计算机图形学实验指导书

陈鹏 编写

安徽师范大学物理与电子信息学院
自动化教研室
二０一六年四月

前　言

《计算机图形学》是电子信息类、自动化类等专业的专业方向课，为学习后续课程准备必要的计算机图形学知识，主要培养学生掌握图形学理论基础知识和图形程序设计方法、培养三维图形编程技能、引领学生初步具备解决绘制三维图形界面、绘制真实感三维图形等方面的知识和能力，课程的主要内容包括图形渲染流水线、扫描转换算法、裁剪算法、二维及三维图形绘制、投影变换等。

本实验指导书是为了配合“计算机图形学”课程的教学而编写的．指导书包括 4 个实验，每个实验均由实验目的、基本概念和实验内容三个部分构成．其中实验内容包括“验证性内容”和“设计性内容”两部分．“验证性内容”提供了参考程序，主要是加深对基本概念的理解、增加感性认识．“设计性内容”则是考察学生对所学知识灵活运用的能力．为了帮助同学们对所学知识的理解，在基本概念中提供了实验内容的实现原理即相关的知识。

实验操作是本课程教学内容的一个重要方面．因此，要求同学在每次实验操作之前，仔细阅读实验指导书和教材中的相关内容，复习和掌握好算法工具（C++语言），实验前，要求同学们读懂“验证性内容”，准备好“设计性内容”的方案．操作过程中细心观察各种现象和结果，做好记录，最终完成实验报告．

实验一　用 opengl 绘制三角形 1
一、实验目的 1
二、基本概念 1
三、实验内容 9
实验二　着色器及纹理映射 24
一、实验目的 24
二、基本概念 24
三、实验内容 44
实验三　几何变换 66
一、实验目的 66
二、基本概念 66
三、实验内容 78
实验四　坐标系统与相机漫游 106
一、实验目的 106
二、基本概念 106
三、实验内容 118

实验一　用 opengl 绘制窗口及基本图形

一、实验目的

1．了解和掌握 opengl 编程的基本概念和开发环境的配置方法.

2．掌握 GLFW 绘制窗口的方法、并在 GLFW 绘制的窗口中用 opengl 函数绘制三角形.

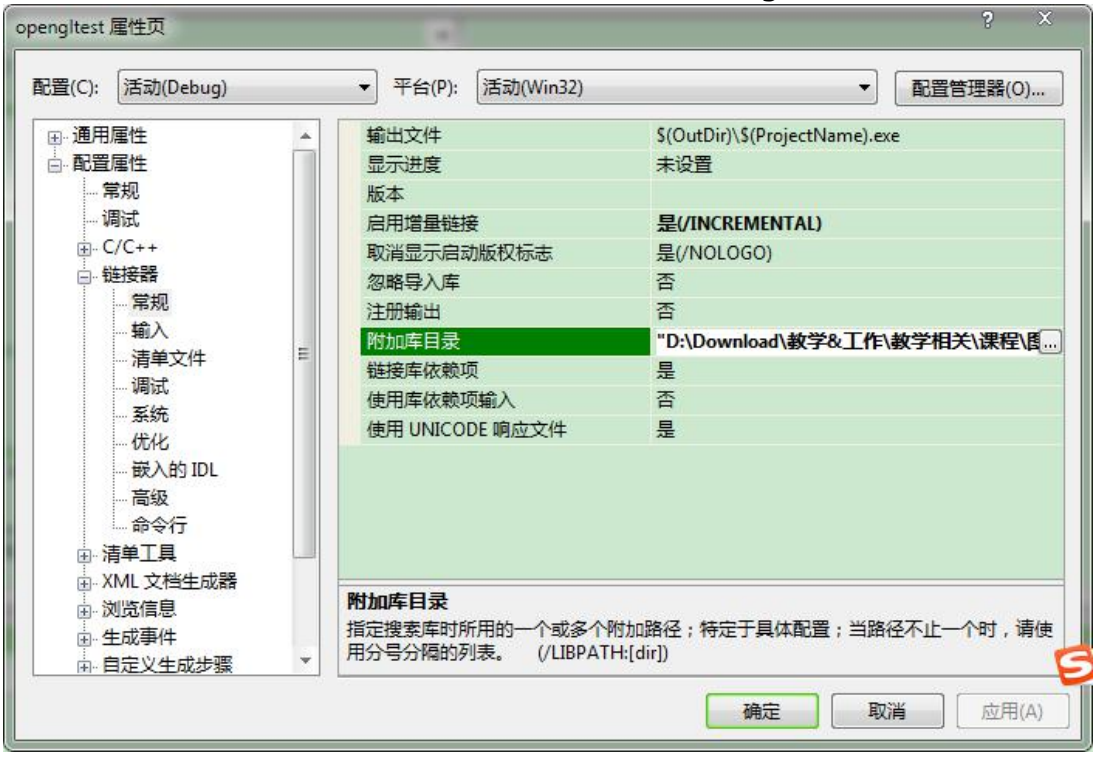
3. 掌握 OPENGL 图形流水线的概念。

二、基本概念

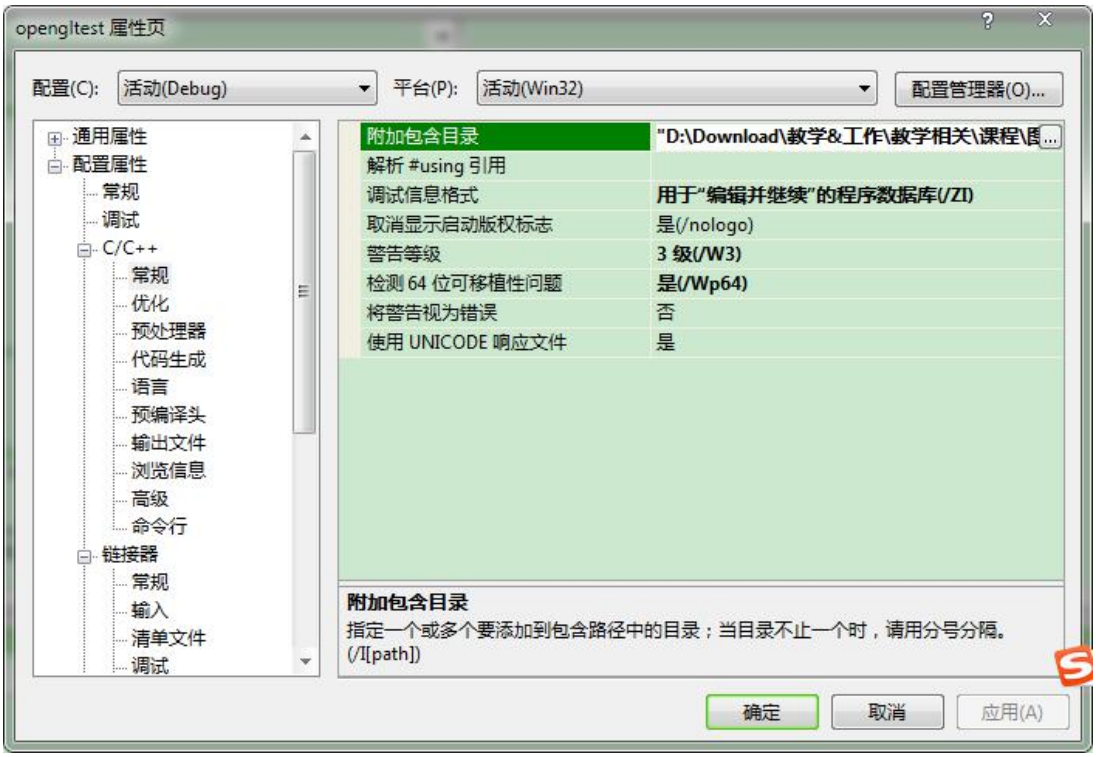
1．第三方库的配置和使用。

A）GLFW库

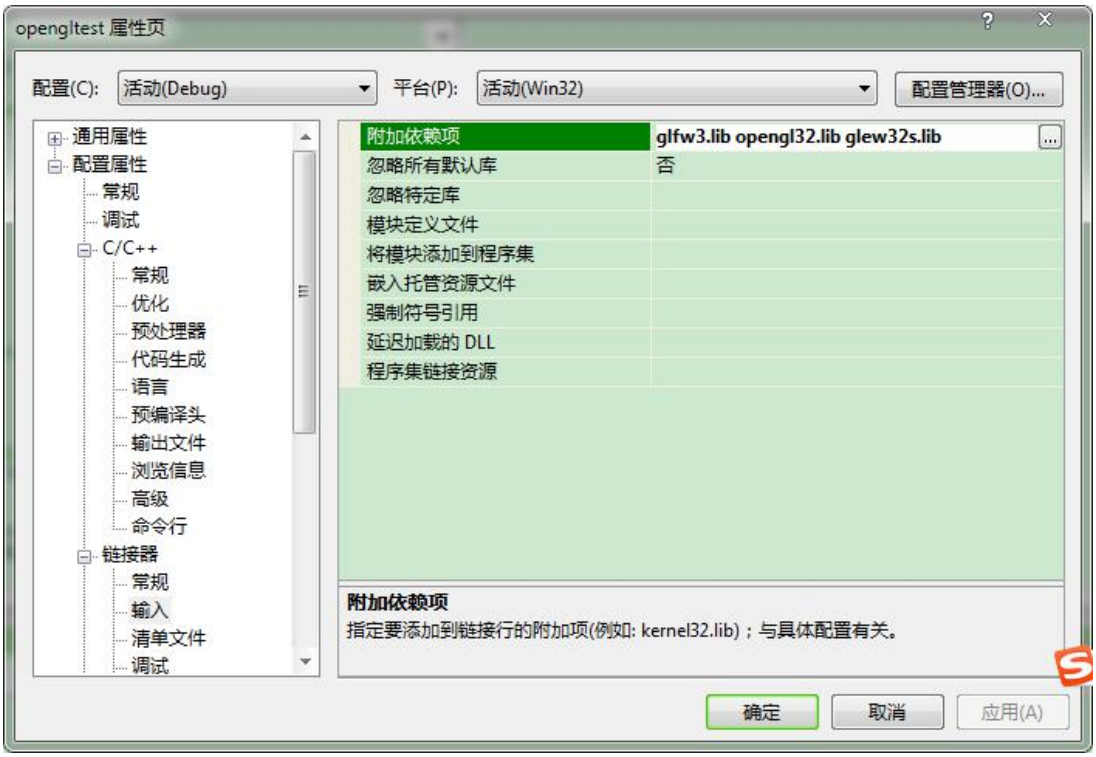
GLFW是一个用c写的库，用来供opengl使用，以提供基本的窗口及输入输出等功能（同GLUT库类似）。为了使用GLFW库，需要下载相应的库，或从源码中编译出库来。然后在我们建立的项目中，1）设置glfw3.lib所在的路径：



2）设置头文件所在路径：



3）设置所要包含的库文件：



B)GLEW 库

类似的，要在我们的项目中使用GLEW库，则需要相应的设置头文件目录，库

文件目录及设置所要包含的库文件glew32s.lib。由于opengl是一个标准，规定了各个函数的输入和输出，每个显卡生产厂家的具体实现则可能不同，各种功能是否完整提供也无法确定。需要编程人员确定各个函数的地址以便在以后使用。

在 windows 中方法如下：

```
// Define the function's prototype
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);
// Find the function and assign it to a function pointer
GL_GENBUFFERS glGenBuffers = (GL_GENBUFFERS)wglGetProcAddress("glGenBuffers
    ");
// Function can now be called as normal
GLuint buffer;
glGenBuffers(1, &buffer);
```

可以看出，要对所使用的每个函数都采取这种方式的话，代码将很复杂。而 GLEW 库则可以简化这个工作。

C)opengl 库

Opengl 库在安装 visual studio 时已经默认的将库文件及头文件放在了项目所能找到的系统目录位置中了，因此不需要在项目中设置头文件目录和库文件目录所在路径。只需要设置所要包含的库文件即可。注意的是，我们采用的 opengl 版本为 3.3 版本，如果低于此版本则参考程序会出错。另外，如果显卡驱动程序版本过低，不支持 opengl3.3 版本，则窗口也建立不成功（可以用 glviewer 查看版本）。此时需要更新显卡驱动程序。

注：静态库 lib 和动态库 DLL 的区别：

> **Static** linking of a library means that during compilation the library will be integrated in your binary file. This has the advantage that you do not need to keep track of extra files, but only need to release your single binary. The disadvantage is that your executable becomes larger and when a library has an updated version you need to re-compile your entire application.
> **Dynamic** linking of a library is done via `.dll` files or `.so` files and then the library code and your binary code stays separated, making your binary smaller and updates easier. The disadvantage is that you'd have to release your DLLs with the final application.

2. GLFW 窗口创建

a)首先创建一个空的控制台项目。然后按照上面所述，配置好项目中相应的库路径等参数。在 main.cpp 文件中，包含以下的头文件：

```
// GLEW
#define GLEW_STATIC
#include <GL/glew.h>
// GLFW
#include <GLFW/glfw3.h>
```

b)下面对 GLFW 窗口进行初始化：

```
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;
}
```

其中，glfwWindowHint是用来对GLFW进行参数配置的函数。具体参数含义可以参考GLFW窗口相关的文档。

c)初始化以后，我们需要创建一个GLFW窗口对象：

```
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);
```

创建窗口的函数glfwCreateWindow中，前两个参数分别是窗口的宽和高，第三个参数是窗口的名称。暂时不用理会后两个参数。

注：如果vc++版本中不认识nullptr，将nullptr换为NULL即可。

d)由于我们采用GLEW来管理所有的opengl函数，因此下面需要先初始化GLEW，再进行opengl函数调用：

```
glewExperimental = GL_TRUE;
```

```
if (glewInit() != GLEW_OK)
{
    std::cout << "Failed to initialize GLEW" << std::endl;
    return -1;
}
```

e)视区（viewport）

在绘制之前，我们还需要告诉opengl绘制区域的尺寸。这是通过glViewport这个函数实现的：

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

前两个参数设置了绘制窗口的左下角位置。第三和第四个参数分别设置了绘制窗口的宽和高。

此处我们将绘制区的宽和高设为同GLFW窗口一样，也可以设的比GLFW窗口小一些，这样就可以在绘制区绘制opengl图形，在绘制区以外的地方显示其他内容。

注：opengl内部的窗口映射机制:

> Behind the scenes OpenGL uses the data specified via `glViewport` to transform the 2D coordinates it processed to coordinates on your screen. For example, a processed point of location `(-0.5,0.5)` would (as its final transformation) be mapped to `(200,450)` in screen coordinates. Note that processed coordinates in OpenGL are between -1 and 1 so we effectively map from the range (-1 to 1) to (0, 800) and (0, 600).

f)事件循环

我们不是绘制一个窗口，然后就结束了。而是希望绘制窗口后，等待用户的事件，然后根据事件的内容决定我们的动作。这一过程由以下程序实现：

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

在这个循环中，每次循环开始前，glfwWindowShouldClose都要检查是否GLFW窗口被命令关闭。如果是，则结束循环，然后我们的应用程序就结束了。

循环体中的glfwPollEvents函数会检查是否有事件发生（如鼠标移动，按键按下等），并调用相应的事件处理函数（我们可通过callback函数设置）。glfwSwapBuffers函数将交换缓存并将缓存中内容在屏幕上显示。

注：双缓存机制:

> **Double buffer**
> When an application draws in a single buffer the resulting image might display flickering issues. This is because the resulting output image is not drawn in an instant, but drawn pixel by pixel and usually from left to right and top to bottom. Because these images are not displayed at an instant to the user, but rather via a step by step generation the result may contain quite a few artifacts. To circumvent these issues, windowing applications apply a double buffer for rendering. The **front** buffer contains the final output image that is shown at the screen, while all the rendering commands draw to the **back** buffer. As soon as all the rendering commands are finished we **swap** the back buffer to the front buffer so the image is instantly displayed to the user, removing all the aforementioned artifacts.

g)最后，则需要结束程序：

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

整个完整程序可参考实验内容中的参考程序1。

3．输入处理及渲染

a)输入处理

我们可以利用GLFW中的回调函数对输入的事件进行处理。回调函数实际上是一个函数指针。我们可以设置的一个回调函数是处理键盘事件的KeyCallback回调函数，其函数原型为：

```
void key_callback(GLFWwindow* window, int key, int scancode, int action,
    int mode);
```

其中第二个参数key表明是哪个键。第四个参数action是指该键是“按下”还是“释放”。最后一个参数mode是指是否有功能键如alt，ctrl，shift等被按下。只要用户按了某个键，GLFW就会调用这个函数，并设置好正确的参数。我们可以编写这个函数，以对按键事件作出处理：

```
void key_callback(GLFWwindow* window, int key, int scancode, int action,
    int mode)
{
    // When a user presses the escape key, we set the WindowShouldClose
    property to true,
    // closing the application
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

例如，以上代码说明，当空格键被按下时，我们将关闭GLFW。

最后，我们还需要注册这个回调函数：

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

当然，还可以有很多回调函数，例如处理窗口尺寸变化的回调函数等。注册回调函数的时机是在我们创建窗口后，事件循环开始前。

b）渲染（即绘制）

在什么地方绘制呢？因为我们希望在每个事件循环时都要执行所有的绘制命令，因此可以将绘制命令放在如下地方：

```
// Program loop
while(!glfwWindowShouldClose(window))
{
    // Check and call events
    glfwPollEvents();

    // Rendering commands here
    ...

    // Swap the buffers
    glfwSwapBuffers(window);
}
```

为了看看结果如何，我们可以在这里用我们指定的颜色清空屏幕；

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

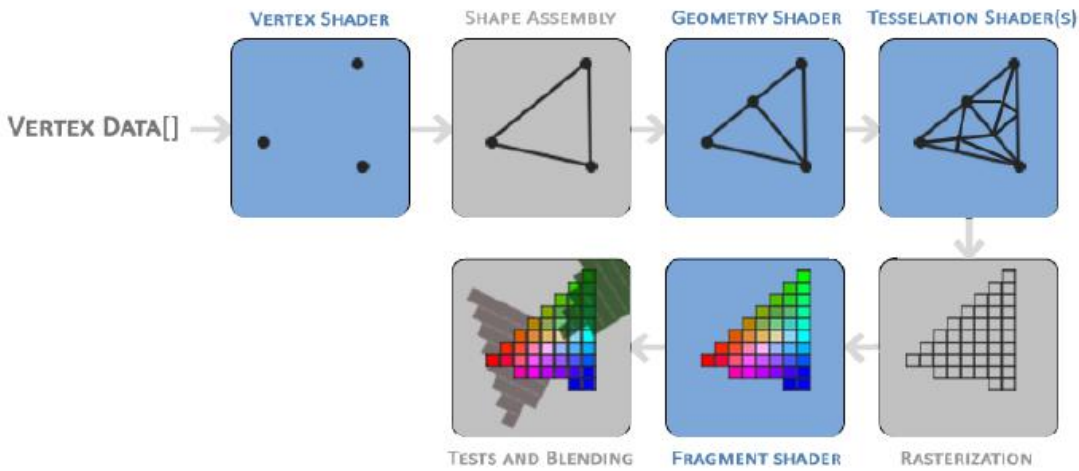
还记得吗？opengl是一个状态机。这里第一个函数是用来设置清空颜色（即设置状态的函数）。第二个函数是用这个颜色对缓存进行清空（即使用状态的函数）。

4.opengl 图形流水线（graphics pipeline）

4.1图形流水线简介

Opengl中所有的东西都是三维的，而屏幕则是二维的。因此opengl的很大一部分工作是将所有的三维坐标转成同屏幕相适应的二维像素点。这个转换过程由opengl的图形流水线来管理。图形流水线可以分为两大部分：首先将3D坐标转为2D坐标。然后将2D坐标转为实际的带有颜色的2D像素点。这两大部分可以进一步分成许多步，每一步工作都很具体（即有个专门的函数），前一步的输出作为后一步的输入，而且这些步可以很容易的并行化。因为可并行化的性质，因此现在的图形卡都有上千颗小处理核，流水线步骤中的每一步都有许多小程序来执行。运行这些小程序也称为着色器（shaders）。

一些着色器可以由开发者来配置，也就是说，自己写着色器替换掉默认的着色器。这就使得我们可以对流水线进行更细颗粒度的控制。而且我们写的着色器是在GPU中运行，因此可以节省大量的CPU时间。着色器用GLSL(Opengl Shading Language)来写。图形流水线的示意图如下：



流水线图，蓝色的为可配置部分。

下面对流水线的各个步骤做简要介绍。

a)输入数据

输入流水线的数据位Vertex Data。比如组成三角形的3D坐标（每点一个，共3个）形成的顶点数组。这个顶点主要由3D坐标组成，还可以有数据的属性。为简单起见，我们假设顶点为3D坐标及颜色组成。为了让Opengl知道这些数据的用途，我们需要暗示opengl这些数据的绘制类型。比如是用来绘制三个点，还是三角形，还是线段？这些暗示也称为图元（primitives），如：GL_POINTS,GL_TRIANGLES,GL_LINE_STRIP等。

b)vertex shader

流水线的第一个部分是vertex shader，其输入是顶点。Vertex shader的主要目的是将3D坐标转换为不同的3D坐标，而且可以对顶点的属性数据进行基本处理。

c)primitive assembly

vertex shader输出的顶点作为primitive assembly的输入数据。然后primitive assembly将这些点组合成图元，这里是三角形。

d)geometry shader

primitive assembly输出的由顶点组成的基本图元会传给geometry shader。然后geometry shader会产生新的点来生成新的图元。这里是生成了第二个三角形。

e)tessellation shaders

tessellation shaders能够将输入的图元分成许多更小的图元。这可以让我们能够生成更细致平滑的效果。比如距离观察者越近就生成更多的小图元，从而改善观察效果。

f)rasterization stage

tessellation shaders输出的数据作为输入送入rasterization stage，由其将这些图元映射到最终的屏幕像素点上，形成片元(fragments)。这些片元将供下个流水线步骤fragment shader使用。在fragment shader运行前，还会执行裁剪操作，以剪除所有不再我们视野内的片元，提高性能。

> A fragment in OpenGL is all the data required for OpenGL to render a single pixel.

g)fragment shader

fragment shader的主要目的是计算像素点的最终颜色。这也是高级opengl效果呈现的阶段。通常，fragment shader包含有3D场景的数据（如光照，阴影，光的颜色等），用来计算像素点的最终颜色。

h)alpha test and blending

当所有的颜色值都计算完毕后，最终会来到最后一步，即alpha test and blending步骤。在这个步骤里，将检查片元的深度值，以确定最终的片元是在其他物体的前方还是后方。在后方的话，是否要舍去不再显示。这个阶段还会检查alpha值（决定一个物体的透明度），并据此将不同物体的颜色进行混合（blends）。因此，当绘制多个三角形时，即使计算出了一点的片元颜色，可能最终显示的结果颜色会完全不同。

在现代Opengl编程中，GPU没有默认的vertex shader 及fragment shader，需要编程者自己定义。正因为如此，现代opengl编程更困难一些。不过等我们最终能够绘制出自己的三角形后，我们会对图形编程有更深入的理解。

4.2图形流水线详解

4.2.1 vertex input

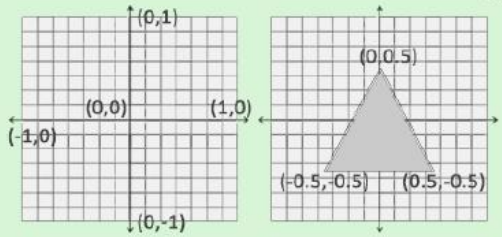
为了绘制图形，我们需要给opengl提供数据。Opengl是个3D的图形库，因此所有的坐标点都是3维的。Opengl不是简单的将所有我们的3D坐标转换为我们屏幕上的2D像素，而是只处x，y，z轴上在-1到1之间的数据。所有在这个坐标范围内（即一个立方体，也称为规范化设备坐标）的数据均会显示在我们屏幕上，在这个之外的则不会。

我们要绘制一个三角形，因此用一个数组来指定三个点的坐标位置(均在规范化坐标范围内)。

```
// Set up vertex data (and buffer(s)) and attribute pointers
GLfloat vertices[] = {
```

```
    -0.5f, -0.5f, 0.0f, // Left
     0.5f, -0.5f, 0.0f, // Right
     0.0f,  0.5f, 0.0f  // Top
};
```

注：规范化坐标

> **Normalized Device Coordinates (NDC)**
> Once your vertex coordinates have been processed in the vertex shader, they should be in normalized device coordinates which is a small space where the x, y and z values vary from -1.0 to 1.0. Any coordinates that fall outside this range will be discarded/clipped and won't be visible on your screen. Below you can see the triangle we specified within normalized device coordinates (ignoring the z axis):
>
> 
>
>
> Unlike usual screen coordinates the positive y-axis points in the up-direction and the (0,0) coordinates are at the center of the graph, instead of top-left. Eventually you want all the (transformed) coordinates to end up in this coordinate space, otherwise they won't be visible.
>
> Your NDC coordinates will then be transformed to screen-space coordinates via the viewport transform using the data you provided with glViewport. The resulting screen-space coordinates are then transformed to fragments as inputs to your fragment shader.

我们定义好了数据后，需要将其送到下一个流水线步骤里，及vertex shader。这需要在GPU里创建一片保存我们数据的内存，对其进行配置，以便让opengl知道怎么解释这片内存中的数据，以及如何将这些数据送到显卡中。

我们通过顶点缓存对象（vertex buffer objects（VBO））来管理这片内存，这样的好处是可以保存大量的顶点数据在GPU的存储器里，从而一次向显卡发送大量数据。从CPU向显卡发数据比较慢，因此尽可能一次多发些数据。

顶点缓存对象也是我们介绍的第一个Opengl对象。它有个唯一的ID号，我们可以这样来生成这个ID号：

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

Opengl有许多种类型的缓存对象，顶点缓存对象的类型是GL_ARRAY_BUFFER。我们通过以下方法将我们新创建的缓存绑定到GL_ARRAY_BUFFER类型上：

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

这之后的针对GL_ARRAY_BUFFER对象的缓存调用，都将用来对目前所绑定缓存（即VBO）进行配置。然后，我们将以前所定义的顶点数据拷贝到缓存区中：

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

第一个参数是缓存的类型，第二个参数是数据大小，第四个参数说明了如何对此数据管理。

- GL_STATIC_DRAW: the data will most likely not change at all or very rarely.
- GL_DYNAMIC_DRAW: the data is likely to change a lot.
- GL_STREAM_DRAW: the data will change every time it is drawn.

现在我们已经将顶点数据保存在显卡的存储器中了，并由一个称为VBO的顶点缓存目标来管理。下面我们想创建vertex shader 和 fragment shader，以实际处理这个数据。

4.2.2 vertex shader

Vertex shader 是一种我们编程人员可以编程的着色器。现代opengl要求我们至少要建立vertex shader和fragment shader。首先，我们要用一种GLSL（opengl shading language）语言来编写着色器，然后对着色器进行编译。下面是GLSL编写的着色器代码：

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

可以看出，GLSL语言同c语言很相似。每个着色器开始都声明它的版本。由于我们使用opengl3.3及以上版本，因此我们声明了330。Core是指我们采用了核心模式（core profile），即完全新式纯可编程流水线模式，但兼容废弃特性。第二行的in关键字，用来声明所有输入顶点的属性。目前我们只关心坐标数据，因此我们只需要一个顶点属性。GLSL有个向量类型，可以包含1到4个浮点数。由于我们的每个顶点是个3D的坐标，因此我们创建一个类型为vec3的变量position。我们还设置了输入变量的位置：layout（location = 0）。Vertex shader的输出是一个vec4类型的变量gl_Position。这里我们将第4维设置为1。这是一个最简单的vertex shader了，因为我们什么都没有做，只是将输入的数据传递给输出。

现在我们要编译这个着色器。为了让opengl能够使用它，我们要在运行时动态的编译我们写的着色器代码。首先我们创建一个着色器对象，用ID来标识。

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

然后我们将着色器源代码付给这个着色器对象，并编译它：

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

我们也许想检查下是否编译成功了。可以用以下代码检查：

```
// Check for compile time errors
GLint success;
GLchar infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

用glGetShaderiv来检查是否成功。如果编译失败，在可以用glGetShaderInfoLog函数来得到错误信息并显示出来。

4.2.3 fragment shader

Fragment shader是用来计算我们输出像素的颜色的，为简单起见，我们将所有的输出像素颜色都设为橘红色：

```
const GLchar* fragmentShaderSource = "#version 330 core\n"
    "out vec4 color;\n"
    "void main()\n"
    "{\n"
    "color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
    "}\n\0";
```

编译方法同vertex shader类似。具体可以参考参考代码1.2。

4.2.4着色器程序（shader program）

着色器程序对象是多个着色器链接以后的结果。为了能使用我们编译好的着色器，我们必须将他们连接到着色器程序对象里，并将这个着色器程序激活。这样当我们绘制命令发出后，将会使用这个激活的着色器。首先创建一个着色器程序对象：

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

然后将以前编译好的着色器附加在这个程序对象里，并用glLinkProgram链接它们：

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

类似的，我们可以用glGetShaderiv来检查是否链接成功：

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

然后我们可以用以下代码激活创建的程序对象：

```
glUseProgram(shaderProgram);
```

这之后的着色器及绘制命令都会使用这个程序对象（以及我们的着色器）了。

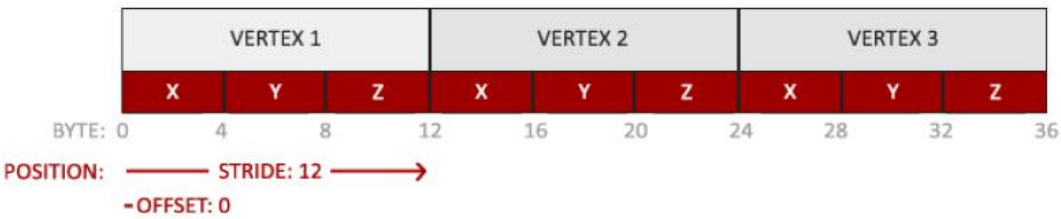
将着色器对象链接到程序对象后，着色器对象就可以删除了：

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

现在，我们已经将顶点数据输入到GPU里，并让GPU知道怎么用vertex 及fragment着色器处理这些顶点数据。不过opengl还不知道怎么样解释内存里的顶点数据，已经怎样将顶点数据同顶点着色器的属性联系起来。下面让我们来告诉opengl怎么做。

4.2.5连接顶点属性

虽然通过vertex 着色器，我们可以用顶点属性的方式输入任何我们想输入的数据，可是我们也不得不手工确定哪些输入数据对应哪些顶点属性。也就是说，我们需要让opengl知道如何在解释顶点数据。我们的顶点缓存区形式如下：



现在我们就来告诉opengl怎么解释我们的输入数据：

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

glVertexAttribPointer函数参数较多，第一个参数说明了我们想配置那个顶点属性。记得前面我们在vertex shader里用layout(location = 0)指定了position顶点属性中的位置属性。也就是说，将顶点属性中的位置设为了0.由于我们想传递数据给这个顶点属性，因此我们将第一个参数设为0。

第二个参数指定了顶点属性的大小。这个顶点属性是一个vec3的3D向量，因此第二个参数为3。

第三个参数指定了数据的类型。第四个参数指明我们是否希望对数据归一化。如果设为GL_TRUE，则所有数据都将被映射到0到1之间（对无符号数）或-1至1之间（对有符号数）。

第五个参数也称为跨度，告诉我们相邻顶点数据之间的距离。因为两个顶点属性之间相隔3个坐标点，因此我们设为3*sizeof（GLfloat）。最后一个参数要求GLvoid*类型，因此我们要做个类型转换。这个参数说明位置数据在缓存中的偏移量。由于数组中的第一个位置就是我们的位置数据，因此我们将这个参数设为0。

glEnableVertexAttribArray是用来激活顶点属性的（默认是不激活的）。其参数为顶点属性的位置。

从此处开始，我们已经将一切工作都准备好了：我们用一个顶点缓存对象对顶点数据初始化，建立了vertex shader和fragment shader，并告诉opengl怎样将顶点数据同vertex shader的顶点属性联系起来。下面就可以用opengl绘制物体了：

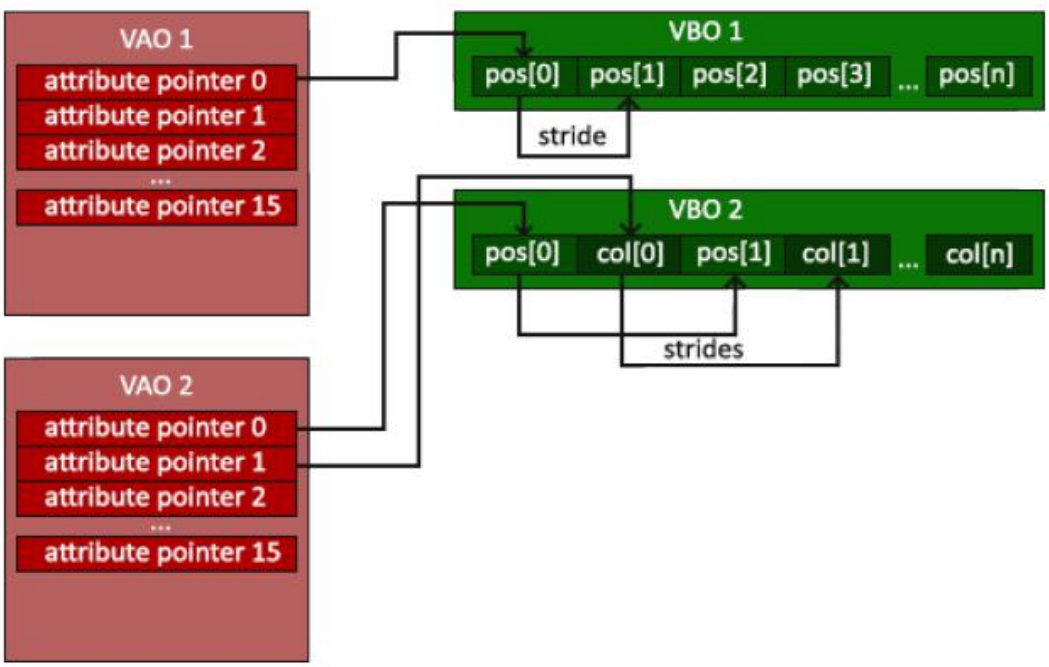
每次绘制物体时，这个过程就要重复一遍。如果我们有5个顶点属性，100多个不同的物体奥绘制，则将缓存对象绑定好，对每个缓存对象配置好所有的顶点属性就是一个很繁琐的过程了。如果能将这些状态

都保存到一个对象里，然后绑定这个对象就可以装载它的状态多好？下面我们就来做这个事情。

4.2.6顶点数组对象(VAO)

可以像绑定顶点缓存对象（VBO）一样绑定顶点数组对象（vertex array objectt（VAO）），绑定后，任何调用的顶点属性都将被保存在VAO里。这样，我们配置顶点属性指针时，只需调用一次配置函数。当我们想绘制物体时，我们只要绑定相应的VAO即可。这样，我们在改变不同的顶点数据和顶点属性的配置时，就可以简单的绑定不同的VAO了。所有的状态都保存在VAO里。

VAO保存的内容见以下示意图：



同VAO相关的过程：

- 调用glEnableVertexAttribArray 或 glDisableVertexAttribArray。
- 通过函数glBufferDatar配置顶点属性。
- 通过glVertexAttribPointer函数将VBO同顶点属性联系起来。

生成VAO的过程同生成VBO的过程类似：

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

要使用VAO，我们所要做的就是用glBindVertexArray函数将这个VAO绑定起来。从此，我们就可以绑定/配置相应的VBO以及属性指针，然后可以解除VAO绑定以备后用。当我们要绘制一个物体时，我们只需要绑定这个具有相应配置的VAO即可。代码类似下面：

```
// ..:: Initialization code (done once (unless your object frequently
    changes)) :: ..
// 1. Bind Vertex Array Object
glBindVertexArray(VAO);
    // 2. Copy our vertices array in a buffer for OpenGL to use
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);
    // 3. Then set our vertex attributes pointers
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (
    GLvoid*)0);
    glEnableVertexAttribArray(0);
//4. Unbind the VAO
glBindVertexArray(0);

[...]

// ..:: Drawing code (in Game loop) :: ..
// 5. Draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
glBindVertexArray(0);
```

重要的时刻到了！一个VAO，保存着我们的顶点属性配置及相应的VBO。通常，我们有多个物体需要绘制，我们可以先生成/配置所有的VAO(包括相应的VBO及属性指针)以备后用。当我们需要绘制某个物体时，我们只需要绑定相应的VAO，绘制这个物体，然后解绑定VAO即可。

4.2.7绘制三角形

我们用glDrawArrays函数还绘制物体，绘制时使用当前激活的着色器，前面定义的顶点属性配置及VBO的顶点数据（这些通过绑定的VAO完成）：

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

其中，glDrawArrays函数第一个参数告诉opengl我们要绘制什么样的图形，第二个参数指定了我们准备从顶点数组中的第几个位置开始绘制，这里是0.最后一个参数指定了我们要绘制多少顶点。这里是3（因为三角形只有3个顶点）。具体代码参考参考代码1.2。

4.2.8元素缓存对象（EBO）

最后我们要了解下元素缓存对象（element buffer objects（EBO））。假如我们要绘制个长方形，则由于长方形可以看成由两个三角形组成，因此我们可以绘制两个三角形。为此我们可以生成以下顶点：

```
GLfloat vertices[] = {
    // First triangle
     0.5f, 0.5f, 0.0f, // Top Right
     0.5f, -0.5f, 0.0f, // Bottom Right
    -0.5f, 0.5f, 0.0f, // Top Left
    // Second triangle
     0.5f, -0.5f, 0.0f, // Bottom Right
    -0.5f, -0.5f, 0.0f, // Bottom Left
    -0.5f, 0.5f, 0.0f // Top Left
};
```

可以看到，由长方形由4个顶点组成，而这里确需要指定6个顶点（有2个顶点都被指定了2次）。如果一个模型由1000多个三角形组成的话，这种重复指定的现象会更加严重。更好的解决方案是将顶点只保存一次，然后指定我们绘制这些顶点的顺序。EBO就是这样工作的。EBO同顶点缓存对象VBO类似，只是还保存着绘制顶点的下标：

```
GLfloat vertices[] = {
     0.5f, 0.5f, 0.0f, // Top Right
     0.5f, -0.5f, 0.0f, // Bottom Right
    -0.5f, -0.5f, 0.0f, // Bottom Left
    -0.5f, 0.5f, 0.0f // Top Left
};
GLuint indices[] = { // Note that we start from 0!
    0, 1, 3, // First Triangle
    1, 2, 3    // Second Triangle
};
```

下面，我们需要创建EBO:

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

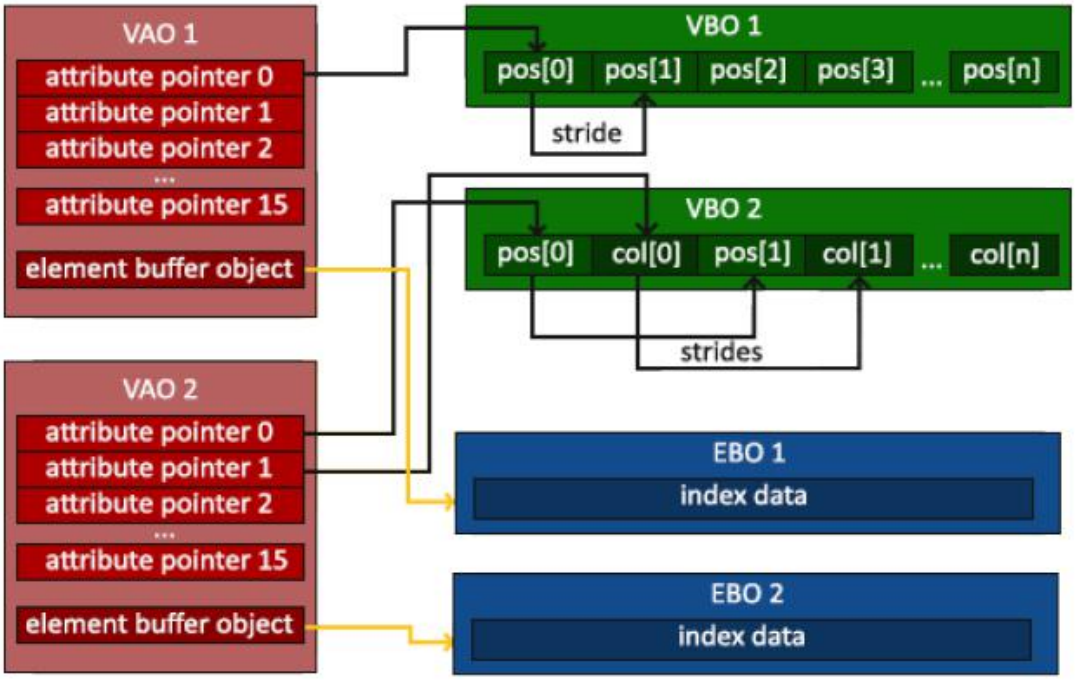
然后，我们要绑定EBO，并用glBufferData将顶点下标拷贝到缓存区：

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

绘制函数也要有glDrawArrays换为glDrawElements：

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

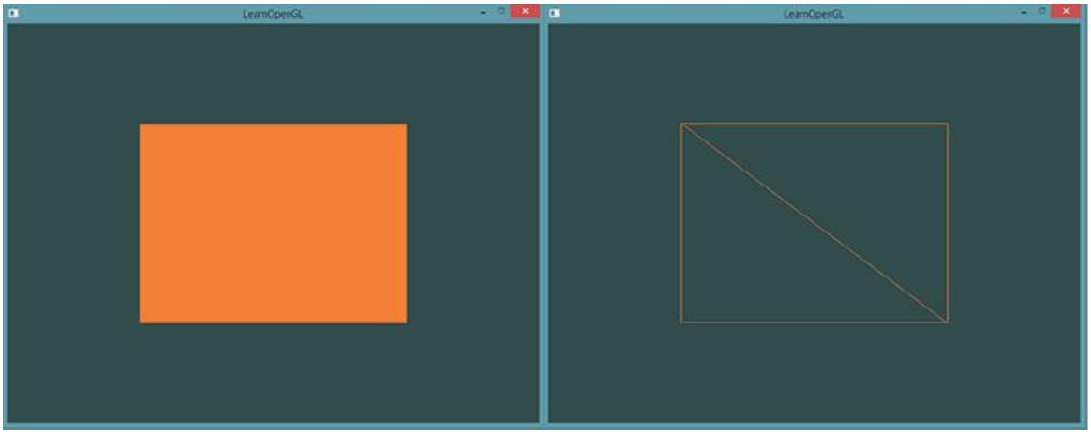
glDrawElements函数的第二个参数指明了我们要绘制6个顶点。最后一个参数指明了EBO中的偏移量。这个函数从EBO里得到顶点下标（即glBindBuffer函数绑定的到GL_ELEMENT_ARRAY_BUFFER的EBO）。也就是说，我们每次要绘制物体前，都要将相应的EBO绑定起来。这样就比较反锁了。解决方法：类似前面的VBO，当我们绑定VAO时，也会将保存在VAO中的EBO绑定起来。



最终，采用EBO的初始化命令绘制命令为：

```
// ..:: Initialization code :: ..
// 1. Bind Vertex Array Object
glBindVertexArray(VAO);
    // 2. Copy our vertices array in a vertex buffer for OpenGL to use
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);
    // 3. Copy our index array in a element buffer for OpenGL to use
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
    GL_STATIC_DRAW);
    // 3. Then set the vertex attributes pointers
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (
    GLvoid*)0);
    glEnableVertexAttribArray(0);
// 4. Unbind VAO (NOT the EBO)
glBindVertexArray(0);

[...]

// ..:: Drawing code (in Game loop) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
```

运行效果：



右边的图是用线框模式绘制的图形。

注：线框模式

> **Wireframe mode**
> To draw your triangles in wireframe mode, you can configure how OpenGL draws its primitives via `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. The first argument says we want to apply it to the front and back of all triangles and the second line tells us to draw them as lines. Any subsequent drawing calls will render the triangles in wireframe mode until we set it back to its default using `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)`.

三、实验内容

（一）分析以下程序的原理并上机验证

1．首先调试并运行以下参考程序，以建立一个窗口。然后在参考程序基础上，加入键盘事件处理过程，使得按空格键可以退出应用，并用指定的颜色（如红色）绘制窗口。

[参考程序 1.1]

```
#include <iostream>

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

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


// Window dimensions
```

```
const GLuint WIDTH = 800, HEIGHT = 600;

// The MAIN function, from here we start the application and run the game
loop
int main()
{
    std::cout << "Starting GLFW context, OpenGL 3.3" << std::endl;
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "LearnOpenGL",
nullptr, nullptr);
    if (window == nullptr)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    // Set this to true so GLEW knows to use a modern approach to
retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    if (glewInit() != GLEW_OK)
    {
        std::cout << "Failed to initialize GLEW" << std::endl;
        return -1;
    }

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);

    // Game loop
```

```
    while (!glfwWindowShouldClose(window))
    {
        // Check if any events have been activiated (key pressed, mouse
moved etc.) and call corresponding response functions
        glfwPollEvents();
        // Swap the screen buffers
        glfwSwapBuffers(window);
    }

    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}
```

2. 调试并运行以下程序，在屏幕上绘制一个三角形. 然后在此基础上，根据 EBO 概念，绘制由 2 个三角形组成的长方形。

[参考程序 1.2]

```
#include <iostream>

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

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


// Function prototypes
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode);

// Window dimensions
const GLuint WIDTH = 800, HEIGHT = 600;

// Shaders
const GLchar* vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 position;\n"
    "void main()\n"
```

```
    "{\n"
    "gl_Position = vec4(position.x, position.y, position.z, 1.0);\n"
    "}\0";
const GLchar* fragmentShaderSource = "#version 330 core\n"
    "out vec4 color;\n"
    "void main()\n"
    "{\n"
    "color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
    "}\n\0";

// The MAIN function, from here we start the application and run the game loop
int main()
{
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "LearnOpenGL", NULL, NULL);
    glfwMakeContextCurrent(window);

    // Set the required callback functions
    glfwSetKeyCallback(window, key_callback);

    // Set this to true so GLEW knows to use a modern approach to retrieving function pointers
and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);


    // Build and compile our shader program
```

```
// Vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
// Check for compile time errors
GLint success;
GLchar infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
// Fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
// Check for compile time errors
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
}
// Link shaders
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
// Check for linking errors
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

```
    // Set up vertex data (and buffer(s)) and attribute pointers
    GLfloat vertices[] = {
        -0.5f, -0.5f, 0.0f, // Left
         0.5f, -0.5f, 0.0f, // Right
         0.0f,  0.5f, 0.0f  // Top
    };
    GLuint VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    // Bind the Vertex Array Object first, then bind and set vertex buffer(s) and attribute pointer(s).
    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

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

    glBindBuffer(GL_ARRAY_BUFFER, 0); // Note that this is allowed, the call to glVertexAttribPointer registered VBO as the currently bound vertex buffer object so afterwards we can safely unbind

    glBindVertexArray(0); // Unbind VAO (it's always a good thing to unbind any buffer/array to prevent strange bugs)

    // Game loop
    while (!glfwWindowShouldClose(window))
    {
        // Check if any events have been activiated (key pressed, mouse moved etc.) and call corresponding response functions
        glfwPollEvents();

        // Render
        // Clear the colorbuffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Draw our first triangle
```

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

        // Swap the screen buffers
        glfwSwapBuffers(window);
    }
    // Properly de-allocate all resources once they've outlived their purpose
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}

// Is called whenever a key is pressed/released via GLFW
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

（二）按要求进行程序设计并调试运行

1．用两个不同 VAO 和 VBO，绘制两个相同的三角形。.

2．建立两个着色器程序，第二个程序使用不同的 fragment shader，它输出黄色。然后再绘制 2 个三角形，第二个三角形为黄色（利用着色器程序）。.

实验二　着色器及纹理映射

一、实验目的

1．了解和掌握着色器的基本概念、使用方法.

2．掌握纹理映射的基本概念及编程方法，并能灵活应用.

二、基本概念

1. 着色器及 GLSL 语言

在上一个实验中，我们已经知道着色器是一个在 GPU 中的小程序，用于完成图形流水线上个相应工作步骤。简单来说，着色器就是一个将输入变换为输出的程序，且各自独立，除了通过输入和输出外，不允许相互间通信。下面我们要对着色器及 Opengl 的着色器语言进行更深入的了解。

1.1 GLSL 语言

．GLSL 语言同 c 很类似，适用于处理图形，处理向量和矩阵很方便。着色器用 GLSL 语言编写，开始时是版本声明，然后是输入输出变量，uniform，main 函数。每个着色器的入口是它的 main 函数，在这个函数里，我们处理输入变量，将结果输出的输出变量。一般的结构如下：

```
#version version_number

in type in_variable_name;
in type in_variable_name;

out type out_variable_name;
```

```
uniform type uniform_name;

int main()
{
// Process input(s) and do some weird graphics stuff
...
// Output processed stuff to output variable
out_variable_name = weird_stuff_we_processed;
}
```

当我们说到顶点着色器时，也称输入变量为顶点属性。因为硬件限制，我们允许声明的顶点属性数量是有限的。Opengl 保证至少可声明 16 个 4 分量的顶点属性。可通过以下方法得到可声明的顶点属性最大数量：

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

1.2 着色器的输入和输出

着色器虽然是个很小的程序，不过他们也是整个流水线上的一部分，每个着色器都有自己的输入和输出变量。GLSL 用 in 和 out 关键字来说明。每当一个着色器的输出变量同后面阶段着色器的输入变量匹配时，这两个变量就联系在一起了。不过顶点着色器和面片着色器（fragment shader）稍有点不同。

顶点着色器直接从顶点数据中接受输入数据。为了定义顶点数据的组织形式，我们指定了具有位置信息的输入变量。在前面已经看到了这样的例子：layout（location = 0）。也就是说，顶点着色器需要输入数据额外的布局（layout）规定，以便我们可以将其同顶点数据联系起来。

而面片着色器的不同之处在于它需要一个 vec4 类型的颜色输出变量。这是因为面片着色器需要产生最终的输出颜色。如果我们在面片着色器里忘记指定输出颜色，则 opengl 会将我们的物体绘制为黑色（或白色）。

因此，如果我们想将数据从一个着色器传送到另外一个着色器，我们需要将发送方着色器的输出数据类型和名字同接受方着色器输入数据类型和名字设置为相同。这样，opengl 就会将这两个变量联系起来。下面是一个例子，展示了顶点着色器如何用这种方法确定面片着色器的输出颜色：

顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 position; // The position variable has
    attribute position 0

out vec4 vertexColor; // Specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(position, 1.0); // See how we directly give a vec3
    to vec4's constructor
    vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f); // Set the output variable
    to a dark-red color
}
```

面片着色器：

```
#version 330 core
in vec4 vertexColor; // The input variable from the vertex shader (same
    name and same type)

out vec4 color;

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

1.3 Uniforms 变量

在应用程序同着色器之间传递数据的方式是通过 uniforms 变量。uniforms 同顶点属性有所不同。首先 uniforms 是全局变量，也就是说，uniforms 变量是唯一的，任何着色器在流水线中的任意阶段都可访问它。其次，我们设置好了 uniform 值以后，uniform 将一直保持这个值，直到被更新或重置。

下面我们就用 uniform 来设置三角形的颜色：

面片着色器：

```
#version 330 core

out vec4 color;

uniform vec4 ourColor; // We set this variable in the OpenGL code.

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

这里，我们声明了一个 uniform 变量 ourColor，并将输出颜色设为 ourColor。由于 uniforms

是全局变量，因此我们可以在任意一个着色器中定义它，而不是非要在顶点着色器中定义。现在对这个 uniform 变量赋值。这次我们将颜色值设为随时间逐渐改变：

```
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);
```

以上程序中，我们首先得到 uniform 属性在我们着色器中的索引/位置，然后就可以更新它的值了。注意，我们更新一个 uniform 变量时，要首先用 glUseProgram 来使用着色器程序，因为设置的 uniform 值是用于当前被激活的着色器程序的。

因为我们想让三角形的颜色是渐变的，因此我们每个事件循环中都对这个 uniform 变量进行更新：

```
while(!glfwWindowShouldClose(window))
{
    // Check and call events
    glfwPollEvents();

    // Render
    // Clear the colorbuffer
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Be sure to activate the shader
    glUseProgram(shaderProgram);

    // Update the uniform color
    GLfloat timeValue = glfwGetTime();
    GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
    GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "
    ourColor");
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);

    // Now draw the triangle
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);
}
```

现在，我们想将三角形的每个顶点颜色设为不同的。如果还是用 uniforms 的话，我们就需要同顶点个数一样多的 uniforms 变量。更好的方案是在顶点属性中设置更多的数据。

1.4 更多的属性

在以前的实验中，我们已经知道如何设置一个 VBO，配置顶点属性指针，并将他们都保存在 VAO 中。现在，我们还想在顶点数据中增加颜色数据：

```
GLfloat vertices[] = {
    // Positions         // Colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // Bottom Right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // Bottom Left
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // Top
};
```

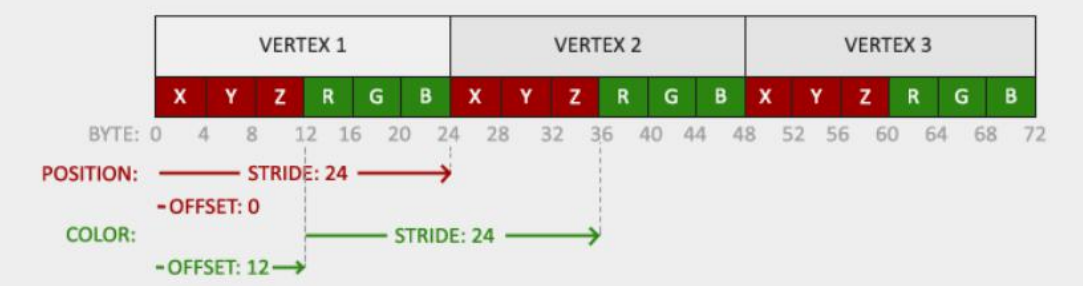
由于我们要向顶点着色器发送更多的数据，因此需要调整顶点着色器代码：

```
#version 330 core
layout (location = 0) in vec3 position; // The position variable has attribute position 0
layout (location = 1) in vec3 color;    // The color variable has attribute position 1

out vec3 ourColor; // Output a color to the fragment shader

void main()
{
    gl_Position = vec4(position, 1.0);
    ourColor = color; // Set ourColor to the input color we got from the vertex data
}
```

注意，我们将坐标的位置设为 0，将 color 的位置设为 1。由于我们用 ourColor，而不再用 uniform 变量为面片着色器设置颜色，因此面片着色器也做相应更改：

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

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

因为我们添加了一个顶点属性，因此我们要重新配置顶点属性指针。现在 VBO 的内存布局如下：

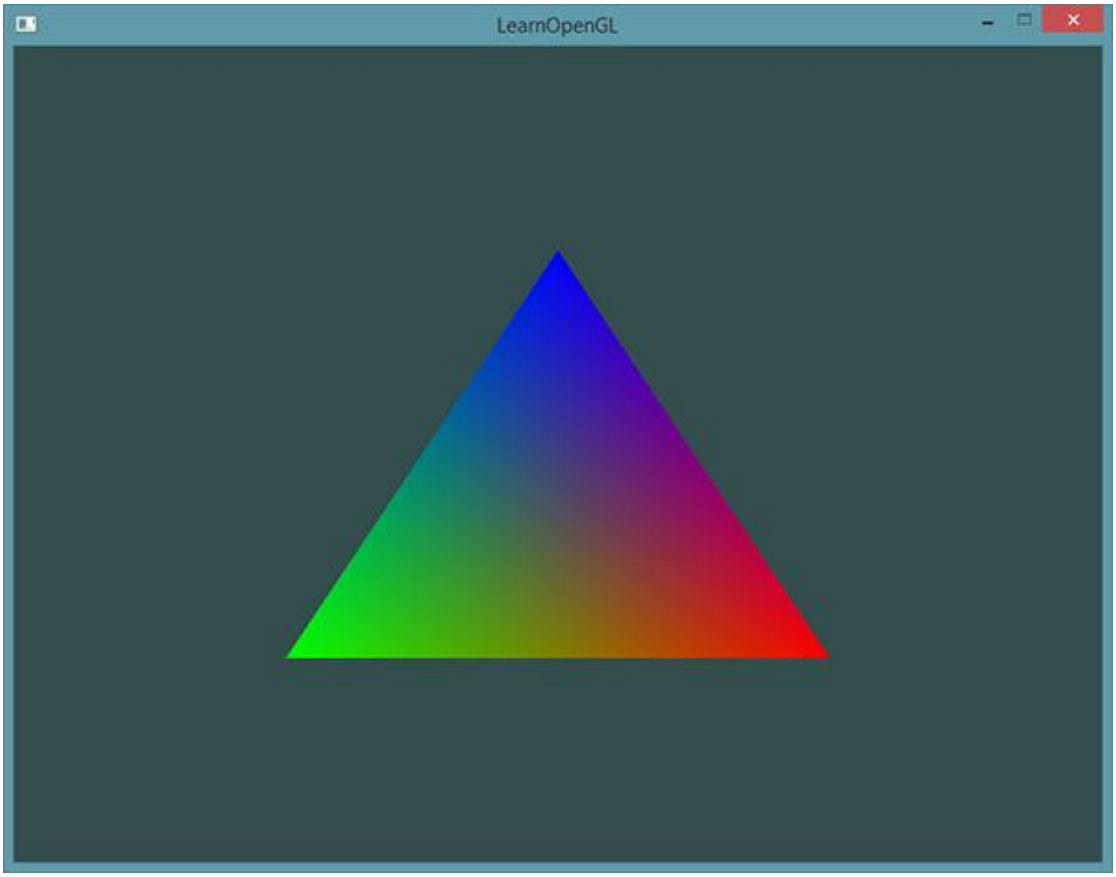


根据此时的 VBO 内存布局，我们对 glVertexAttribPointer 的调用更改为：

```
// Position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
// Color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3* sizeof(GLfloat
glEnableVertexAttribArray(1);
```

在配置 color 的属性时，我们将 glVertexAttribPointer 函数的第一个参数，也就是属性位置

参数设为 1（同 layout 中设置的对应）。由于移动到下一个数据需要越过 6 个数，因此上面 2 个 glVertexAttribPointer 函数的第五个参数都是 6* sizeof(GLfloat)。对于偏移量（offset）的设置，顶点位置属性在最开始，因此第一个 glVertexAttribPointer 函数的最后一个参数为 0，而第二个 glVertexAttribPointer 函数的最后一个参数为 3 * sizeof(GLfloat)。运行参考程序 2.1，可得如下输出结果：



输出的图形也许同我们所期望的有所不同。我们只提供了三个顶点的颜色，而输出的是个彩色图形。这是由于面片着色器的面片差值的结果。当绘制三角形时，在 rasterization 阶段，会自动产生很多小三角形，形成很多小面片。然后会确定大三角形中这很多小面片的坐标。基于这些坐标，它会内插面片着色器的输入数据。比如，我们有段直线，一个端点是绿色，另一个是蓝色。如果面片着色器运行到一个小面片，位置在线段的 70%处，则其输入的颜色属性就是绿色和蓝色的插值，即 30%的蓝色，70%的绿色。我们虽然绘制的三角形只有 3 个顶点，3 种颜色，不过这个三角形内部却可能包含着 50000 多个小面片，面片着色器则会对这许多顶点的颜色进行插值，最终形成我们看到的结果。

1.5 着色器类

可以看到，编写着色器，编译并管理着色器非常繁琐。为了简明期间，我们可以编写一个着色器类，以便从文件中读取着色器，编译并连接它们，检查错误等。这样用起来就会更方便，也可以练习一下如果将有用的知识用一个抽象的类封装起来。以下是声明类的头文件：

```
#ifndef SHADER_H
#define SHADER_H

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
using namespace std;

#include <GL/glew.h>; // Include glew to get all the required OpenGL
    headers

class Shader
{
public:
    // The program ID
    GLuint Program;
    // Constructor reads and builds the shader
    Shader(const GLchar* vertexSourcePath, const GLchar* fragmentSourcePath
    );
    // Use the program
    void Use();
};

#endif
```

这个着色器类有个 GLuint 类型的成员 Program，用来保存着色器程序的 ID 号。类的构造甘薯需要顶点和面片着色器源码文件的路径。从文件中读数据需要几个 string 对象：

```
Shader(const GLchar* vertexPath, const GLchar* fragmentPath)
{
    // 1. Retrieve the vertex/fragment source code from filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // ensures ifstream objects can throw exceptions:
    vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit)
    ;
```

```
    fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit)
    ;
    try
    {
        // Open files
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // Read file's buffer contents into streams
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // close file handlers
        vShaderFile.close();
        fShaderFile.close();
        // Convert stream into GLchar array
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch(std::ifstream::failure e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::
    endl;
    }
    const GLchar* vShaderCode = vertexCode.c_str();
    const GLchar* fShaderCode = fragmentCode.c_str();
    [...]
```

下面我们要对着色器进行编译和链接。

```
// 2. Compile shaders
GLuint vertex, fragment;
GLint success;
GLchar infoLog[512];

// Vertex Shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
// Print compile errors if any
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog
    << std::endl;
};

// Similiar for Fragment Shader
[...]

// Shader Program
this->Program = glCreateProgram();
glAttachShader(this->Program, vertex);
glAttachShader(this->Program, fragment);
glLinkProgram(this->Program);
// Print linking errors if any
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;
}

// Delete the shaders as they're linked into our program now and no longer
    necessery
glDeleteShader(vertex);
glDeleteShader(fragment);
```

下面是 use 函数的实现：

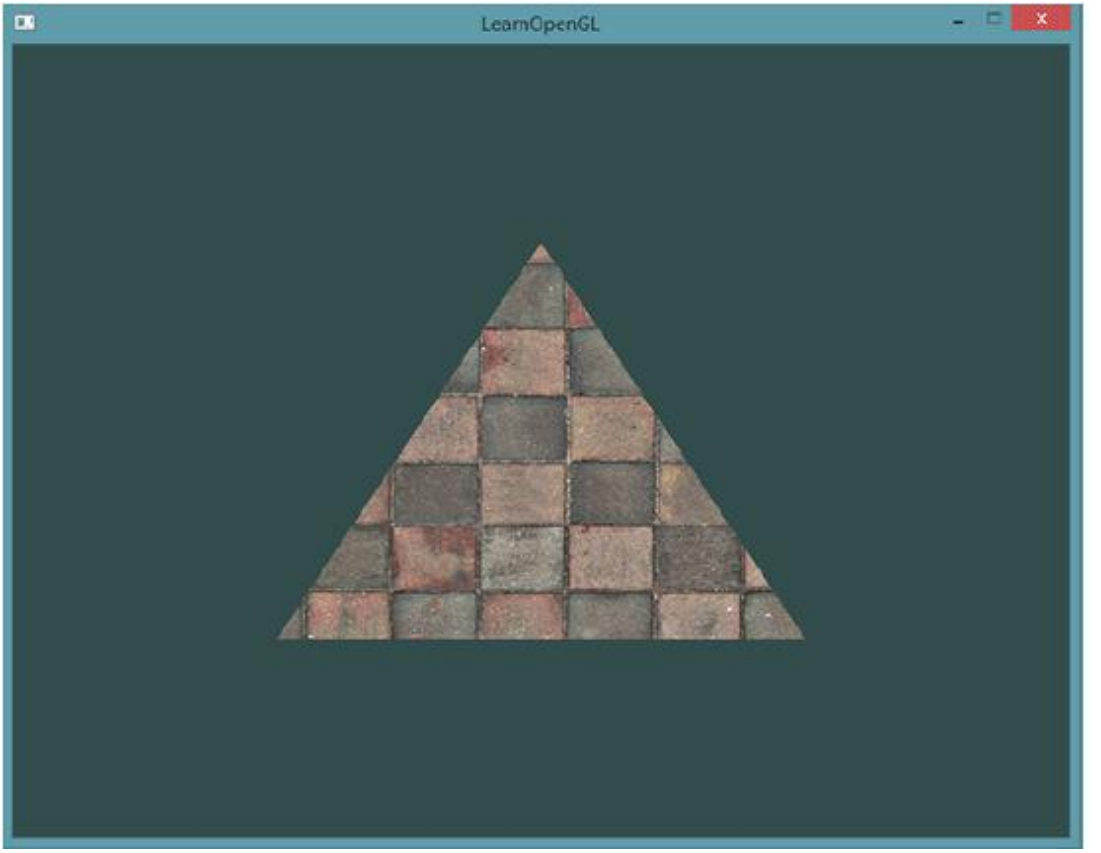
```
void Use() { glUseProgram(this->Program); }
```

有了着色器类以后，使用起来就方便多了。我们只需要创建一个着色器对象，然后使用它就可以了：

```
Shader ourShader("path/to/shaders/shader.vs", "path/to/shaders/shader.frag"
    );
...
while(...)
{
    ourShader.Use();
    glUniform1f(glGetUniformLocation(ourShader.Program, "someUniform"), 1.0
    f);
    DrawStuff();
}
```

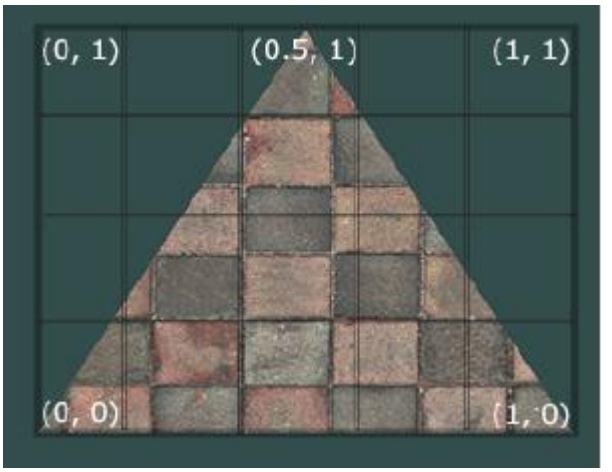
2. 纹理

想绘制具有更多细节的物体，我们可以给每个顶点赋予不同的颜色。Range，如果想让物体很逼真的话，我们需要非常多的顶点，从而要指定很多颜色值。这个工作量就太大了。实际上，艺术家和程序员更喜欢使用纹理。纹理是一个 2D 图像，以便赋予一个物体更多的细节。比如，可将纹理想象成是一个有砖头样的纸张，用它将我们的 3D 房屋模型包裹起来，这样看起来就像我们的房屋是用砖头砌成的一样。通过这种方式，我们就可以在不指定许多顶点的情况下，赋予物体更多的细节。
下图就是添加了砖头纹理的三角形：



为了将纹理映射到我们的三角形上面，我们需要知道三角形的每个顶点对应着纹理图像的哪个部分。也就是说，每个顶点都有个纹理坐标，以指明它的颜色从纹理图像的哪个部位取得。然后面片插值功能会处理其余的面片。

纹理坐标的 x，y 范围均是 0-1。利用纹理坐标取得纹理颜色成为采样。纹理坐标从左下角的（0,0）开始，到右上角的（1,1）。如下图：



我们三角形的三个顶点分别被指定了 3 个纹理坐标，分别是（0,0），（1,0）和（0.5,1）。

将这三个纹理坐标传给顶点着色器即可，顶点着色器会将其传送给面片着色器，面片着色器会对每个小面片的纹理坐标进行插值处理。纹理坐标如下：

```
GLfloat texCoords[] = {

    0.0f, 0.0f, // Lower-left corner
    1.0f, 0.0f, // Lower-right corner
    0.5f, 1.0f // Top-center corner
};
```

下面我们要告诉 opengl 如何对纹理进行采样。

2.1 纹理包装

纹理坐标的范围一般是从（0，0）到（1,1）。但如果我们指定的纹理坐标超出这个范围了会怎样呢？Opengl 的默认行为是重复这个纹理图像。Opengl 提供了以下行为可供选择：

- GL_REPEAR:默认行为，重复纹理图像。
- GL_MIRRORED_REPEAT:同 GL_REPEAT 相同，不过图像是镜像重复的。
- GL_CLAMP_TO_EDGE:将坐标限制到 0 和 1 之间。结果是坐标值大的就被拉到边缘了。
- GL_CLAMP_TO_BORDER：超出边界的坐标位置被赋予用户指定的边界色。

下图为各种方式的效果：

可以用 glTexParameter*函数来 设置各个坐标的选项（s，t（3D 纹理还可以有 r）同 x，y，z 对应）：

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

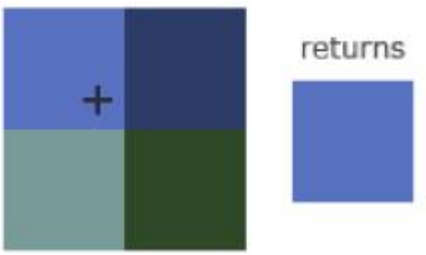
如果我们选择了 GL_CLAMP_TO_BORDER 选项，则我们还需要指定边界颜色：

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

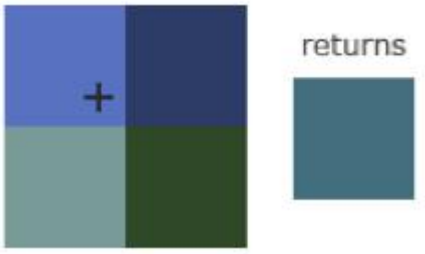
2.2 纹理滤波

Opengl 必须指出纹理像素映射 到哪个纹理坐标上。我们现在讨论两个最重要的选项：GL_NEAREST 和 GL_LINEAR.

GL_NEAREST(也称为最近邻滤波）是 opengl 默认的纹缕滤波方法。Opengl 会将像素点映射到距其最近的纹理坐标。下图有四个像素，十字星表示纹理坐标位置。此时由于左上的像素中心店距离纹理坐标最近，因此这个像素被选中，作为采样色：



GL_LINEAR（也称为双线性滤波）对纹理坐标的相邻像素点进行线性组合，作为此像素点出的颜色。当纹理坐标距离相邻的像素中心越近时，则这个像素就会向采样颜色贡献出更多的颜色值：



为了看看这两种方式不同的效果，现在我们将一个低分辨率的纹理图像用于一个大物体上（纹理图被放大了，这样可看到单个像素）：

GL_NEAREST

GL_LINEAR

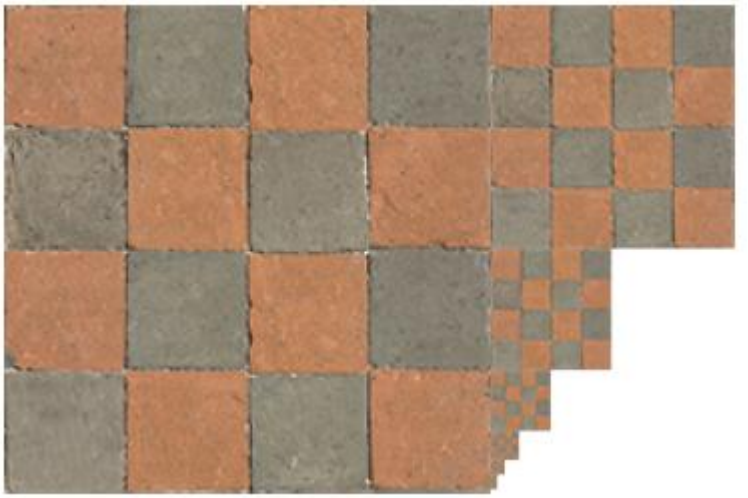
纹理滤波可以设成扩大或缩小操作。当纹理缩小时，我们可以用最近邻滤波，当纹理扩大时，我们可用线性滤波。

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

2.3 分级细化纹理

假设我们有一个大房间，里面有许多物体，每个物体都有纹理。如果不论距离我们观察者远近，物体的纹理分辨率都一样高，则由于远处物体看起来很小，opengl 就很难从高分辨率的纹理中生成这个面片的颜色值。因为此时这个小面片的一个像素点覆盖了纹理的很大一部分。而且对看起来很小的物体用高分辨率纹理也浪费内存等资源。

为解决这个问题，opengl 采用了分级细化纹理的概念。就是说有一系列的纹理图像，每个图像的分辨率是上一级的一半。当观察者同物体超过一定距离时，就使用不同级别的纹理图。分级细化纹理图如下：



手工创建这一系列的分级细化纹理图比较繁琐。Opengl 可以用函数 glGenerateMipmaps 帮我们完成这个工作。

同纹理滤波一样，在不同级的分级细分纹理之间，也可以用 NEAREST 和 LINEAR 滤波，来取得更平滑的现象。

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

2.4 创建及装载纹理

首先，我们要装载纹理。纹理图一般以文件的格式保存，可能有许多文件格式。怎样得到这些纹理图像呢？一种方案是选择一种文件格式，如.PNG 格式，然后自己写个加载程序，以便 将图像转换成我们所要的数组的形式。虽然这种方法不是很麻烦，不过如果我们要支持的文件格式很多时，也很繁琐。另一种方案就是采用一个可支持很多图像格式的图像装载库，如 SOIL 库。

2.5 图像装载库 SOIL

可以从网上下载到 SOIL 库源码，然后编译即可生成相应的 SOIL.lib 文件，同以前所述类似，将头文件和库文件目录等参数配置好，即可使用了（可参考实验一的“第三方库的配置和使用”部分）。

用以下代码将纹理文件装入内存：

```
int width, height;
unsigned char* image = SOIL_load_image("container.jpg", &width, &height, 0,
    SOIL_LOAD_RGB);
```

这个函数首先将纹理图像的路径作为输入参数，然后第二第三个参数用来获取图像的宽和高。第四个参数是图像的通道（channels）个数。这里为 0。最后一个参数告诉 SOIL 怎样装载图像。这里我们只对图像的 RGB 值感兴趣。

2.6 创建纹理

同 opengl 中其他对象类似，纹理对象也有个 ID 号：

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

第一个参数是我们需要生成纹理的个数。然后，我们要绑定纹理：

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

现在可以生成纹理了：

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

第一个函数参数较多，第二个参数可以指定分级细化纹理的细化等级，这里我们指定了 0，及最低级。第三个参数告诉我们纹理图像只保存了 RGB 值。第 6 个参数一直为 0。第七第八个参数设置了纹理图像的格式和数据类型。这里是 RGB 格式，将它们保存为字节类型。最后一个参数是实际的图像数据。

执行 glTexImage2D 函数后，当前绑定的纹理对象就有了纹理图像了。不过现在只有最低级的纹理图像。执行 glGenerateMipmap 函数则自动生成所有其他的分级纹理图像。

生成了纹理图像及相关的分级纹理图像后，现在可以释放图像内存，解绑定纹理对象：

```
SOIL_free_image_data(image);
glBindTexture(GL_TEXTURE_2D, 0);
```

创建纹理的整个过程如下：

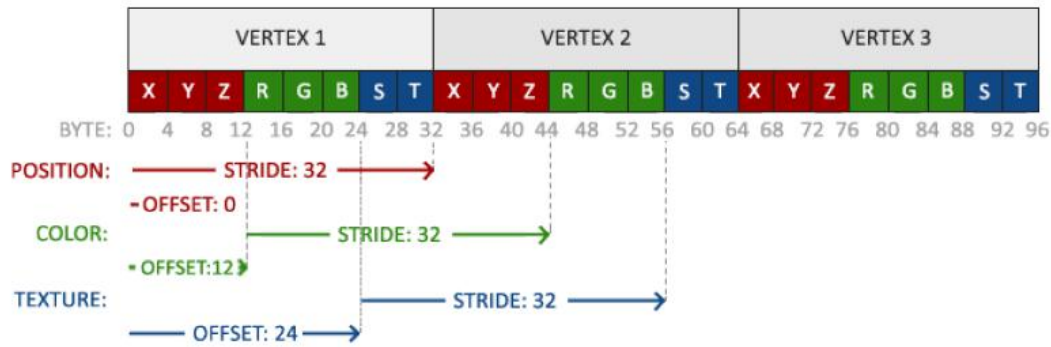
```
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// Set the texture wrapping/filtering options (on the currently bound
    texture object)
...
// Load and generate the texture
int width, height;
unsigned char* image = SOIL_load_image("container.jpg", &width, &eight, 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);
```

2.7 应用纹理

我们在绘制的长方形中应用纹理。为此，我们首先建立含有纹理坐标的顶点数据：

```
GLfloat vertices[] = {
    // Positions          // Colors           // Texture Coords
     0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // Top Right
     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Bottom Right
    -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // Bottom Left
    -0.5f,  0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f    // Top Left
};
```

由于我们添加了纹理属性，顶点现在的内存分布变为：



为此，相应函数的参数也需要修改：

```
glVertexAttribPointer(2, 2, GL_FLOAT,GL_FALSE, 8 * sizeof(GLfloat), (GLvoid
    *)(6 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
```

注意，这里函数的跨度参数改为了 8*sizeof(GLfloat)。

下面，我们需要调整顶点着色器，以接受输入的顶点纹理数据并输出相应的纹理坐标到面片着色器中:

```
#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;
}
```

面片着色器会将纹理坐标 TexCoord 作为输入变量，而且面片着色器还需要能访问纹理对象。可怎样才能将纹理对象传送给面片着色器呢？GLSL 有内置的数据类型 sampler1D，sampler2D 等，我们可以通过声明一个 uniform sampler2D 类型的变量，将纹理对象传送给面片着色器：

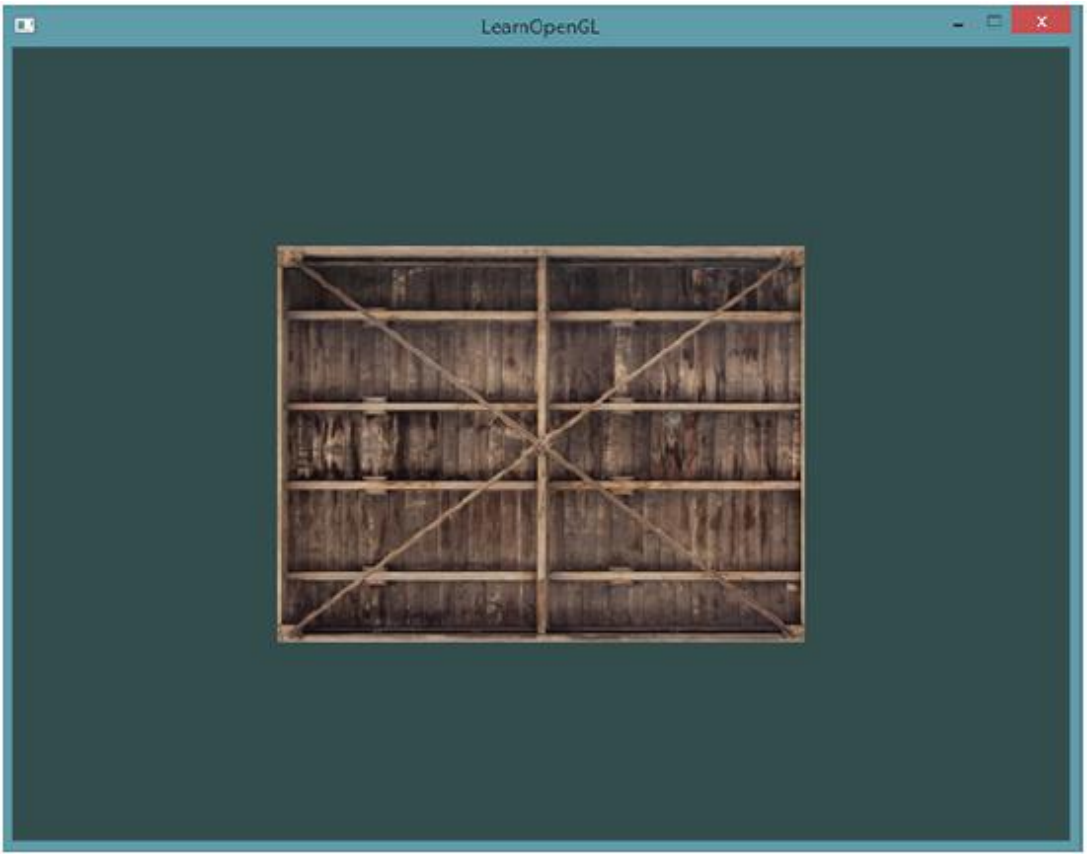
```
#version 330 core
in vec3 ourColor;
in vec2 TexCoord;

out vec4 color;

uniform sampler2D ourTexture;

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

以上着色器的代码中，为了能对纹理颜色采样，我们使用了 GLSL 的内置函数 texture，其第一个参数是纹理对象，第二个参数是相应的纹理坐标。函数输出则是在纹理坐标（或插值的坐标）上的纹理颜色。

我们在绘制物体之前，将纹理绑定起来即可：

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

绘制的图形如下：



如果将纹理颜色同顶点颜色混合起来，则可以在面片着色器中将这两个颜色乘起来即可：

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

2.8 纹理单元

我们可以利用纹理单元来使用多个纹理。通过给采样器指定多个纹理单元，我们可以同时绑定多个纹理。

```
glActiveTexture(GL_TEXTURE0);     // Activate the texture unit first before
    binding texture
glBindTexture(GL_TEXTURE_2D, texture);
```

我们首先要记过纹理单元，然后进行绑定。
我们还需要对面片着色器进行相应的更改。

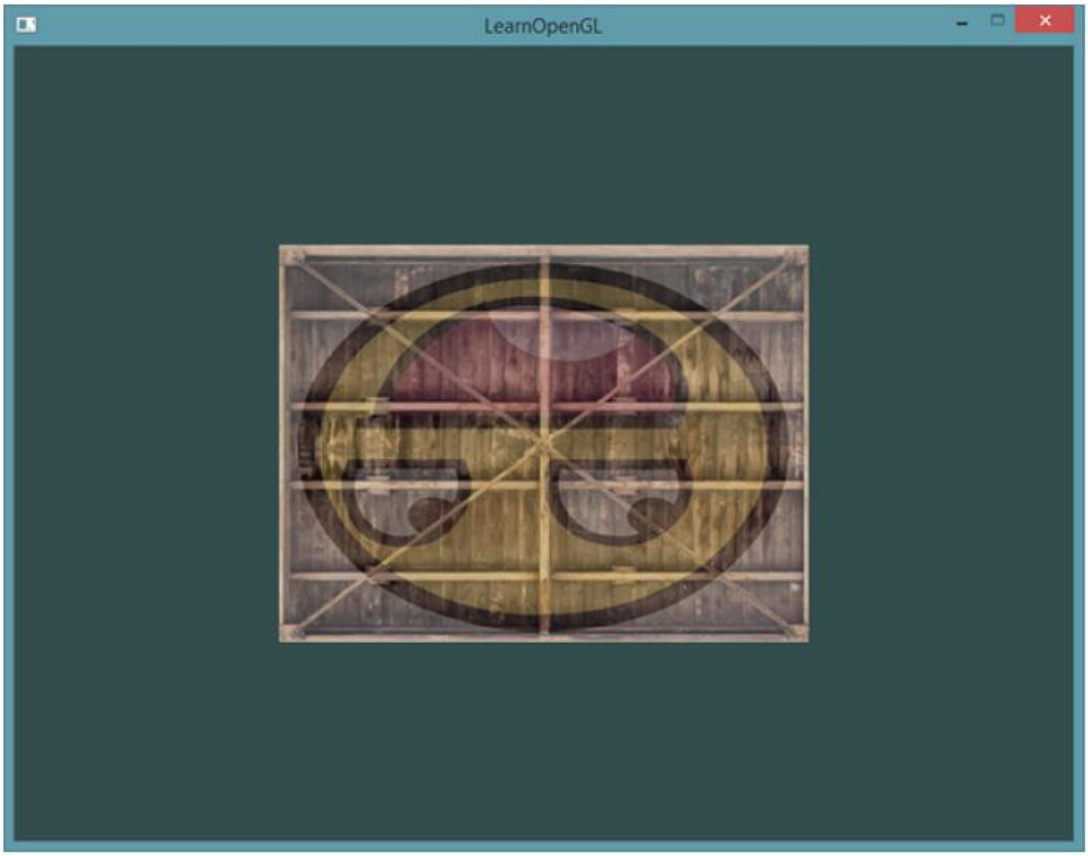
```
#version 330 core
...

uniform sampler2D ourTexture1;
uniform sampler2D ourTexture2;

void main()
{
    color = mix(texture(ourTexture1, TexCoord), texture(ourTexture2,
    TexCoord), 0.2);
}
```

这里，用了 GLSL 的内置函数 mix，将两个纹理颜色混合（混合比例由第三个参数确定，这里是 0.2），得到输出颜色。

然后利用以前的知识，载入并创建另一个纹理。为了同时使用 2 个纹理，我们需要改变相应的代码：

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glUniform1i(glGetUniformLocation(ourShader.Program, "ourTexture1"), 0);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);
glUniform1i(glGetUniformLocation(ourShader.Program, "ourTexture2"), 1);

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

两个纹理混合后的绘制效果如下：



三、实验内容

（一）分析以下程序的原理并上机验证

1.首先调试并运行参考程序 2.1，绘制一个三个顶点为不同颜色的三角形。然后在参考程序基础上，将着色器的编译连接等做成一个类，读取着色器源码也从文件中读入（具体做法可参考前面着色器的概念讲述部分 1.5）。

[参考程序 2.1]

```
. #include <iostream>
// GLEW#define GLEW_STATIC#include <GL/glew.h>
// GLFW#include <GLFW/glfw3.h>
```

```
// Function prototypesvoid key_callback(GLFWwindow* window, int key,
int scancode, int action, int mode);
// Window dimensionsconst GLuint WIDTH = 800, HEIGHT = 600;
// Shadersconst GLchar* vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 position;\n"
    "layout (location = 1) in vec3 color;\n"
    "out vec3 ourColor;\n"
    "void main()\n"
    "{\n"
    "gl_Position = vec4(position, 1.0);\n"
    "ourColor = color;\n"
    "}\0";const GLchar* fragmentShaderSource = "#version 330 core\n"
    "in vec3 ourColor;\n"
    "out vec4 color;\n"
    "void main()\n"
    "{\n"
    "color = vec4(ourColor, 1.0f);\n"
    "}\n\0";
// The MAIN function, from here we start the application and run the game
loopint main()
{
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "LearnOpenGL",
nullptr, nullptr);
    glfwMakeContextCurrent(window);

    // Set the required callback functions
    glfwSetKeyCallback(window, key_callback);
```

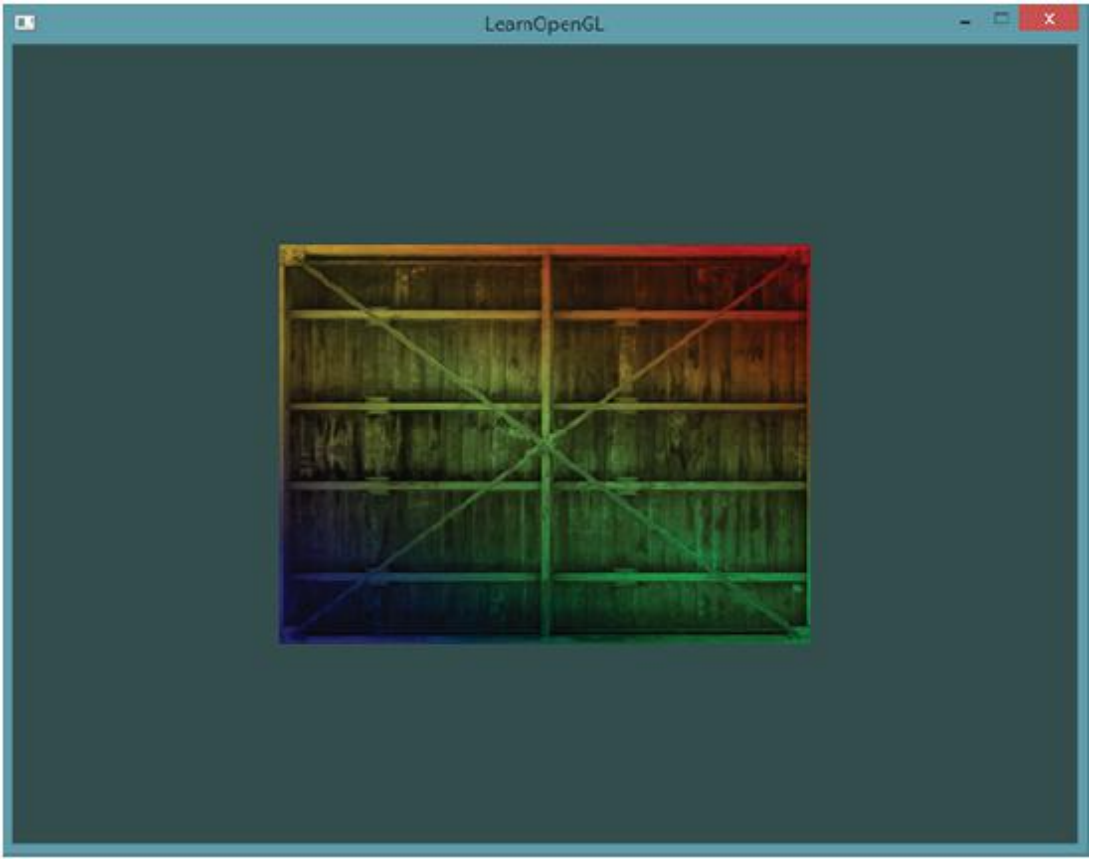
```
    // Set this to true so GLEW knows to use a modern approach to
retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);


    // Build and compile our shader program
    // Vertex shader
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);
    // Check for compile time errors
    GLint success;
    GLchar infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
infoLog << std::endl;
    }
    // Fragment shader
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);
    // Check for compile time errors
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n"
<< infoLog << std::endl;
    }
    // Link shaders
    GLuint shaderProgram = glCreateProgram();
```

```
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    // Check for linking errors
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<
infoLog << std::endl;
    }
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);



    // Set up vertex data (and buffer(s)) and attribute pointers
    GLfloat vertices[] = {
        // Positions         // Colors
         0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,  // Bottom Right
        -0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,  // Bottom Left
         0.0f,  0.5f, 0.0f,   0.0f, 0.0f, 1.0f   // Top
    };
    GLuint VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    // Bind the Vertex Array Object first, then bind and set vertex
buffer(s) and attribute pointer(s).
    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

    // Position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat),
(GLvoid*)0);
    glEnableVertexAttribArray(0);
    // Color attribute
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat),
(GLvoid*)(3 * sizeof(GLfloat)));
```

```
    glEnableVertexAttribArray(1);

    glBindVertexArray(0); // Unbind VAO


    // Game loop
    while (!glfwWindowShouldClose(window))
    {
        // Check if any events have been activiated (key pressed, mouse
moved etc.) and call corresponding response functions
        glfwPollEvents();

        // Render
        // Clear the colorbuffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Draw the triangle
        glUseProgram(shaderProgram);
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 3);
        glBindVertexArray(0);

        // Swap the screen buffers
        glfwSwapBuffers(window);
    }
    // Properly de-allocate all resources once they've outlived their
purpose
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}
// Is called whenever a key is pressed/released via GLFWvoid
key_callback(GLFWwindow* window, int key, int scancode, int action, int
mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
```

```
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

2.首先调试并运行参考程序 2.2，绘制一个有纹理的四边形。然后在参考程序基础上，修改面片着色器代码，将最终颜色变为纹理和坐标颜色的混合色，做出以下效果来：



[参考程序 2.2]

////////Main.cpp:

```
#include <iostream>
```

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

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

// Other Libs
#include <SOIL.h>

// Other includes
#include "Shader.h"



// Function prototypes
void key_callback(GLFWwindow* window, int key, int scancode, int
action, int mode);

// Window dimensions
const GLuint WIDTH = 800, HEIGHT = 600;
```

```
// The MAIN function, from here we start the application and run the
game loop
int main()
{
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's
functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT,
"LearnOpenGL", NULL, NULL);
    glfwMakeContextCurrent(window);
```

```
    // Set the required callback functions
    glfwSetKeyCallback(window, key_callback);

    // Set this to true so GLEW knows to use a modern approach to retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);


    // Build and compile our shader program
    Shader ourShader("textures.vs", "textures.frag");


    // Set up vertex data (and buffer(s)) and attribute pointers
    GLfloat vertices[] = {
        // Positions          // Colors           // Texture
```

```
Coords
            0.5f,  0.5f, 0.0f,      1.0f, 0.0f, 0.0f,      1.0f, 1.0f, // Top
Right
            0.5f, -0.5f, 0.0f,      0.0f, 1.0f, 0.0f,      1.0f, 0.0f, // Bottom
Right
           -0.5f, -0.5f, 0.0f,      0.0f, 0.0f, 1.0f,      0.0f, 0.0f, // Bottom
Left
           -0.5f,  0.5f, 0.0f,      1.0f, 1.0f, 0.0f,      0.0f, 1.0f    // Top Left
    };
    GLuint indices[] = {    // Note that we start from 0!
        0, 1, 3, // First Triangle
        1, 2, 3    // Second Triangle
    };
    GLuint VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);


    glBindVertexArray(VAO);
```

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

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

        // Position attribute
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(0);
        // Color attribute
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(1);
        // TexCoord attribute
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 *
sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
        glEnableVertexAttribArray(2);
```

```
glBindVertexArray(0); // Unbind VAO



// Load and create a texture
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture); // All upcoming
GL_TEXTURE_2D operations now have effect on this texture object
// Set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);     // Set texture wrapping to GL_REPEAT (usually
basic wrapping method)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
// Set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

```
    // Load image, create texture and generate mipmaps
    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); // Unbind texture when
done, so we won't accidentily mess up our texture.


    // Game loop
    while (!glfwWindowShouldClose(window))
    {
        // Check if any events have been activiated (key pressed,
mouse moved etc.) and call corresponding response functions
        glfwPollEvents();

        // Render
```

```
        // Clear the colorbuffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);


        // Bind Texture
        glBindTexture(GL_TEXTURE_2D, texture);

        // Activate shader
        ourShader.Use();

        // Draw container
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
0);
        glBindVertexArray(0);

        // Swap the screen buffers
        glfwSwapBuffers(window);
    }
```

```
    // Properly de-allocate all resources once they've outlived their
purpose
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}

// Is called whenever a key is pressed/released via GLFW
void key_callback(GLFWwindow* window, int key, int scancode, int
action, int mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}

//////////shader.h:
```

```
#ifndef SHADER_H
#define SHADER_H

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

#include <GL/glew.h>

class Shader
{
public:
    GLuint Program;
    // Constructor generates the shader on the fly
    Shader(const GLchar* vertexPath, const GLchar* fragmentPath)
    {
        // 1. Retrieve the vertex/fragment source code from filePath
        std::string vertexCode;
```

```
std::string fragmentCode;
std::ifstream vShaderFile;
std::ifstream fShaderFile;
// ensures ifstream objects can throw exceptions:
vShaderFile.exceptions (std::ifstream::badbit);
fShaderFile.exceptions (std::ifstream::badbit);
try
{
    // Open files
    vShaderFile.open(vertexPath);
    fShaderFile.open(fragmentPath);
    std::stringstream vShaderStream, fShaderStream;
    // Read file's buffer contents into streams
    vShaderStream << vShaderFile.rdbuf();
    fShaderStream << fShaderFile.rdbuf();
    // close file handlers
    vShaderFile.close();
    fShaderFile.close();
    // Convert stream into string
    vertexCode = vShaderStream.str();
```

```
            fragmentCode = fShaderStream.str();
        }
        catch (std::ifstream::failure e)
        {
            std::cout <<
"ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" <<
std::endl;
        }
        const GLchar* vShaderCode = vertexCode.c_str();
        const GLchar * fShaderCode = fragmentCode.c_str();
        // 2. Compile shaders
        GLuint vertex, fragment;
        GLint success;
        GLchar infoLog[512];
        // Vertex Shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
        glCompileShader(vertex);
        // Print compile errors if any
        glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
```

```
        if (!success)
        {
            glGetShaderInfoLog(vertex, 512, NULL, infoLog);
            std::cout <<
"ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
infoLog << std::endl;
        }
        // Fragment Shader
        fragment = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragment, 1, &fShaderCode, NULL);
        glCompileShader(fragment);
        // Print compile errors if any
        glGetShaderiv(fragment, GL_COMPILE_STATUS,
&success);
        if (!success)
        {
            glGetShaderInfoLog(fragment, 512, NULL, infoLog);
            std::cout <<
"ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" <<
infoLog << std::endl;
```

```
        }
        // Shader Program
        this->Program = glCreateProgram();
        glAttachShader(this->Program, vertex);
        glAttachShader(this->Program, fragment);
        glLinkProgram(this->Program);
        // Print linking errors if any
        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;
        }
        // Delete the shaders as they're linked into our program now
and no longer necessery
        glDeleteShader(vertex);
```

```
        glDeleteShader(fragment);


    }
    // Uses the current shader
    void Use()
    {
        glUseProgram(this->Program);
    }
};


#endif


//////////textures.vs
#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;
}
/////////textures.frag
#version 330 core
in vec3 ourColor;
in vec2 TexCoord;

out vec4 color;

uniform sampler2D ourTexture;

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

（二）按要求进行算法设计并调试运行

1. 请调整顶点着色器，以便绘制一个头朝下的三角形.
2. 请根据“纹理单元”（2.8）修改程序 2.2，用两个纹理图绘制正方形。

实验三　几何变换及三维绘图

一、实验目的

1．掌握几何变换的概念、相关的数学库及其基本操作.

2．应用 GLM 数学库，对基本图形进行几何变换.

3．掌握绘制有纹理贴图的三维立方体。

二、基本概念

1 几何变换

Opengl 没有矩阵、向量等内嵌的数学库。GLM 是专门为 opengl 开发的一个简单易用的数学库。下面的实验将要利用这个数学库完成相应的变换操作。

1.1 GLM 数学库

GLM，意思是 opengl Mathematics，是一种只需要头文件的数学库。也就是说，我们只要在项目中配置好头文件路径，在源码中包含相应的头文件即可.很多 GLM 的功能都可以在以下三个头文件中找到：

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

以下例子显示了如何利用 GLM 库将一个向量平移：

```
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;
```

我们首先使用 GLM 内置的向量类定义了一个向量 vec。然后定义了一个默认值为单位矩阵的 4*4 矩阵 trans。随后用 translate 函数创建平移矩阵。随后用平移矩阵乘原始向量，就得到变换后的结果了。以下代码构建了缩放和旋转操作的变换矩阵：

```
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));
```

注意，以上代码的变换矩阵是先进行 0.5 倍的缩放操作，然后进行 90 度的旋转操作。

不过我们怎样才能将变换矩阵传递给着色器呢？GLSL 也有 mat4 数据类型，因此我们可以让顶点着色器接受一个 mat4 uniform 的变量，然后将用这个矩阵乘顶点的位置即可：

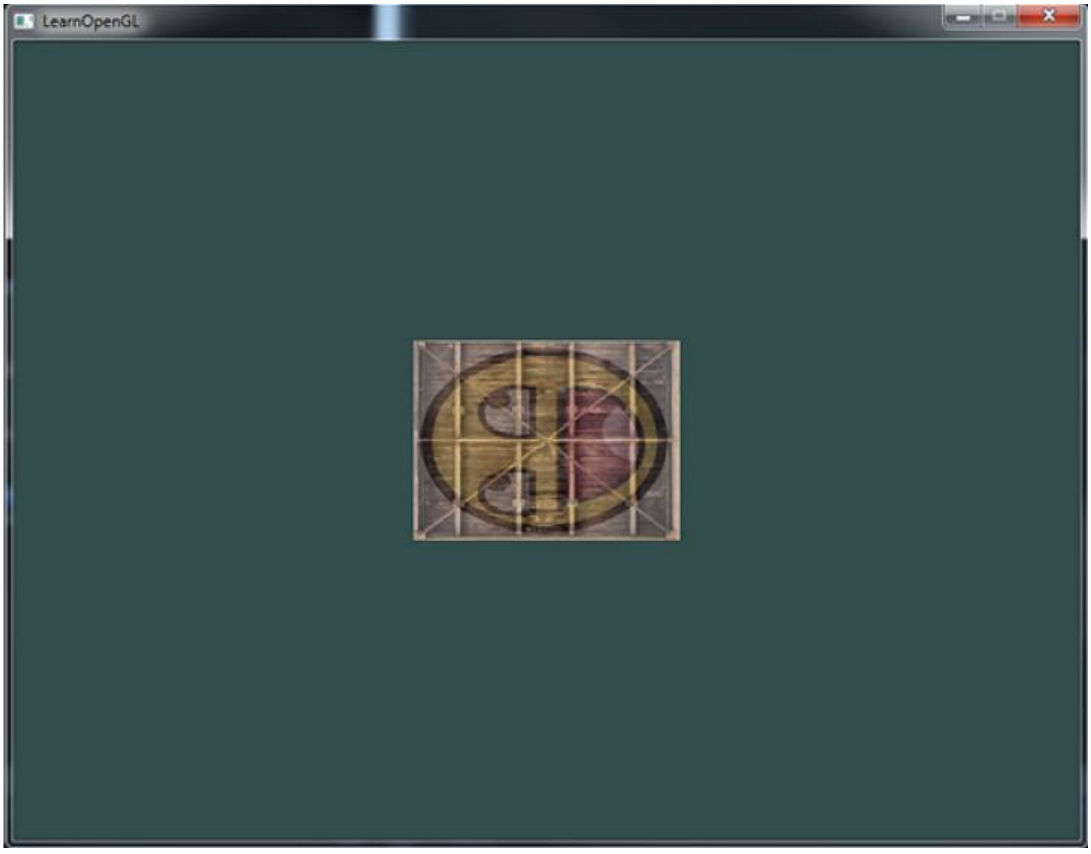
```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

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);
}
```

这样，我们绘制的物体就缩小了一半，然后旋转了 90 度。当然，我们还是需要将变换矩阵传给着色器：

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

以上代码中，我们先得到 uniform 变量的位置，然后将变换矩阵传给着色器。glUniformMatrix4fv 函数的第一个参数是 uniform 变量的位置，第二个参数告诉 opengl 需要传递多少个矩阵，这里是 1 个矩阵。第三个参数意思为是否需要对矩阵转置（opengl 按列存放矩阵，GLM 也是按列存放，故不需转置）。第四个参数为变换矩阵。这里用 GLM 的内置函数 value_ptr 对矩阵变换了一下，以匹配 opengl 所要求的类型。

变换后的图形效果如下：



现在，我们能否让物体动起来呢？为了让物体随时间变化不断宣战，我们需要在事件循环中不断更新变换矩阵：

```
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));
```

2.三维绘图

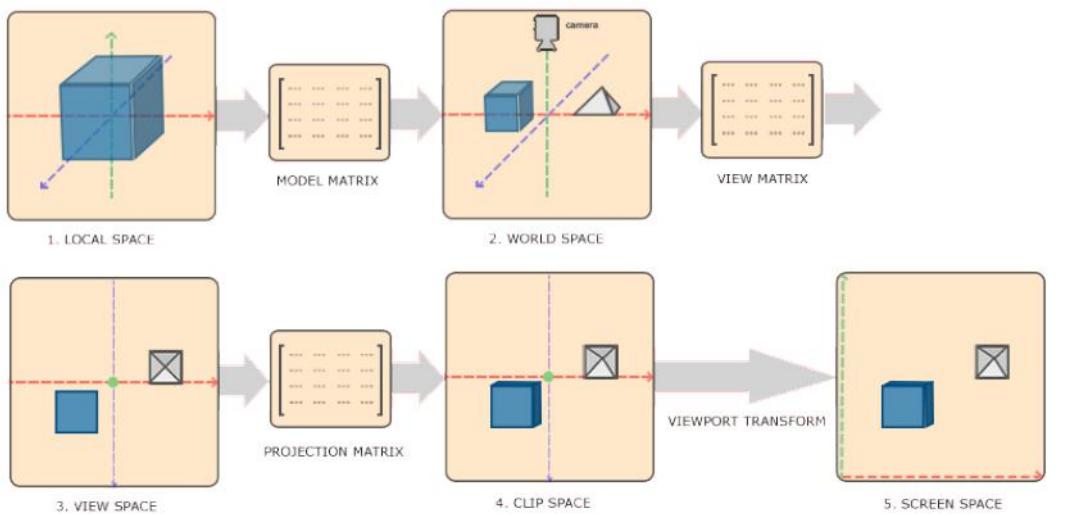
2.1 坐标系统

我们已经知道如何用变换矩阵对所有的顶点进行几何变换。在着色器运行完成后，OPENGL 期望可见顶点都在规整化的设备坐标(NDC)中。也就是说，x，y，z 坐标都在-1.0 到 1.0 之间，在此范围之外的顶点均不会被显示。为此，我们通常会先自己设置好要显示

的坐标范围，然后在顶点着色器中将这些坐标转换到 NDC。这些 NDC 坐标随后会送到渲染器（rasterizer）中，并被转换成我们屏幕上的 2D 坐标/像素。
将坐标转换到 NDC，然后转换为屏幕坐标的这个过程要经过几个步骤来完成。我们在将物体的顶点坐标转为屏幕坐标前，需要先将顶点转换到几个坐标系统中。将顶点转换为几个中间坐标系统中的好处是在一些坐标系统中，一些计算或操作会更加方便。对我们比较重要的坐标系统有如下 5 个：
模型坐标系（或局部坐标系）
世界坐标系
观察坐标系
裁剪坐标系
屏幕坐标系
下面，我们逐一说明每个坐标系的作用：

2.1.2 整体图像

为了将坐标从一个坐标系统转换到下一个坐标系统，我们将使用好几个不同的转换矩阵。最重要的矩阵为模型矩阵，观察矩阵和投影矩阵。顶点坐标首先从局部坐标系中以局部坐标开始，然后被转换为世界坐标，观察坐标，裁剪坐标，最后是屏幕坐标。以下显示了处理过程：



1. 局部坐标是物体相对其局部原点的坐标。
2. 下一步是将局部坐标转换为世界坐标，即相对一个更大世界的坐标。这些坐标是相对世界的原点来说的。世界坐标系中有很多其他的物体，其顶点坐标也是相对世界原点来说的。
3. 下一步，我们要讲世界坐标转换为观察坐标。这样，每个坐标都是相对于相机或观察者来说的。
4. 在坐标转换为观察坐标后，我们希望将它们映射为裁剪坐标。裁剪的处理范围是-1.0 到

1.0，将确定哪些顶点将会在屏幕上显示。
5. 最后，我们将裁剪坐标转换为屏幕坐标。这个过程是用过视口变换完成的，即将-1.0 到 1.0 坐标范围转换为由 glViewport 定义的屏幕坐标范围。最终的坐标将传给渲染器以将它们转换成面片（fragments）。
我们将顶点坐标转换成这么多不同的坐标系统中去，其原因是在特定操作时，采用某个坐标系统会让我们的工作更轻松。例如，在创建物体时，采用局部坐标更加容易，而当进行同其他物体位置相关的操作时，则采用世界坐标系更方便，如此等等。下面我们对每个坐标系统进行更详细的讨论：

2.1.3 局部坐标系统

局部坐标系统是指对物体来说，顶点坐标是局部的。例如，当我们利用工具软件创建一个立方体时，虽然最终这个立方体会放置在整个场景中的不同位置，这个立方体原点的局部坐标为（0,0,0）。

2.1.4 世界坐标系统

如果在工具软件中，将所有我们创建的物体都导入进来，则会产生哪个才是真正的世界原点（0,0,0）的困惑。我们要定义这个世界原点，并将每个物体放置在这个更大的世界中去。世界坐标的含义就是顶点相对于这个世界原点的坐标值。模型矩阵可以用来完成从局部坐标到世界坐标的转换。
模型矩阵是一个变换矩阵，可以将我们的物体进行平移、缩放及旋转操作，以便将物体放置在世界中的相应位置。

2.1.4 观察坐标系统

人们通常称观察者为 OpenGL 中的摄像机。观察坐标是将我们的世界坐标转换为用户镜头为参考的坐标，也就是以摄像头为参考原点看到的坐标。这个变换通常是平移即旋转等几个变换的复合，复合变换一般保存在观察矩阵中。

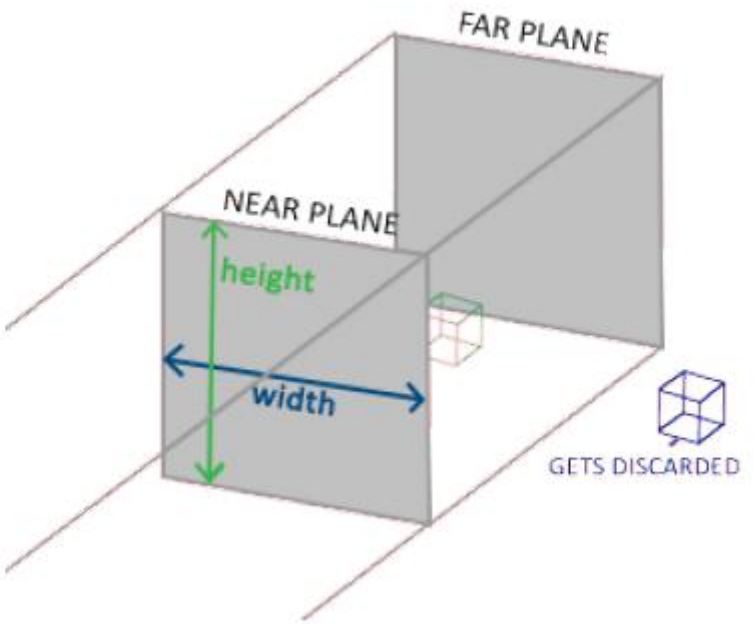
2.1.5 裁剪坐标系统

在顶点着色器运行结束时，OpengGL 期望顶点的坐标都落在特定的范围内，范围外的顶点都被裁剪掉。被裁剪掉的顶点就会被丢弃，只有余下的顶点在会显示在屏幕上。
为了将顶点坐标从观察坐标转换为裁剪坐标，我们需要定义一个投影矩阵，并指定每个方向的坐标范围，如-1000 到 1000。随后投影矩阵将这个范围转换到 NDC 坐标，即（-1.0，1.0）。所有在指定范围外的坐标将不会被映射到-1.0-1.0 之间，因此将被裁剪掉。以上面

为例，坐标（1250,500,750）将不可见，因为 x 坐标值落在了指定范围之外，因此转换后的 NDC 中 x 值将大于 1.0。
注意：如果一个三角形只有一部分位于裁剪体之外，则 OpenGL 将重建这个三角形为一个或多个三角形，以适应这个裁剪体内的范围。
这个投影矩阵创建的裁剪体也叫裁剪椎体，每个坐标都落在此椎体内的顶点经会显示在屏幕上。将此指定范围的椎体转换为 DNC（更方便映射到二维的观察坐标上去）的过程称为投影。
在所有顶点的坐标都被转换为裁剪坐标后，会进行一个操作：透视除法。透视除法将顶点齐次坐标的 x，y，z 分量都除以 w 分量，即将裁剪齐次坐标的 4D 形式转化为通常的 3D 形式。这个步骤在顶点着色器运行最后会自动进行。此后才会将 DNC 坐标映射到屏幕坐标上去（使用 glViewport 中的设置）。
将观察坐标转换为裁剪坐标的投影矩阵可以有两种形式，每种都有各自独特的锥台。即正投影矩阵和透视投影矩阵。

2.1.5.1 正投影

正投影矩阵定义了一个立方体状的裁剪锥台。当撞见一个正投影矩阵时，我们要指定可视锥台的长，宽，高。所有在此锥台内的顶点，当将观察坐标转换到裁剪坐标后，都不会被裁剪掉。锥台的形状如下：



锥台定义了可视坐标范围，并利用宽，高和近平面及远平面进行定义。所有在近平面之前及远平面之后的顶点都会被裁剪。由于在这中映射下，每个顶点向量的 w 分量都没有涉及到，因此正投影将锥台内的顶点直接映射到正规化的设备坐标上（NDC）（注意当 w 分量是 1.0 时，透视除法不会改变坐标值）。

要创建一个正投影矩阵，我们可以利用 GLM 中的内建函数 glm::ortho：

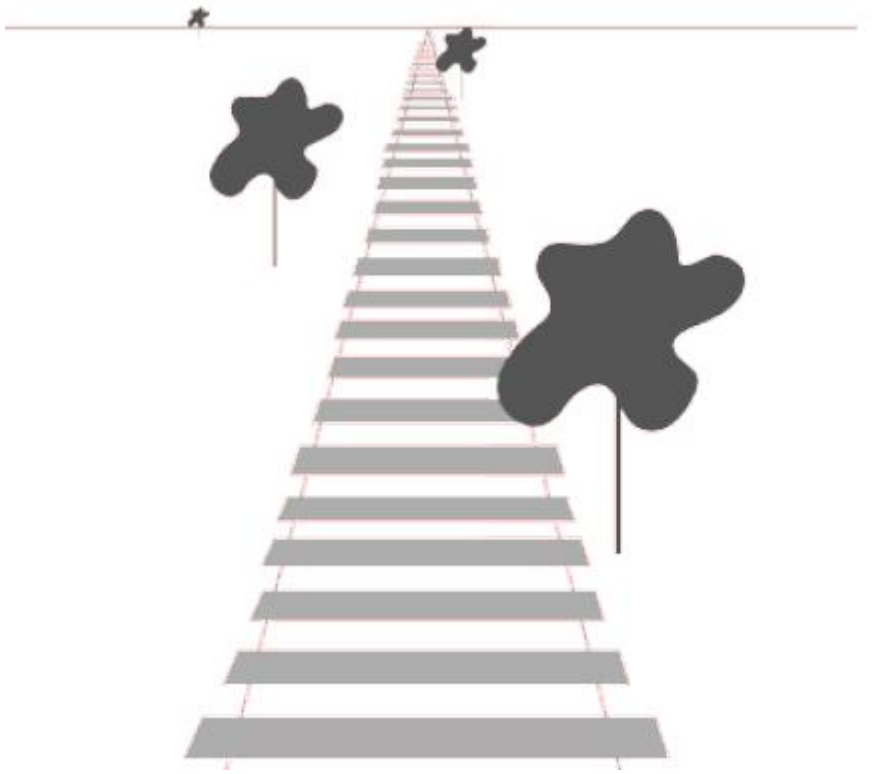
```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

前两个参数定义了锥台的左边界和右边界位置，第 3，第 4 个参数指定了锥台的底面和顶面。第 5，第 6 个参数定义了近平面及远平面的位置。这个投影矩阵将会将以上锥台内的顶点转换为 NDC 坐标。

正投影矩阵将坐标直接映射为 2D 的屏幕坐标。不过由于没有考虑透视效果，因此显得不是很真实。

2.1.5.1 透视投影

如果显示物体时，远处的物体小些，就会给人更真实的感觉。这就是透视的效果，如下所示：



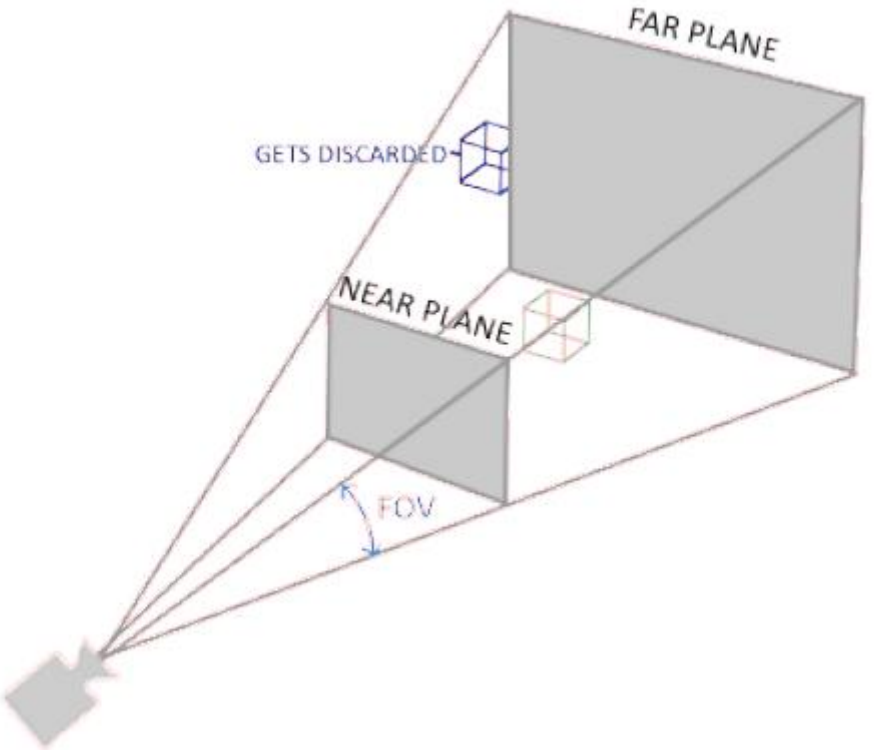
可以看到，由于透视，直线在很远的地方看起来汇聚在一起了。为了达到这种效果，OpenGL 采用了透视投影矩阵。投影矩阵将给定的锥台映射到裁剪空间中，而且会对每个顶点的 w 坐标进行操作。即越远的顶点坐标，其 w 坐标会变得越大。当坐标被转换为裁剪坐标后，其值在-w 到 w 之间（之外的都会被裁剪掉）。OpenGL 要求所有可视顶点的坐标都处于-1.0-1.0 之间。当坐标处于裁剪空间后，就会对其进行透视除法：

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

此时坐标就处于规格化设备坐标空间中了。
透视投影矩阵可以利用 GLM 创建：

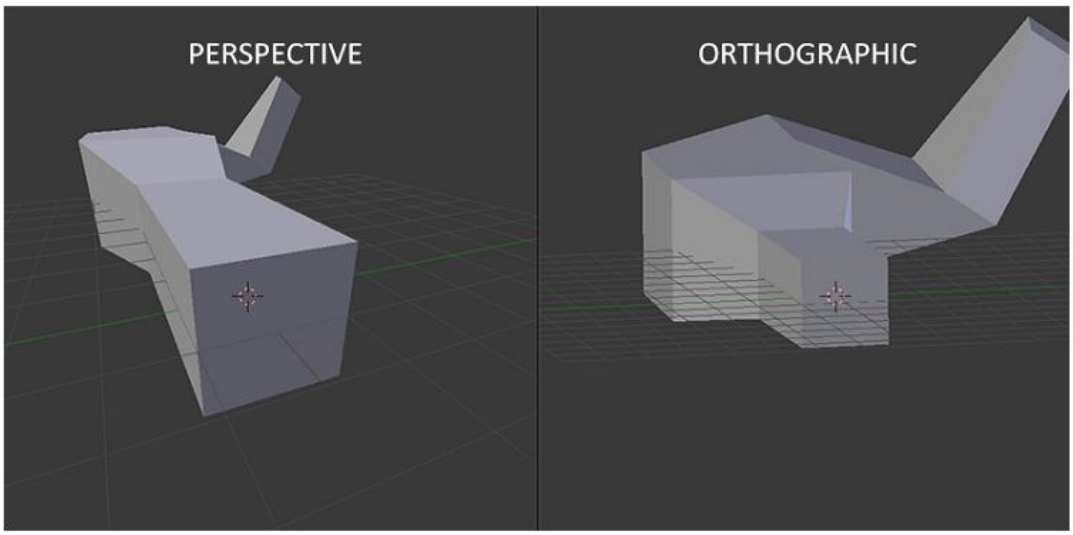
```
glm::mat4 proj = glm::perspective(45.0f, (float)width/(float)height, 0.1f,
    100.0f);
```

glm::perspective 做的事情是创建一个锥台，锥台之外的将落在裁剪范围之外，被裁剪掉。透视锥台可以看做是一个不堆成的盒子，盒子中的每个顶点都会映射到裁剪空间中。锥台形状如下：



这个函数的第一个参数定义了 fov 值，即视野的大小。视野值通常设为 45.0f，你也可以设为更大的值。第 2 个参数设置了长宽比率，由视口的宽除以长得到。第 3,4 个参数分别设置了近平面和远平面。一般讲近平面的距离设为 0.1f，远平面的距离设为 100.0f。
在使用正投影矩阵时，每个顶点坐标都直接映射到裁剪空间中，不会进行投射除法。由于正投影不进行透视投影，因此远处的物体也不会变小。因为这个特点，正投影通常用于建

筑设计或工程应用中。下图为透视投影和正投影的对比：



2.1.6 完整过程

我们创建了模型矩阵，观察矩阵和投影矩阵。一个顶点坐标由以下过程转换为裁剪坐标：

$$V\_clip = M\_projection \cdot M\_view \cdot M\_model \cdot V\_local$$

注意，矩阵相乘的顺序是反过来的。最后得到的顶点会赋给顶点着色器中的 gl_Position，然后 OpenGL 会自动执行透视除法和裁剪。

再然后呢？顶点着色器输出的坐标就是裁剪坐标了。OpenGL 会执行透视除法，以将它们转换到规格化设备坐标（NDC）。随后，会使用 glViewPort 中的参数将 NDC 坐标映射到屏幕坐标中。这个过程叫做视口变换。

2.2 三维绘图

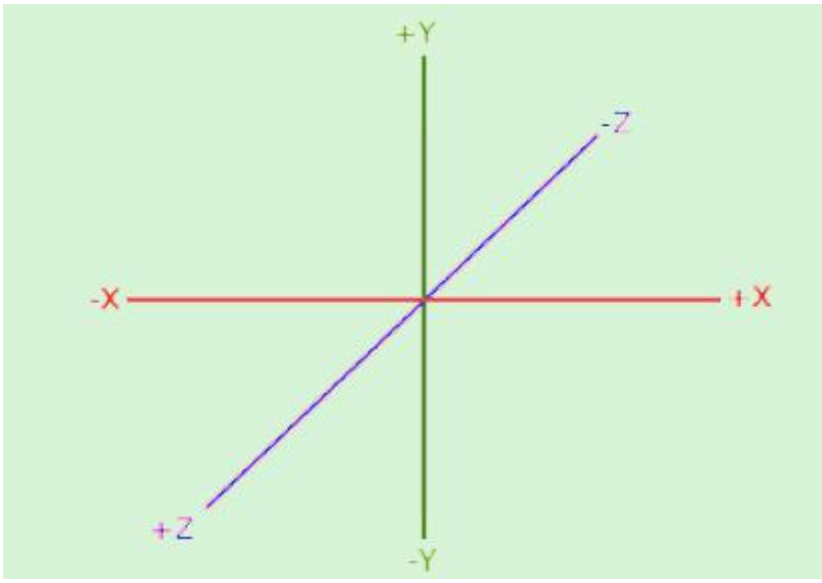
现在我们知道了如何将 3D 坐标转换为 2D 坐标，可以开始绘制真正的 3D 物体了。
要绘制 3D 物体，我们首先创建一个模型矩阵。模型矩阵含有平移，缩放和旋转，以便让我们将物体放置在世界空间中。让我们将以前的平面绕 x 轴旋转下，以便看起来像地面。模型矩阵如下：

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

模型矩阵乘顶点坐标后，就会将顶点坐标转换到世界坐标系中。
下面我们要创建观察矩阵。我们想让观察者后退一点，以便能看到整个物体（在世界坐标

系中，观察者位于原点（0,0,0））。注意，观察者（或摄像机）后退一点，同将整个场景前移一点是等价的。观察矩阵就是如此做的，想让观察者后退多少，我们就将整个场景前移多少。

由于我们采用右手坐标系，想将观察者前移，就是要想 z 轴正向移动。因此，我们将整个场景向-z 轴方向移动即可。Opengl 的坐标系统如下所示：



观察矩阵为：

```
glm::mat4 view;
//Notethatwe'retranslatingthesceneinthereversedirectionofwhere
    wewanttomove
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

最后我们要创建投影矩阵。因为我们想采用透视投影，因此我们需要声明如下的投影矩阵：

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

现在我们创建好了转换矩阵，我们还需要将其传送给着色器。首先我们要在顶点着色器中将转换矩阵声明为 uniforms，并将其同顶点坐标相乘：

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

voidmain()
{
    //NotethatwereadthemultiplicationfromrighttoleftgI
    gl_Position = projection * view * model * vec4(position, 1.0f);
    ...
}
```

然后我们再将矩阵传递给着色器（因为转换矩阵通常每个渲染循环都会发生变化，因此每个渲染循环时都要这样做）：

```
GLint modelLoc = glGetUniformLocation(ourShader.Program,"model"));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
...//SameforViewMatrixandProjectionMatrix
```

2.2 三维立方体

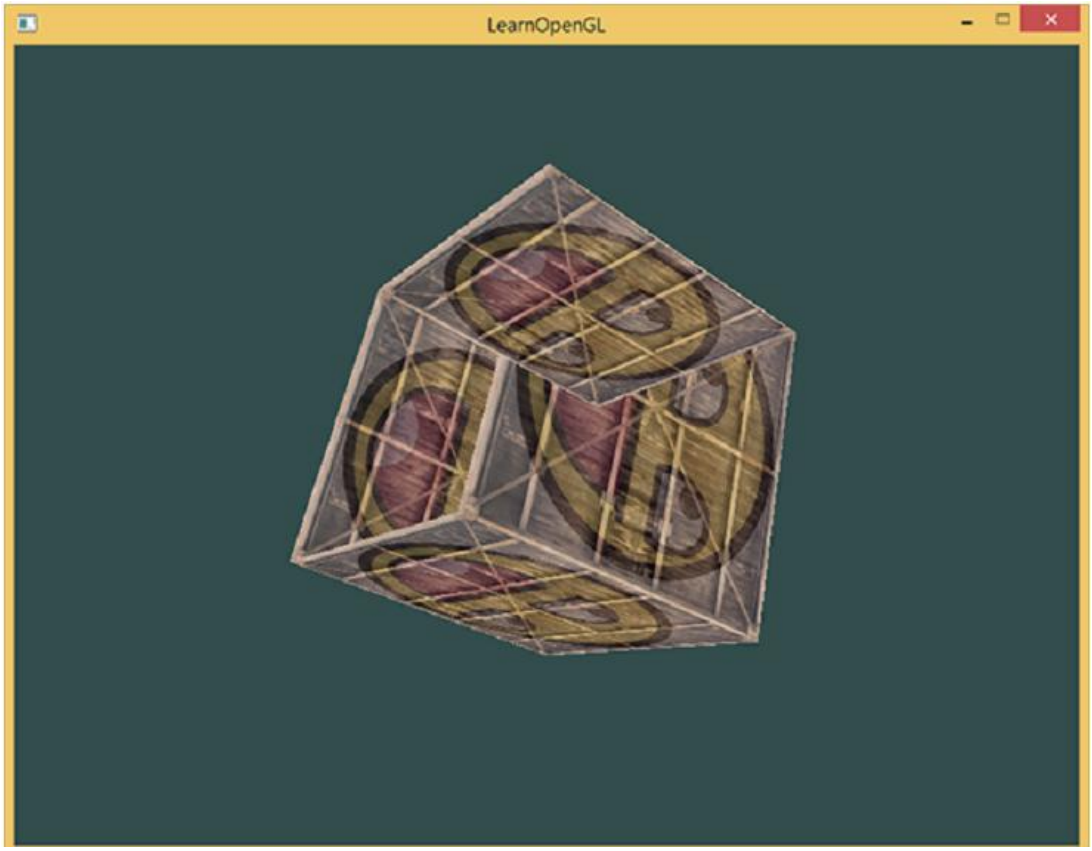
现在，我们要用所学知识绘制三维的立方体。为此，我们需要 36 个顶点（6 个面*2 三角形*3 顶点）。36 个顶点的数据很多，我们可以从本实验书的附件中拷贝下（注意，由于我们用文理得到最终颜色值，因此顶点数据中我们忽略了颜色值）。为了更有趣，我们还可以让立方体动态的旋转起来：

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

然后，就可以将其绘制出来了（注意，这次要有 36 个顶点）：

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

效果如图所示：



不过仔细观察会发现，立方体的一些边挡住了另外一些边。这是因为 opengl 绘制立方体时是逐个三角形绘制的，在绘制一个三角形时，就可能会覆盖掉原先绘制的三角形。
幸好 opengl 在一个 z-byffer 的缓存里保存了像素的深度值，这样就可以知道哪些三角形应该被遮挡，哪些不应该被挡住了。利用 z-buffer，我们可以让 opengl 执行深度测试。

2.2.1 z-buffer

Opengl 在 z-buffer 中保存所有的深度信息。GLFW 自动为我们创建了这个缓冲区。深度值保存在每个面片中（作为面片的 z 值），每当面片想输出其颜色值是，opengl 都会将其 z 值同 z-buffer 中的值进行比较。如果当前面片在其他面片之后，则当前面片的 z 值就会丢弃，否则将用当前面片的 z 值覆盖 z-buffer 中的值。这个过程也称为深度检测，也是由 opengl 自动完成的。
如果我们想让 opengl 执行深度检测，则我们需要告诉 opengl 启动深度检测功能。默认情况下是不启动的。我们用 glEnable 函数来做这件事：

```
glEnable(GL_DEPTH_TEST);
```

既然我们使用了深度缓存，因此我们在进行渲染迭代之前，还要清空深度缓存：

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

此时再运行程序，就会得到一个正确的旋转立方体了！

2.2.2 绘制更多的立方体

假如我们想显示 10 个立方体，每个立方体都相同，只是在世界坐标系中位置不同，旋转的角度不同。由于立方体每个面的空间位置已经定义好了，因此在绘制更多立方体时，我们不需要改变缓存区或属性数组了。唯一需要改变的是每个立方体的模型矩阵。我们用模型矩阵将立方体的坐标转换到世界坐标系中。
首先，我们对每个立方体定义一个平移向量，这个平移向量确定了立方体在世界空间中的位置。我们定义十个立方体的位置：

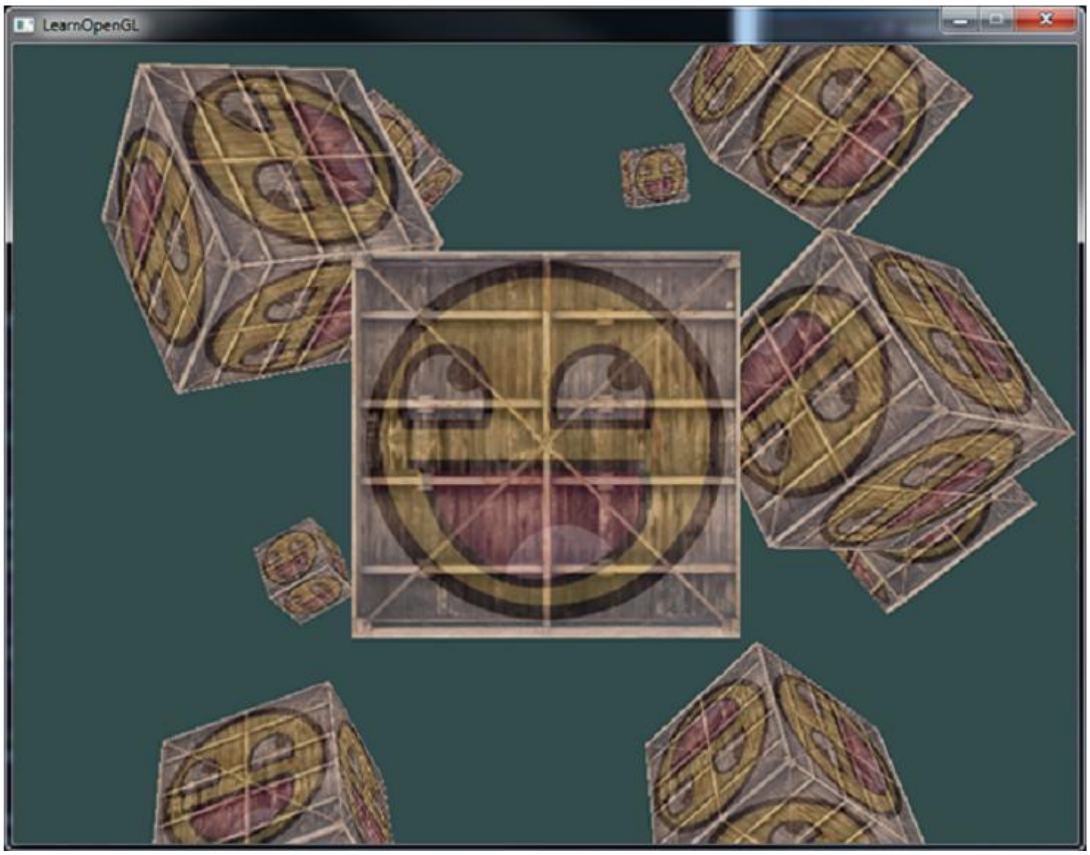
```
glm::vec3 cubePositions[] = {
glm::vec3( 0.0f, 0.0f, 0.0f),
glm::vec3( 2.0f, 5.0f, -15.0f),
glm::vec3(-1.5f, -2.2f, -2.5f),
glm::vec3(-3.8f, -2.0f, -12.3f),
glm::vec3( 2.4f, -0.4f, -3.5f),
glm::vec3(-1.7f, 3.0f, -7.5f),
glm::vec3( 1.3f, -2.0f, -2.5f),
glm::vec3( 1.5f, 2.0f, -2.5f),
glm::vec3( 1.5f,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

现在，在循环体中我们要调用 glDrawArrays 函数 10 次。不过每次要传递给顶点着色器不同的模型矩阵。我们将创建一个小循环来做这件事。注意，我们还对每个立方体施加了一个不同角度的旋转：

```
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);
```

现在绘制的效果如下所示：

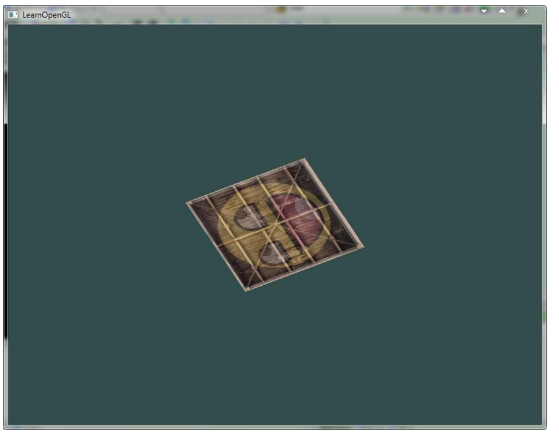


三、实验内容

（一）分析以下程序的原理并上机验证

1.首先调试并运行参考程序 3.1，用 GLM 数学库实现对长方形的几何变

换(旋转及缩放，见下图)。然后在参考程序基础上， 让长方形运动起来（即随时间变化,旋转不同角度)。(具体做法可参考前面着色器的概念讲述部分 1.1)。



[参考程序 3.1]

//

```
#include <iostream>

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

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

```
// Other Libs
#include <SOIL.h>
// GLM Mathematics
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Other includes
#include "Shader.h"



// Function prototypes
void key_callback(GLFWwindow* window, int key, int scancode, int
action, int mode);

// Window dimensions
const GLuint WIDTH = 800, HEIGHT = 600;

// The MAIN function, from here we start the application and run the
```

```
game loop
int main()
{
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's
functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT,
"LearnOpenGL", NULL, NULL);
    glfwMakeContextCurrent(window);

    // Set the required callback functions
    glfwSetKeyCallback(window, key_callback);
```

```
    // Set this to true so GLEW knows to use a modern approach to
retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);


    // Build and compile our shader program
    Shader ourShader("transformations.vs", "transformations.frag");


    // Set up vertex data (and buffer(s)) and attribute pointers
    GLfloat vertices[] = {
        // Positions          // Colors           // Texture
Coords
         0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f, // Top
```

```
Right
        0.5f, -0.5f, 0.0f,      0.0f, 1.0f, 0.0f,      1.0f, 0.0f, // Bottom
Right
        -0.5f, -0.5f, 0.0f,     0.0f, 0.0f, 1.0f,      0.0f, 0.0f, // Bottom
Left
        -0.5f,    0.5f, 0.0f,     1.0f, 1.0f, 0.0f,      0.0f, 1.0f    // Top Left
    };
    GLuint indices[] = {    // Note that we start from 0!
        0, 1, 3, // First Triangle
        1, 2, 3    // Second Triangle
    };
    GLuint VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
```

```
GL_STATIC_DRAW);

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

        // Position attribute
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(0);
        // Color attribute
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(1);
        // TexCoord attribute
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 *
sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
        glEnableVertexAttribArray(2);

        glBindVertexArray(0); // Unbind VAO
```

```
        // Load and create a texture
        GLuint texture1;
        GLuint texture2;
        // ====================
        // Texture 1
        // ====================
        glGenTextures(1, &texture1);
        glBindTexture(GL_TEXTURE_2D, texture1); // All upcoming
GL_TEXTURE_2D operations now have effect on our texture object
        // Set our texture parameters
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);     // Set texture wrapping to GL_REPEAT
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
        // Set texture filtering
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
```

```
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // Load, create texture and generate mipmaps
        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); // Unbind texture when
done, so we won't accidentily mess up our texture.
        // ===================
        // Texture 2
        // ===================
        glGenTextures(1, &texture2);
        glBindTexture(GL_TEXTURE_2D, texture2);
        // Set our texture parameters
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
```

```
GL_REPEAT);
        // Set texture filtering
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // Load, create texture and generate mipmaps
        image = SOIL_load_image("awesomeface.png", &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);



        // Game loop
        while (!glfwWindowShouldClose(window))
        {
            // Check if any events have been activiated (key pressed,
```

```
mouse moved etc.) and call corresponding response functions
        glfwPollEvents();

        // Render
        // Clear the colorbuffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);


        // Bind Textures using texture units
        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);

        // Activate shader
```

```
        ourShader.Use();

        // Create transformations
        glm::mat4 transform;
    transform = glm::rotate(transform, 90.0f, glm::vec3(0.0, 0.0,
1.0));
    transform = glm::scale(transform, glm::vec3(0.5, 0.5, 0.5));

        // Get matrix's uniform location and set matrix
        GLint transformLoc =
glGetUniformLocation(ourShader.Program, "transform");
        glUniformMatrix4fv(transformLoc, 1, GL_FALSE,
glm::value_ptr(transform));

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

```
        // Swap the screen buffers
        glfwSwapBuffers(window);
    }
    // Properly de-allocate all resources once they've outlived their purpose
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}

// Is called whenever a key is pressed/released via GLFW
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
```

```
//////////////////////////
//shadeer.h
#ifndef SHADER_H
#define SHADER_H

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

#include <GL/glew.h>

class Shader
{
public:
    GLuint Program;
    // Constructor generates the shader on the fly
    Shader(const GLchar* vertexPath, const GLchar* fragmentPath)
```

```
{
    // 1. Retrieve the vertex/fragment source code from filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // ensures ifstream objects can throw exceptions:
    vShaderFile.exceptions (std::ifstream::badbit);
    fShaderFile.exceptions (std::ifstream::badbit);
    try
    {
        // Open files
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // Read file's buffer contents into streams
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // close file handlers
        vShaderFile.close();
```

```
            fShaderFile.close();
            // Convert stream into string
            vertexCode = vShaderStream.str();
            fragmentCode = fShaderStream.str();
        }
        catch (std::ifstream::failure e)
        {
            std::cout <<
"ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" <<
std::endl;
        }
        const GLchar* vShaderCode = vertexCode.c_str();
        const GLchar * fShaderCode = fragmentCode.c_str();
        // 2. Compile shaders
        GLuint vertex, fragment;
        GLint success;
        GLchar infoLog[512];
        // Vertex Shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
```

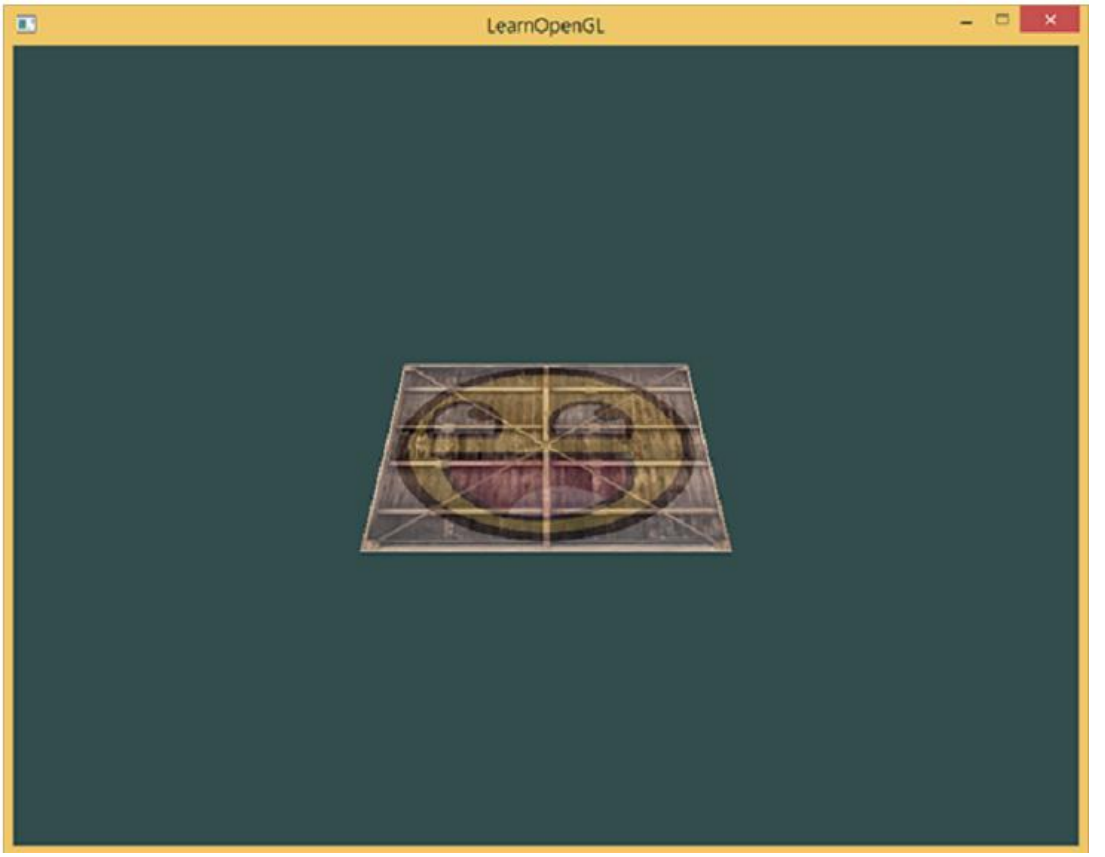
```
            glCompileShader(vertex);
            // Print compile errors if any
            glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
            if (!success)
            {
                glGetShaderInfoLog(vertex, 512, NULL, infoLog);
                std::cout <<
"ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
infoLog << std::endl;
            }
            // Fragment Shader
            fragment = glCreateShader(GL_FRAGMENT_SHADER);
            glShaderSource(fragment, 1, &fShaderCode, NULL);
            glCompileShader(fragment);
            // Print compile errors if any
            glGetShaderiv(fragment, GL_COMPILE_STATUS,
&success);
            if (!success)
            {
                glGetShaderInfoLog(fragment, 512, NULL, infoLog);
```

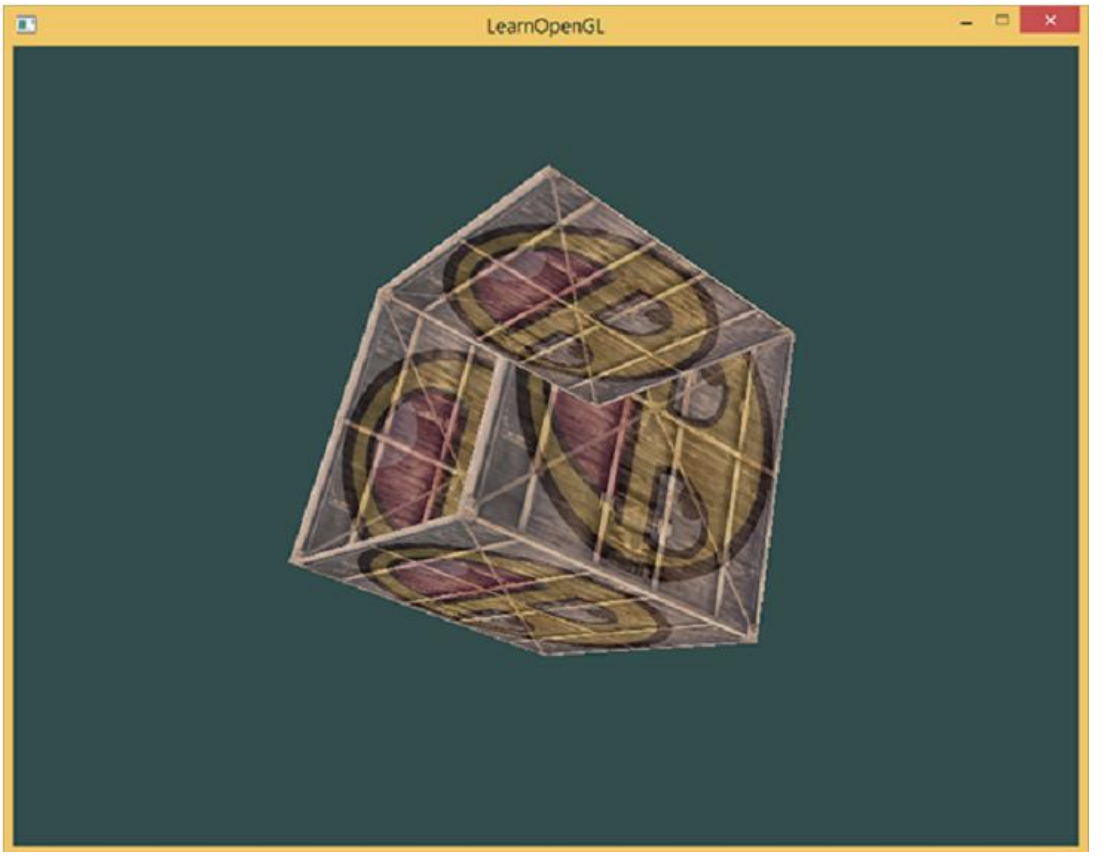
```
            std::cout <<
"ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" <<
infoLog << std::endl;
        }
        // Shader Program
        this->Program = glCreateProgram();
        glAttachShader(this->Program, vertex);
        glAttachShader(this->Program, fragment);
        glLinkProgram(this->Program);
        // Print linking errors if any
        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;
        }
```

```
        // Delete the shaders as they're linked into our program now and no longer necessery
        glDeleteShader(vertex);
        glDeleteShader(fragment);


    }
    // Uses the current shader
    void Use()
    {
        glUseProgram(this->Program);
    }
};


#endif
//////////////////
//transformations.frag
#version 330 core
in vec3 ourColor;
in vec2 TexCoord;

out vec4 color;

// Texture samplers
```

```
uniform sampler2D ourTexture1;
uniform sampler2D ourTexture2;

void main()
{
    // Linearly interpolate between both textures (second texture is only slightly combined)
    color = mix(texture(ourTexture1, TexCoord), texture(ourTexture2, TexCoord), 0.2);
}
/////////////////
//transformations.vs
#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);
}
```

2. 首先调试并运行参考程序 3.2，绘制一个有透视效果的的四边形。然后在参考程序基础上，修改代码，绘制出一个立方体：





[参考程序 3.2]

```
//////////////////////////////////
#include <iostream>

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

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

// Other Libs
#include <SOIL.h>
// GLM Mathematics
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

```
// Other includes
#include "Shader.h"


// Function prototypes
void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mode);

// Window dimensions
const GLuint WIDTH = 800, HEIGHT = 600;

// The MAIN function, from here we start the application and run the game
loop
int main()
{
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "LearnOpenGL",
nullptr, nullptr);
    glfwMakeContextCurrent(window);

    // Set the required callback functions
    glfwSetKeyCallback(window, key_callback);

    // Set this to true so GLEW knows to use a modern approach to
retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);
```

```
    // Build and compile our shader program
    Shader ourShader("path/to/shaders/shader.vs",
"path/to/shaders/shader.frag");


    // Set up vertex data (and buffer(s)) and attribute pointers
    GLfloat vertices[] = {
        // Positions          // Texture Coords
         0.5f,  0.5f, 0.0f,   1.0f, 1.0f, // Top Right
         0.5f, -0.5f, 0.0f,   1.0f, 0.0f, // Bottom Right
        -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, // Bottom Left
        -0.5f,  0.5f, 0.0f,   0.0f, 1.0f  // Top Left
    };
    GLuint indices[] = {  // Note that we start from 0!
        0, 1, 3, // First Triangle
        1, 2, 3  // Second Triangle
    };
    GLuint VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

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

    // Position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
(GLvoid*)0);
    glEnableVertexAttribArray(0);
    // TexCoord attribute
```

```
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
(GLvoid*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(2);

    glBindVertexArray(0); // Unbind VAO


    // Load and create a texture
    GLuint texture1;
    GLuint texture2;
    // =====================
    // Texture 1
    // =====================
    glGenTextures(1, &texture1);
    glBindTexture(GL_TEXTURE_2D, texture1); // All upcoming
GL_TEXTURE_2D operations now have effect on our texture object
    // Set our texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);    // Set te
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Set texture filtering
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Load, create texture and generate mipmaps
    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); // Unbind texture when done, so we
won't accidentily mess up our texture.
    // =====================
    // Texture 2
    // =====================
    glGenTextures(1, &texture2);
    glBindTexture(GL_TEXTURE_2D, texture2);
    // Set our texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Set texture filtering
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Load, create texture and generate mipmaps
    image = SOIL_load_image("awesomeface.png", &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);


    // Game loop
    while (!glfwWindowShouldClose(window))
    {
        // Check if any events have been activated (key pressed, mouse
moved etc.) and call corresponding response functions
        glfwPollEvents();

        // Render
        // Clear the color buffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);


        // Bind Textures using texture units
        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);

        // Activate shader
        ourShader.Use();
```

```
        // Create transformations
        glm::mat4 model;
        glm::mat4 view;
        glm::mat4 projection;
        model = glm::rotate(model, -55.0f, glm::vec3(1.0f, 0.0f,
0.0f));
        view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
        projection = glm::perspective(45.0f, (GLfloat)WIDTH /
(GLfloat)HEIGHT, 0.1f, 100.0f);
        // Get their uniform location
        GLint modelLoc = glGetUniformLocation(ourShader.Program,
"model");
        GLint viewLoc = glGetUniformLocation(ourShader.Program,
"view");
        GLint projLoc = glGetUniformLocation(ourShader.Program,
"projection");
        // Pass them to the shaders
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(model));
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE,
glm::value_ptr(view));
        // Note: currently we set the projection matrix each frame, but
since the projection matrix rarely changes it's often best practice to
set it outside the main loop only once.
        glUniformMatrix4fv(projLoc, 1, GL_FALSE,
glm::value_ptr(projection));

        // Draw container
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

        // Swap the screen buffers
        glfwSwapBuffers(window);
    }
    // Properly de-allocate all resources once they've outlived their
purpose
    glDeleteVertexArrays(1, &VAO);
```

```
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}

// Is called whenever a key is pressed/released via GLFW
void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}
//////////////////////
//shader.vs

#version 330 core
layout (location = 0) in vec3 position;
layout (location = 2) in vec2 texCoord;

out vec2 TexCoord;

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

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    TexCoord = vec2(texCoord.x, 1.0 - texCoord.y);
}

//////////////
//shader.frag

#version 330 core
in vec2 TexCoord;
```

```
out vec4 color;

uniform sampler2D ourTexture1;
uniform sampler2D ourTexture2;

void main()
{
    color = mix(texture(ourTexture1, TexCoord), texture(ourTexture2,
TexCoord), 0.2);
}
```

（二）按要求进行算法设计并调试运行

1．请根据“三维绘图”所述内容，修改参考程序 3.2，绘制多个立方体。

实验四　相机漫游

一、实验目的

1．了解和掌握相机的基本概念、使用方法.

2．掌握相机漫游的基本概念及编程方法，并能灵活应用.

二、基本概念

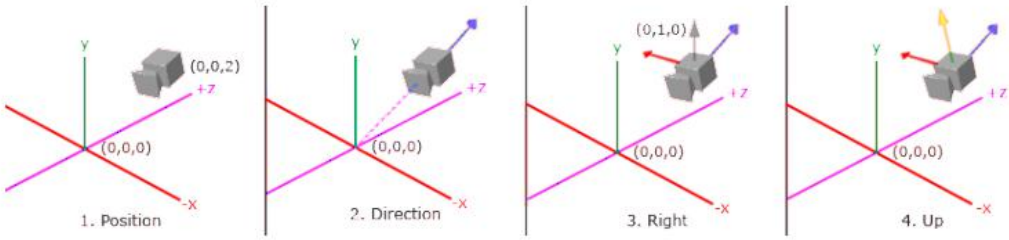
我们已经知道如何利用观察矩阵在场景中移动。Opengl 本身没有相机，可视我们可以通过将所有物体反向移动，造成我们自己在场景中正向移动的效果，来模拟一个相机。

在本实验中，我们将要学习如何在 opengl 中设置一个相机，以及如何利用一个 FPS 风格的相机在 3D 场景中自由漫游。我们还将学习如何处理键盘及鼠标输入，以及如何建立一个相机类。

1．相机/观察空间

当我们说相机/观察空间的时候，我们是以相机作为原点的：观察矩阵将所

有的世界坐标系中的坐标都转换到了观察坐标，即相对相机方位的坐标。为了定义相机，我们需要相机在世界坐标系中的位置，以及相机的朝向，即一个指向相机右方的向量和一个指向上方的向量。细心的人会注意到，我们实际上是在建立一个以相机为原点，三个相互垂直向量为坐标轴的坐标系统。



1.1 相机位置

得到相机位置是很容易的。相机位置实际上就是一个世界坐标系中的一个向量，它指向相机。本实验中，我们设置如下的相机位置：

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

1.2 相机指向

还需要一个相机指向的向量。现在我们让相机指向原点（0,0,0）。两个向量相减，就得到两个向量之差。相机的位置向量减去场景的原点，就得到由原点指向相机的向量了。我们已经知道相机指向 z 的负方向，而我们想让相机指向这个向量指向 z 轴正方向，因此我们需要点到以下相减的顺序：

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

注：称相机指向也许不确切，因为这个向量实际是指向相机朝向的反方向。

1.3 向右向量

下面我们要建立向右的向量，即 x 轴的正向。为此，我们用了一个小技巧，即先利用向上的向量（在世界坐标系中），然后用对向上向量及相机指向向量这两个向量进行叉积，即可得到向右向量：

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

1.4 向上向量

目前，我们已经有 x 轴即 z 轴向量了，下面要得到 y 轴向量就相对容易了：我们将向右向量和指向向量叉积即可：

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

利用这些向量，我们现在可以创建非常有用的 LookAt 矩阵了。

2. Look At

矩阵的一个特点是，如果你用 3 个相互垂直的坐标轴定义了一个坐标系，则你就可以利用这 3 个坐标轴以及一个平移向量创建一个矩阵，这个矩阵同任何一个向量相乘，即可将其转换到以上定义的坐标系里。这也是 LookAt 矩阵索要做的事情。我们有了定义相机/观察空间的 3 个相互垂直的坐标轴以及相机的位置向量，现在我们就可以创建自己的 LookAt 矩阵了：

$$LookAt = \begin{bmatrix} R\_xR & \_yR & \_z & 0 \\ U\_xU & \_yU & \_z & 0 \\ D\_xD & \_yD & \_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P\_x \\ 0 & 1 & 0 & -P\_y \\ 0 & 0 & 1 & -P\_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中 R 是向右向量，U 是向上向量，D 是相机朝向向量，P 是相机位置向量。注意，由于我们想让场景向我们移动方向的相反方向移动，因此位置向量取了个负号。将这个 LookAt 矩阵作为我们的观察矩阵，将会将所有的世界坐标都转换到这个刚刚定义的观察坐标系中。所以说，LookAt 矩阵创建了一个看向给定目标的观察矩阵。

幸运的是，GLM 已经为我们做好了这一切，我们只需要制定相机位置，所看目标位置和在世界坐标系中表示向上的向量即可。GLM 就会创建一个 LookAt 矩阵，我们即可以将其作为观察矩阵：

```
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 函数需要一个位置向量，一个所看目标向量以及一个向上向量，即可创建出我们所需要的观察矩阵。

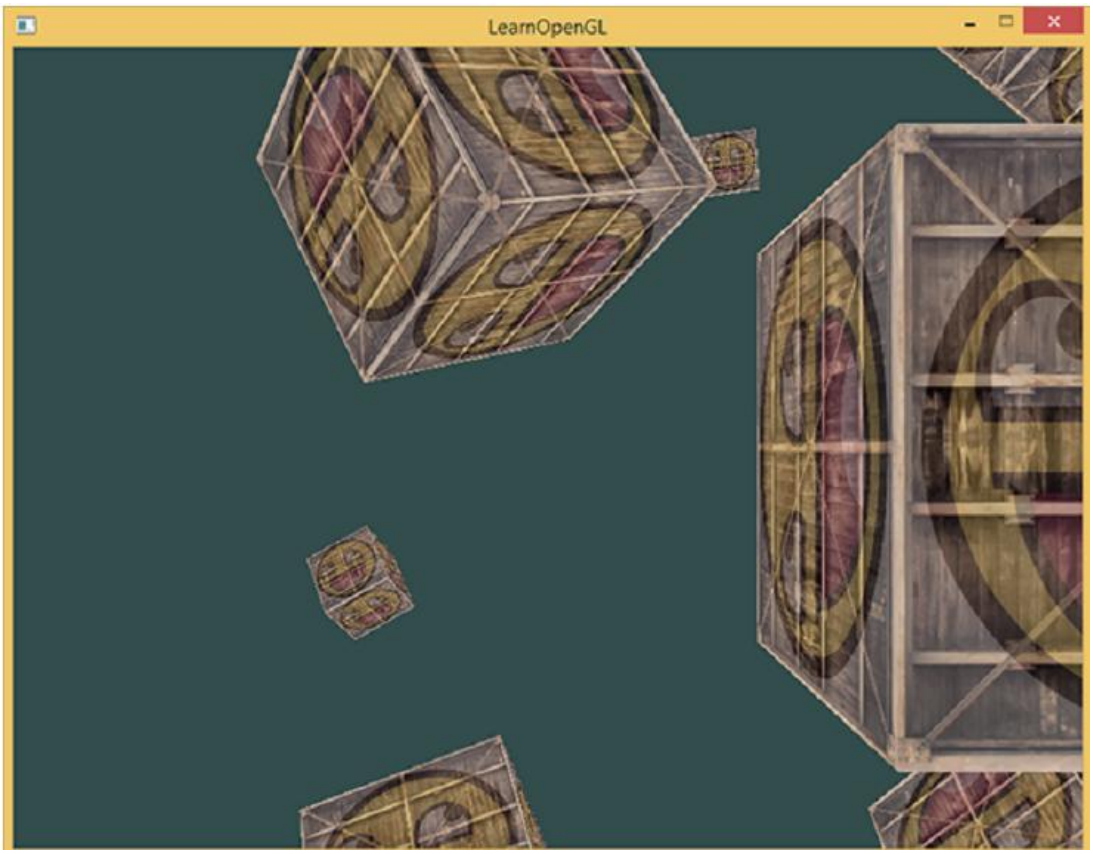
下面我们让相机旋转起来，并保持看向目标点（0,0,0）。

我们用一些几何知识让每帧的相机位置保持一个圆上。通过重复计算 x 和 z 坐标，我们即可在一个圆上运动，从而让相机旋转起来。我们用 radius 这个参数将这个圆放大，并在每个渲染循环中重新定义观察矩阵：

```
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));
```

效果如下：



3. 漫游

虽然让相机在场景中移动是很有意思的，不过如果能让我们自己控制这些移动则更有意思！首先我们需要建立相机系统，我们可以在程序开头定义以下变量：

```
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);
```

现在 LookAt 函数变成：

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

首先我们将相机位置设为前面定义的 cameraPos，所看的方向是当前位置+前面定义的方向向量。这样就可以保证不管我们怎样移动，相机始终指向目标。下面，我们希望按下按键时，cameraPos 向量会更新。

我们已经为 GLFW 的键盘输入定义了 key_callback 函数，让我们添加一些新的按键命令：

```
voidkey_callback(GLFWwindow * window,intkey,intscancode,intaction,
    intmode)
{
    ...
    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;
}
```

每当我们按下了 WASD 中的一个键时，相机的位置都会进行更新。如果我们想向前或向后移动，我们可以从位置向量中加上或减去方向向量。如果我们想想旁边移动，我们可以用叉积来创建一个向右的向量，然后我们顺着此向右向量移动。注意我们要将向右的向量单位化，否则就可能造成移动的速度过快或过慢。

在使用以上的相机漫游系统漫游时，你会发现不能同时向两个方向移动（即向对角方向移动），而且当你按下一个按键不放后，相机会先暂停一下，然后才开始移动。之所以有这个现象，是因为很多事件输入系统中，只能一次处理一个按键，而且相应的回调函数只有在我们激活了这个按键后才会被调用。这对大多数 GUI 系统都是适用的，但它对处理平缓的相机运动却不太合适。我们可以用一些小技巧来解决这个问题。

这个技巧是在回调函数中记录下键的按下/释放状态。在程序循环中，我们读取相应的记录，检查什么键被激活了，并作出相应的反应。首先，我们创建一个记录键的按下/释放状态的布尔类型数组：

```
boolkeys[1024];
```

然后我们在 key_callback 回调函数中将相应的值设为 ture 或 false：

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

下面我们创建一个叫 do_mouement 的新函数，当我们按下按键时，我们将在这里更新相机位置：

```
voiddo_movement()
{
//Cameracontrols
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;
}
```

以前的相应代码现在都移动到这个 do_movement 函数里。由于所有 GLFW 的按键类型实际都是整数，因此我们可以用他们作为状态数组的下标。

当然，我们还要在程序循环中假如调用新函数的代码：

```
while(!glfwWindowShouldClose(window))
{
//Checkandcallevents
glfwPollEvents();
do_movement();

//Renderstuff
...
}
```

现在，我们应该可以向任何方向移动了，而且按下按键不放时，相机也会立即移动。

4. 移动速度

目前，我们是以恒速运动的。实际上，人们移动的速度是不同的。一个人会在一秒内比别人绘制更多的帧。当他绘制更多的帧时，他也比别人调用 do_movement 的次数更多。当我们发布应用时，我们希望在所有的机器上，人们移动的速度都相同。

图形应用软件及游戏软件一般都有个 deltaTime 变量，用来保存渲染最新一帧图片所用的时间。然后我们将所有的速度都同这个变量相乘，结果就是当我们的 deltaTime 变量比较大时，意味着最新一帧需要更多时间来显示，因此需要将此帧中的移动速度调高一些以补偿显示所需时间。这样每个用户都会体验到同样的移动速度了。

为了计算这个 deltaTime 值，我们要定义 2 个全局变量：

```
GLfloat deltaTime = 0.0f;//Timebetweencurrentframeandlastframe
GLfloat lastFrame = 0.0f;//Timeoflastframe
```

每帧中我们计算新的 deltaTime 值以备后用：

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

现在我们在计算速度时，即可考虑到 deltaTime 值了：

```
voidDo_Movement()
{
GLfloat cameraSpeed = 5.0f * deltaTime;
...
}
```

同前面相比，我们现在的系统中相机移动的就平滑很多，而且移动速度在各个系统中也很统一。

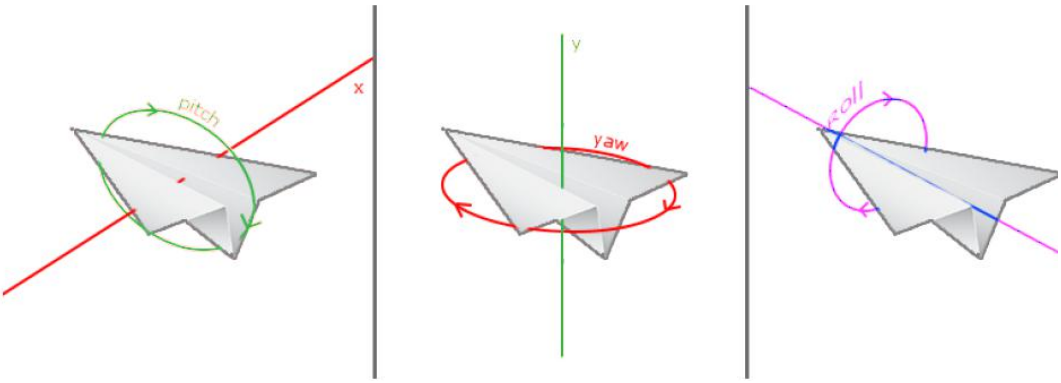
5. Look around

只使用键盘按键在场景中移动不是特别有意思。特别是我们无法翻转，导致移动受到一定限制。这时，就可以利用鼠标了。

为了在场景中翻转，我们需要基于鼠标输入对 cameraFront 向量加以改变。然而，基于鼠标动作改变方向向量有点复杂，需要一些几何知识。同学们也可以先将这部分内容略过去，直接粘贴代码即可。

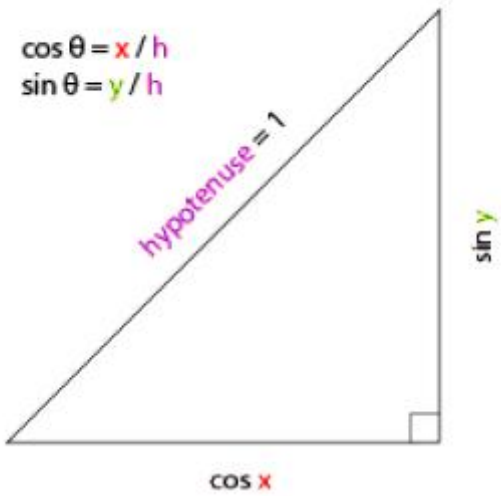
6. 欧拉角

欧拉角由欧拉在大约 1700 年定义，有 3 个值，可以表示三维空间中的任何旋转。这三个角度为：pitch,yaw 及 roll 。以下为图示：

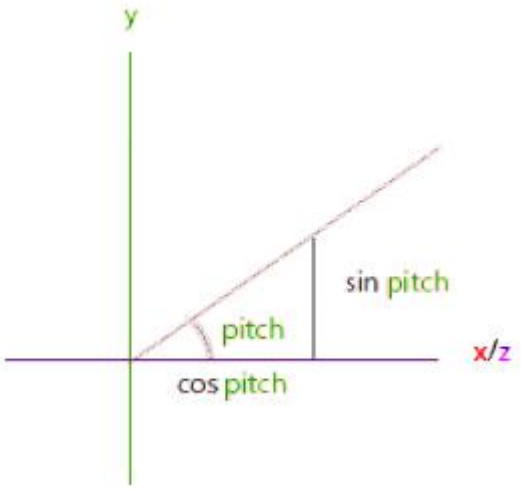


Pitch（俯仰角）是我们上下看的角度。Yaw 是我们向左右看的角度。Roll 则是翻转角度。每个欧拉角都由一个值表示，得到三个值我们即计算出三维空间中的任意旋转了。

对于我们的相机系统，我们只关心 yaw 和 pitch 值。给定一个 pitch 和 yaw 值，我们可以将它们转化为 3 维空间中的一个方向向量。我们先从最基本的开始：



如果斜边长度为 1，则相邻两边长度分别为 cos θ，sin θ。我们可以由角度得到 x 及 y 方向的长度。现在我们计算单位方向向量：



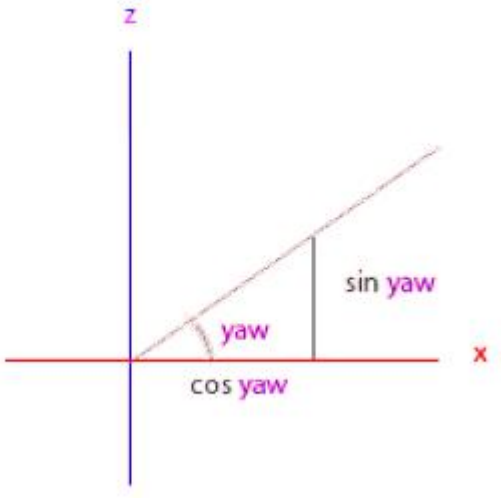
这个图同上一个图类似。若已知 pitch(俯仰角) θ，我们可以得到单位方向向量的 y 值为 sin θ：

```
direction.y = sin(glm::radians(pitch));//Notethatweconverttheangle
    toradiansfirst
```

单位方向向量的 x 和 z 值分别为：

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

类似的，已知 yaw 值，我们也可以得到相应的关系：



可以看出，x 分量同 cos(yaw)相关，z 分量同 sin（yaw）相关。同上面的联系起来，我们即可由 pitch 及 yaw 得到方向向量来：（可以想象下三维情况，y 轴是向上的）

```
direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));
direction.y = sin(glm::radians(pitch));
direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));
```

现在的问题是：如何得到 yaw 及 pitch 值呢？

6. 鼠标输入

我们要从鼠标输入中得到 yaw 和 pitch 值。水平的鼠标运动影响 yaw 值，垂直的鼠标运动影响 pitch 值。其思想是保存最近一帧的鼠标位置，然后在当前帧计算现在鼠标位置同最近一帧鼠标位置的变化值。水平或垂直值相差越大，则我们将 pitch 或 yaw 值设置的越高，相机就会做相应的运动了。

首先我们要告诉 GLFW，让其隐藏并捕获光标。捕获光标意味着当我们的应用得到焦点时，光标会保持在我们的应用窗口之内（除非应用失去焦点或应用关闭）。我们可以调用以下函数完成这件事情：

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

调用过此函数后，无论我们怎样移动鼠标，光标都不可见，而且都不会离开窗口。

为计算 pitch 和 yaw 值，我们需要告诉 GLFW 去监听鼠标移动事件。这可以痛苦创建一个回调函数来实现（同键盘输入的回调函数类似）：

```
voidmouse_callback(GLFWwindow * window,doublexpos,doubleypos);
```

其中 xpos 和 ypos 表示当前鼠标位置。只要我们将这个回调函数注册到 GLFW 中，每次鼠标移动，这个回调函数 mouse_callback 都会被调用：

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

在最后得到方向向量之前，我们还要完成以下几步：

1. 计算自从最近一帧以来的鼠标偏移。

2. 将偏移值加到相机的 yaw 和 pitch 值上。

3. 施加一些诸如最大/最小 yaw/pitch 值的限制条件。

4. 计算方向向量。

第一步是计算自从最近一帧以来的鼠标偏移。我们首先要保存最近一帧时鼠标的位置。我们将初始值设为屏幕中间（屏幕是 800*600）：

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

然后在鼠标的回调函数中，我们计算两帧之间的鼠标位移：

```
GLfloat xoffset = xpos - lastX;
GLfloat yoffset = lastY - ypos;//Reversedsincey-coordinatesrangefrom
    bottomtotop
lastX = xpos;
lastY = ypos;
```

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

注意，我们要将偏移量同 sensitivity 相乘。如果忽略这个的话，鼠标移动的就会太快了，导致我们相机翻转的很厉害。

下面我们要将这个偏移值加到全局变量 pitch 和 yaw 上去：

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

第三步，我们要对相机施加一些限制。Pitch 被限制到不高于 89 度（90 度时，场景看起开会颠倒过来），不低于-89 度。这就保证了观察者可以向上看到天空，向下看到地板：

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

因为我们不想限制水平旋转，因此没有对 yaw 值施加限制。当然，如果你想限制它的话，也可以很容易的做到。

第四和最后一步，是由 yaw 和 pitch 值计算实际的方向向量：

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

如果现在运行代码，会发现每当窗口第一次获得鼠标焦点时，相机都会有个大

的跳跃。这是因为只要光标进入窗口，鼠标回调函数都会执行，此时 xpos 和 ypos 都等于鼠标进入窗口的位置。而这个位置一般距离屏幕中心很远。因而导致了运动的跳跃。我们可以定义一个全局的 bool 型变量，检查是否第一次收到鼠标焦点。如果是的话，则我们将鼠标的初始位置设为这个新的 xpos 和 ypos 值。其余部分则使用鼠标的位置计算相应的偏移：

```
if(firstMouse)//thisboolvariableisinitiallysettotrue
{
lastX = xpos;
lastY = ypos;
firstMouse =false;
}
```

最终代码如下：

```
voidmouse_callback(GLFWwindow * window,doublexpos,doubleypos)
{
    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)) * cos(glm::radians(pitch));
    cameraFront = glm::normalize(front);
}
```

现在我们可以在 3 维世界中自由的漫游了！

7．缩放

我们现在可以加一个缩放功能。以前我们说过，视野（field of view）fov 定义了我们能看到多大的场景。当视野小时，仓井映射后就会很小，给人一种缩小的感觉。为了缩小，我们可以利用鼠标的滑轮。同键盘和鼠标移动的回调函数类似，我们定义鼠标滑轮的回调函数：

```
voidscroll_callback(GLFWwindow * window,doublexoffset,doubleyoffset)
{
```

```
if(fov >= 1.0f && fov <= 45.0f)
    fov -= yoffset;
if(fov <= 1.0f)
    fov = 1.0f;
if(fov >= 45.0f)
    fov = 45.0f;
}
```

当滑动鼠标滑轮时，yoffset 值代表垂直滑动了多少。当调用 scroll_callback 函数时，我们就更改全局变量 fov 的数值。45.0f 是默认的 fov 值，我们将缩放水平限制在 1.0f 到 45.0f 之间。

现在我们在每个渲染迭代中向 GPU 加载以前定义的投影矩阵，只是这次要使用 fov 变量作为视野：

```
projection = glm::perspective(fov, (GLfloat)WIDTH/(GLfloat)HEIGHT, 0.1f,
    100.0f);
```

最后，别忘记注册这个回调函数：

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

现在，我们就实现了一个可以在 3 维场景中自由移动的相机系统了。

三、实验内容

（一）分析以下程序的原理并上机验证

1.首先调试并运行参考程序 4.1，实现一个基本的相机漫游系统。（具体做法可参考前面着色器的概念讲述部分）。

[参考程序 4.1]：

```
#include <iostream>
#include <cmath>

// GLEW
#define GLEW_STATIC
```

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

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

// Other Libs
#include <SOIL.h>
// GLM Mathematics
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Other includes
#include "Shader.h"


// Function prototypes
void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mode);
void do_movement();

// Window dimensions
const GLuint WIDTH = 800, HEIGHT = 600;

// Camera
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);
bool keys[1024];

// Deltatime
GLfloat deltaTime = 0.0f;      // Time between current frame and last
frame
GLfloat lastFrame = 0.0f;      // Time of last frame

// The MAIN function, from here we start the application and run the game
loop
int main()
{
```

```
    // Init GLFW
    glfwInit();
    // Set all the required options for GLFW
    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);

    // Create a GLFWwindow object that we can use for GLFW's functions
    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "LearnOpenGL",
nullptr, nullptr);
    glfwMakeContextCurrent(window);

    // Set the required callback functions
    glfwSetKeyCallback(window, key_callback);

    // Set this to true so GLEW knows to use a modern approach to
retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

    // Define the viewport dimensions
    glViewport(0, 0, WIDTH, HEIGHT);

    glEnable(GL_DEPTH_TEST);


    // Build and compile our shader program
    Shader ourShader("path/to/shaders/shader.vs",
"path/to/shaders/shader.frag");


    // Set up vertex data (and buffer(s)) and attribute pointers
    GLfloat vertices[] = {
        -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,
         0.5f, -0.5f, -0.5f,  1.0f, 0.0f,
         0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
         0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
```

```
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,

    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,

    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
     0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 1.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,

    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f
};
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)
    };
    GLuint VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

    // Position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
(GLvoid*)0);
    glEnableVertexAttribArray(0);
    // TexCoord attribute
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
(GLvoid*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(2);

    glBindVertexArray(0); // Unbind VAO


    // Load and create a texture
    GLuint texture1;
    GLuint texture2;
    // ====================
    // Texture 1
    // ====================
```

```
    glGenTextures(1, &texture1);
    glBindTexture(GL_TEXTURE_2D, texture1); // All upcoming
GL_TEXTURE_2D operations now have effect on our texture object
    // Set our texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);    // Set te
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Set texture filtering
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Load, create texture and generate mipmaps
    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); // Unbind texture when done, so we
won't accidentily mess up our texture.
    // ===================
    // Texture 2
    // ===================
    glGenTextures(1, &texture2);
    glBindTexture(GL_TEXTURE_2D, texture2);
    // Set our texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Set texture filtering
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Load, create texture and generate mipmaps
    image = SOIL_load_image("awesomeface.png", &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);
```

```
    // Game loop
    while (!glfwWindowShouldClose(window))
    {
        // Calculate deltatime of current frame
        GLfloat currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        // Check if any events have been activiated (key pressed, mouse
moved etc.) and call corresponding response functions
        glfwPollEvents();
        do_movement();

        // Render
        // Clear the colorbuffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


        // Bind Textures using texture units
        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);

        // Activate shader
        ourShader.Use();

        // Camera/View transformation
        glm::mat4 view;
        view = glm::lookAt(cameraPos, cameraPos + cameraFront,
cameraUp);
        // Projection
        glm::mat4 projection;
```

```
        projection = glm::perspective(45.0f, (GLfloat)WIDTH /
(GLfloat)HEIGHT, 0.1f, 100.0f);
        // Get the uniform locations
        GLint modelLoc = glGetUniformLocation(ourShader.Program,
"model");
        GLint viewLoc = glGetUniformLocation(ourShader.Program,
"view");
        GLint projLoc = glGetUniformLocation(ourShader.Program,
"projection");
        // Pass the matrices to the shader
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE,
glm::value_ptr(view));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE,
glm::value_ptr(projection));

        glBindVertexArray(VAO);
        for (GLuint i = 0; i < 10; i++)
        {
            // Calculate the model matrix for each object and pass it
to shader before drawing
            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);

        // Swap the screen buffers
        glfwSwapBuffers(window);
    }
    // Properly de-allocate all resources once they've outlived their
purpose
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
```

```
    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}

// Is called whenever a key is pressed/released via GLFW
void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
    if (key >= 0 && key < 1024)
    {
        if (action == GLFW_PRESS)
            keys[key] = true;
        else if (action == GLFW_RELEASE)
            keys[key] = false;
    }
}

void do_movement()
{
    // Camera controls
    GLfloat cameraSpeed = 5.0f * deltaTime;
    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;
}
```

（二）按要求进行算法设计并调试运行

1. 请根据“相机漫游”所述内容，在程序 4.1 基础上，利用鼠标输入功能，

实现向上下及左右观察的功能。