通过关闭字形纹理的绑定，能够给你对文字方块的顶点计算更好的理解，它看上去像这样：



这样你就能清楚地看到那条传说中的基线了。

## 关于未来

本教程演示了如何使用 **FreeType** 绘制 **TrueType** 文字。这种方式灵活、可缩放并支持多种字符编码。然而，你的应用程序可能并不需要这么强大的功能，性能更好的点阵字体也许是更可取的。当然你可以结合这两种方法通过动态生成位图字体中所有字符字形。这节省了从大量的纹理渲染器开关和基于每个字形紧密包装可以节省相当的一些性能。

另一个使用 **FreeType** 字体的问题是字形纹理是对应着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。可以通过将每个像素设为最近的字形轮廓的像素，而不是直接设为实际栅格化的像素，可以减轻这些问题。这项技术被称为 **signed distance fields**，Valve 在几年前发表过一篇了[论文](#)，探讨了他们通过这项技术来获得好得惊人的 3D 渲染效果。

