实验1.2 OpenGL与着色器编程

1. 实验目的
2. 理解OpenGL的原理。
3. 了解和熟悉OpenGL着色语言。
4. 掌握基于OpenGL的C++程序结构。
5. 掌握OpenGL中若干基本二维图形的绘制。
6. 了解顶点着色器的使用。
7. 了解片元着色器的使用。
8. 理论背景
9. OpenGL的含义

OpenGL是一种应用程序编程接口（Application Programming Interface，API），它是一种可以对图形硬件设备特性进行访问的软件库。OpenGL最新的4.3版本包含了超过500个不同的命令，可以用于设置所需的对象、图像和操作，以便开发出交互式的三维计算机图形应用程序。

OpenGL被设计为一个现代化的、硬件无关的接口，因此可以在不考虑计算机操作系统或窗口系统的前提下，在多种不同的图形硬件系统上，或者完全通过软件的方式实现OpenGL的接口。OpenGL自身并不包括任何执行窗口任务或者处理用户输入的函数，OpenGL也没有提供任何用于表达三维物体模型，或者读取图像文件（例如JPEG文件）的操作。这时，需要通过一系列的几何图元（geometric primitive）（包括点、线、三角形）来创建三维空间的物体。

一个用来渲染图像的OpenGL需要执行的主要操作如下所示：

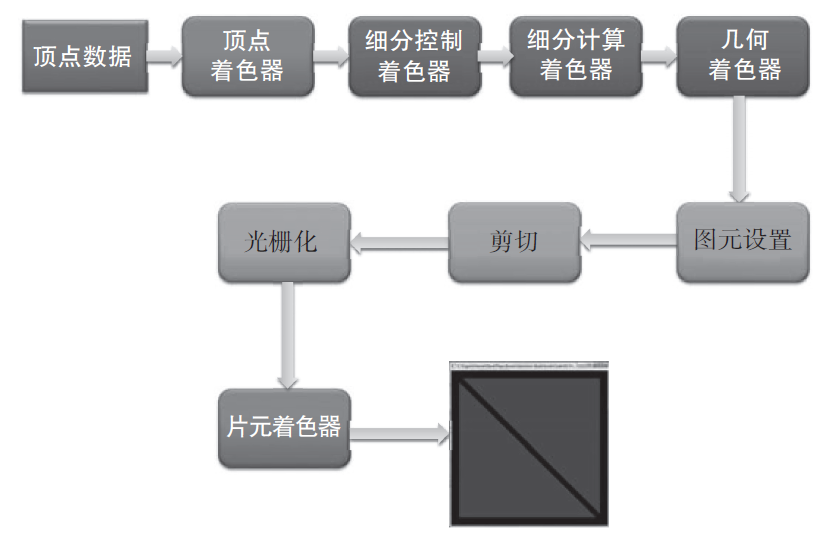
* 1. 从OpenGL的几何图元中设置数据，用于构建形状。

使用不同的着色器（shader）对输入的图元数据执行计算操作，判断它们的位置、颜色，以及其他渲染属性。

* 1. 将输入图元的数学描述转换为与屏幕位置对应的像素偏远（fragment）。这一步也成为光栅化（rasterization）。
  2. 最后针对光栅化过程产生的每个片元，执行片元着色器（fragment shader），从而决定这个片元的最终颜色和位置。
  3. 如果有必要，还需要对每个片元执行一些额外的操作，例如判断片元对应的对象是否可见，或者将片元的颜色与当前屏幕位置的颜色进行融合。

OpenGL另一个最本质的概念叫做着色器，它是图形硬件设备所执行的一类特殊函数。理解着色器最好的方法是把它看作专为图形处理单元（通常称作GPU）编译的一种小型程序。OpenGL在其内部包含了所有的编译器工具，可以直接从着色器源代码创建GPU所需的编译代码并执行。在OpenGL中，会用到四种不同的着色阶段（shader stage）。其中最常用的包括顶点着色器（vertex shader）以及片元着色器（fragment shader），前者用于处理顶点数据，后者用于处理光栅化后的片元数据。所有的OpenGL程序都需要用到这两类着色器。

最终生成的图像包含了屏幕上绘制的所有像素点。像素（pixel）是显示器上最小显示单位。计算机系统将所有的像素保存到帧缓存（frame buffer）。



1. 着色器与OpenGL

OpenGL 实现了通常所说的渲染流程（rendering pipeline），它是一系列数据处理过程，并且将应用程序的数据转换到最终渲染的图像。上图为 OpenGL 4.3 版本的管线。自从 OpenGL 诞生以来，它的渲染流程已经发生了非常大的变化。

OpenGL 首先接收用户提供的几何数据（顶点和几何图元），并且将它输入到一系列着色器阶段中进行处理，包括：顶点着色、细分着色（它本身包含两个着色器），以及最后的几何着色，然后它将被送入光栅化单元（rasterizer）。光栅化单元负责对所有剪切区域（clipping region）内的图元生成片元数据，然后对每个生成的片元都执行一个片元着色器。

对于 OpenGL 应用程序而言着色器扮演了一个最重要的角色。你可完全控制自己需要用到的着色器来实现自己所需的功能。实际上，不需要用到所有的着色阶段，只有顶点着色器和片元着色器是必需的。细分和几何着色器是可选的步骤。

1. 着色器与OpenGL

现代 OpenGL 渲染流程严重依赖着色器来处理传入的数据。在OpenGL 3.0 版本及以前，或者如果你用到了兼容模式（compatibility profile）环境，OpenGL 还包含一个固定功能流程（fixed-function pipeline），它可以在不使用着色器的情况下处理几何与像素数据。从 3.1 版本开始，固定功能流程从核心模式中去除，因此必须使用着色器来完成工作。

对于 OpenGL 而言，会使用 GLSL去编写着色器，也就是 OpenGL Shading Language。虽然 GLSL 是一种专门为图形开发设计的编程语言，但它与C语言非常类似，当然还有一点 C++ 的影子。

1. 着色器的编译

OpenGL 着色器程序的编写与 C 语言等基于编译器的语言非常类似，也就是使用编译器来解析程序，检查是否存在错误，然后将它翻译为目标代码。然后在链接过程中将一系列目标文件合并，并产生最终的可执行程序。下图给出了创建 GLSL 着色器对象并通过链接来生成可执行着色器程序的过程。

对于每个着色器程序，都需要在应用程序中通过下面的步骤进行设置：

* 1. 创建一个着色器对象。
  2. 将着色器源代码编译为对象。
  3. 验证着色器的编译是否成功。

然后需要将所有着色器对象链接为一个着色器程序，包括：

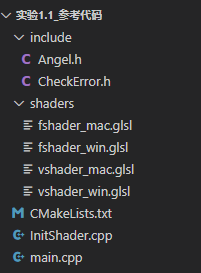
1. 创建一个着色器程序。
2. 将着色器对象关联到着色器程序。
3. 链接着色器程序。
4. 判断着色器的链接过程是否成功完成。
5. 使用着色器来处理顶点和片元。

通常，创建多个着色器对象是因为有可能在不同的程序中复用同一个函数，而 GLSL 程序也是同样的道理。

1. 代码讲解

本次实验内容主要是利用C++和OpenGL着色语言建立一个显示简单二维彩色图形的程序。同学们可以下载实验1.2的代码参照实验进行操作，但建议下载一份实验1.1的代码，然后参考本次实验1.2的实验课的内容逐步添加修改代码，以便更好地理解这些代码的实际含义。

1. OpenGL项目结构
   1. 参照实验1.1的内容，将实验提供代码解压后用自己喜欢的代码编辑器打开。



* 1. include 文件夹:
* Angel.h 主要包含了 GLAD 和 GLFW 的头文件和简单的宏定义
* CheckError.h 定义了输出错误信息的函数
  1. shaders 文件夹：vshader 与 fshader 分别是用 GLSL 编写的顶点着色器和片元着色器，在程序中通过 InitShader() 函数加载，其中文件名中带有“mac”的文件为MacOS下使用的着色器文件。
  2. Initshader.cpp：实现了 InitShader() 函数，是为着色器进入 GPU 的操作专门实现的函数。
  3. main.cpp：项目中的主要逻辑实现文件，包含初始化、绘制、响应控制等功能实现。
  4. CMakeLists是指导CMake生成项目的描述文件， 描述了项目的生成过程，包括基本配置、生成目标、链接库等等。

1. main.cpp核心代码文件

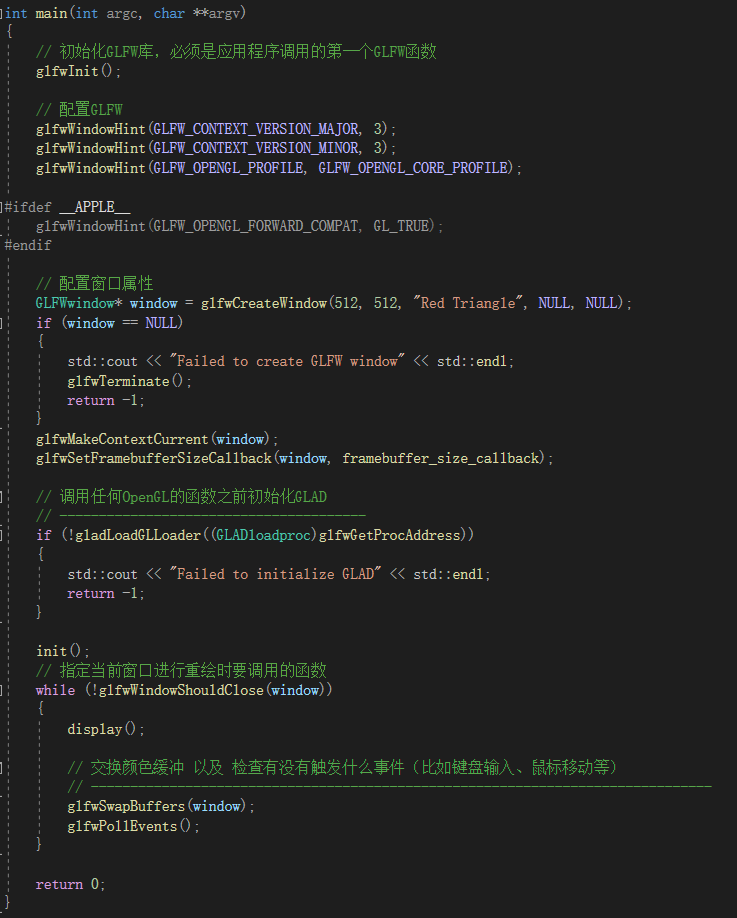
一个OpenGL程序通常会在起始部分，包含必要的头文件，并声明一些全局变量和其他有用的程序结构。程序主体由init()，display()，main()这三个函数组成。

init()函数负责设置程序中需要用到的数据。在本实验中，它负责设置渲染图元时用到的顶点位置信息。然后指定了程序中使用的顶点和片元着色器。最后将应用程序的数据与着色器程序的变量关联起来。

display()函数真正执行了渲染的工作。它负责调用OpenGL函数并渲染需要的内容，几乎所有的display()函数都要完成清除窗口内容、调用OpenGL命令来渲染对象、将最终图像输出到屏幕这三个步骤。

main()函数执行了创建窗口、调用init()以及最终进入时间循环体系的一系列繁重工作。这里会使用到一些以gl开头的函数，这些会是来自第三方库GLAD和GLFW的函数，这些函数的作用是快速完成一些简单功能，并保证OpenGL程序可以运行在不同的操作系统上。

1. 深入main()主函数



glfwInit()初始化GLFW库。glfwInit()必须是应用程序调用的第一个GLFW函数，它会负责设置其他GLFW例程所必需的数据结构。

glfwWindowHint ()用来配置GLFW，第一个参数代表选项的名称，我们可以从很多以GLFW\_开头的枚举值中选择；第二个参数接受一个整型，用来设置这个选项的值。在本例中将主版本号(Major)和次版本号(Minor)都设为3。我们同样明确告诉GLFW我们使用的是核心模式(Core-profile)。明确告诉GLFW我们需要使用核心模式意味着我们只能使用OpenGL功能的一个子集。

glfwCreateWindow ()创建窗口。窗口的宽和高作为它的前两个参数，第三个参数表示这个窗口的名称（标题），第四个和第五个参数可以忽略。

glfwMakeContextCurrent()用来通知GLFW将窗口的上下文设置为当前线程的主上下文。

glfwSetFramebufferSizeCallback()，该函数会调用回调函数framebuffer\_size\_callback()，回调函数中GLFWwindow作为它的第一个参数，以及两个整数表示窗口的新维度。每当窗口改变大小，GLFW会调用这个函数并填充相应的参数供你处理。

gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)，GLAD是用来管理OpenGL的函数指针的，所以在调用任何OpenGL的函数之前需要初始化GLAD。给GLAD传入了用来加载系统相关的OpenGL函数指针地址的函数。GLFW给我们的是glfwGetProcAddress，它根据我们编译的系统定义了正确的函数。

glfwWindowShouldClose()函数在我们每次循环的开始前检查一次GLFW是否被要求退出，如果是的话该函数返回true然后渲染循环便结束，关闭应用程序。

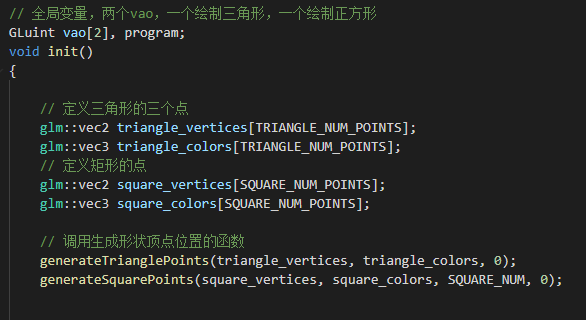
glfwSwapBuffers()函数会交换颜色缓冲（它是一个储存着GLFW窗口每一个像素颜色值的大缓冲），它在这一迭代中被用来绘制，并且将会作为输出显示在屏幕上。

glfwPollEvents()函数检查有没有触发什么事件（比如键盘输入、鼠标移动等）、更新窗口状态，并调用对应的回调函数（可以通过回调方法手动设置）。

1. OpenGL的初始化过程

下面将分析讨论init()函数：

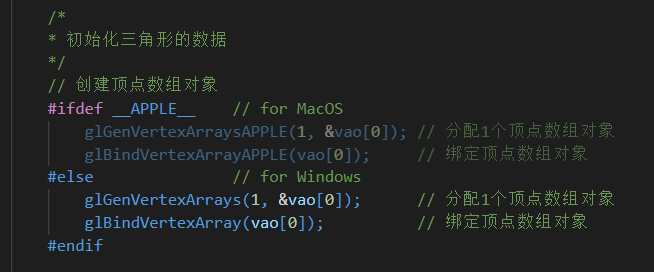
1. 本次实验想要绘制两个形状，一个是三角形，一个是正方形。于是函数开始先创建了两个顶点数组对象vao（vertex array object），这个变量将用于管理顶点缓存对象，一个vao可以管理多个顶点属性（坐标、颜色、法向量等）。然后定义了我们要绘制的三角形与正方形的顶点坐标数组以及对于的顶点颜色数组。其中generateTrianglePoints和generateSquarePoints是代码中生成这些顶点数据的函数。



1. vshader和fshader分别对应shaders文件夹内的顶点与片元着色器文件的路径，我们使用InitShader.cpp内定义好的InitShader函数绑定和编译这两个着色器。函数会返回一个“着色器对象”，我们接着调用glUseProgram函数使用该着色器对象。我们后面会通过缓存对象 buffer 传递数据到渲染管道，而调用glUseProgram之后，渲染管道中会使用到我们指定的这两个着色器进行处理。



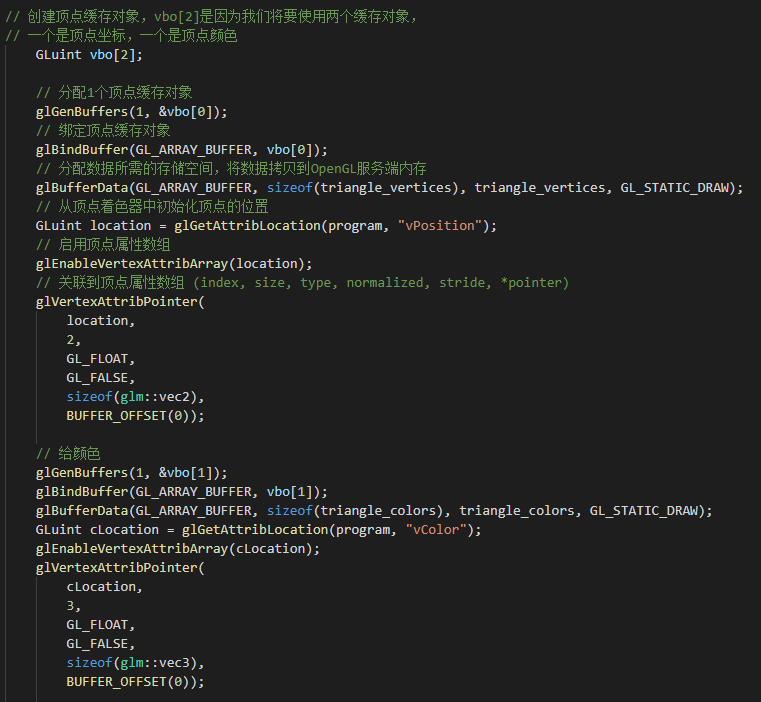
1. 首先我们给三角形的数据进行初始化，使用glGenVertexArrays函数分配一个vao（vertex array object）对象，使用glBindVertexArray函数绑定vao[0]，vao对象将用于管理顶点缓存对象vbo（vertex buffer object），一个vao可以管理多个顶点属性（坐标、颜色、法向量等）



1. 与顶点数组对象vao的创建类似，我们这里定义顶点缓存对象vbo(vertex buffer object)，这个变量是用来真正处理和管理各种顶点数据的，通过vao我们将会向GPU传递数据。使用glGenBuffer函数分配vbo，使用glBindBuffer函数绑定vbo，然后使用glBufferData函数分配一个缓存空间，把我们定义好的 三角形顶点数据triangle\_vertices 传输到缓存对象当中。在glBufferData()函数中，顶点属性数据应设置GL\_ARRAY\_BUFFER，sizeof(vertices)指定了内存分配的大小，由于运行时不做修改所以最后的usage参数为GL\_STATIC\_DRAW。
2. 虽然我们传入了一个顶点坐标数组，但是GPU并不知道要如何使用这个数据，我们还需要告诉GPU如何读取和链接，这里开始会涉及到两个着色器文件的内容。下面先介绍C++代码中的数据操作，然后再介绍着色器文件内代码具体的含义。

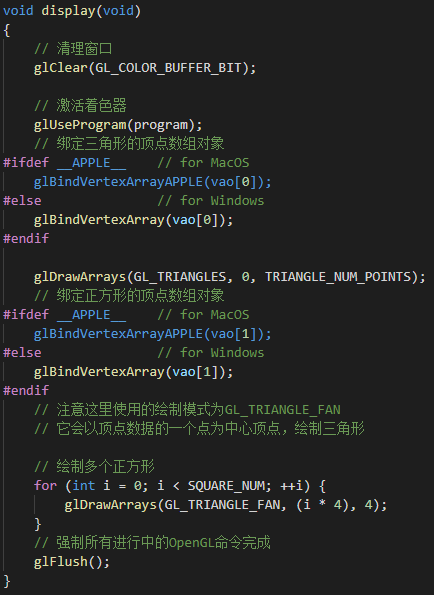
我们希望将triangle\_vertices数组作为顶点坐标传递到渲染管道内，在vshader.glsl中，我们有声明一个in的属性 vPosition ，表示顶点着色器接收的数据，在init函数中，我们使用glGetAttribLocation函数获取这个属性在着色器程序 program 中的位置。

由于顶点属性默认是禁用的，所以获取了这个属性的位置后，我们要使用glEnableVertexAttribArray函数启用该属性，使得着色器中它能够接收数据。之后我们需要手动定义我们传递给着色器的 vertices 数组是如何对应到顶点属性的，也就是和GPU解释这些数据该如何读取，使用的函数为glVertexAttribPointer（文档最后会有函数简介）。同理我们在顶点着色器文件vshader.glsl只也声明了一个 vColor 属性，用来接收顶点颜色数据。



1. 到目前为止，代码都是在glBindVertexArray(vao[0])之后进行的数据初始化，所以这聚代码之后关于vbo的操作，都会指向vao[0]这个顶点数组对象管理，这些数据都是三角形的顶点数据。当我们想初始化正方形的顶点数据时，也需要重复上面三角形的操作，先从glBindVertexArray开始，我们将vao[1]作为正方形数据的顶点数组对象，这里不在赘述，结合代码观看。init函数最后的glClearColor()设置了当前所有使用的清除颜色值，可用来设置背景的颜色。可以参考：<https://learnopengl-cn.github.io/01%20Getting%20started/05%20Shaders/> 进行更加详细的学习。
2. 使用OpenGL进行渲染

下面将分析讨论display()函数。



1. 在渲染前需要利用glClear()函数清除帧缓存的数据。
2. 我们想绘制图像，那么就需要指定着色器，所以需要glUseProgram函数激活我们在init函数中定义的着色器对象program。
3. 先绘制三角形，所以我们先glBindVertexArray绑定三角形数据对应的vao，接着使用glDrawArrays绘制三角形，这函数会使用当前绑定的顶点数组元素来建立一系列的几何图元，实现顶点数据向OpenGL管线的传输。之后重复类似操作绘制正方形。
4. 编写顶点与片元着色器

着色器就是使用OpenGL着色语言（GLSL）编写的一个小型函数。程序可以以字符串的形式传输GLSL着色器到OpenGL，不过为了更容易地使用着色器去进行开发，所有实验都将着色器字符串的内容保存到文件中，并且使用Initshader()读取文件和创建OpenGL着色器程序。下面将深入了解本次实验项目中的顶点着色器与片元着色器。

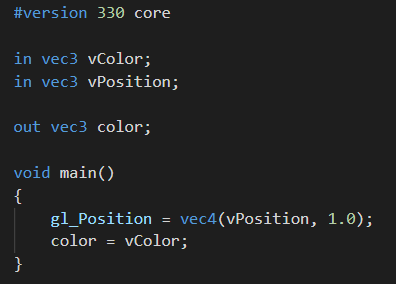
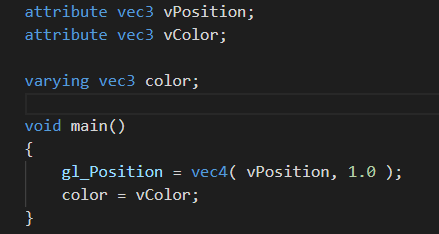
* 1. 顶点着色器vshader.glsl

第一行#version 330 core指定了所用的OpenGL着色语言版本，330 core代表了使用OpenGL 3.3 对应的GLSL语言，core代表使用OpenGL核心模式。每个着色器的第一行都应该设置#version，否则系统会使用110版本（对应OpenGL2.0）。

下一步是分配着色器变量。对于in vec3 vPosition，in字段指定了数据进入着色器的流向，而vPosition变量是一个GLSL中三维浮点数向量。接下来的vColor与vPosition是相似的输入变量。对于out vec3 color，out的限定符代表在顶点着色器中，会把color对应的数值输出。

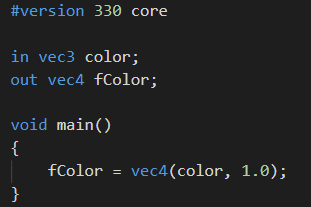
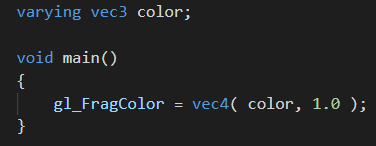
Mac OSX系统会默认使用OpenGL2.1，对应的GLSL版本为120，在shader间传递的变量使用varying字段，传入vshader的使用attribute。

着色器的main()函数实现其主体部分，在OpenGL的所有着色器中，都会有一个main()函数。在这里实现的就是将输入的三维顶点位置转换为四维，最后一位用1.0补齐，并复制到顶点着色器的指定输出变量gl\_Positition中。

* 1. 片元着色器fshader.glsl

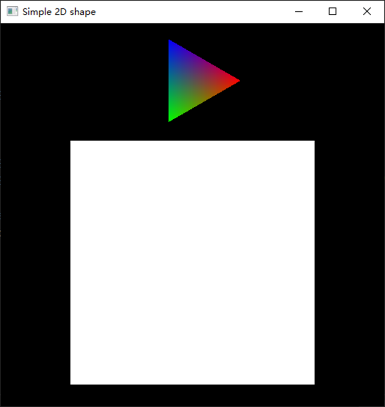
在OpenGL中，还需要一个片元着色器来配合顶点着色器的工作。下面就是片元着色器的内容。

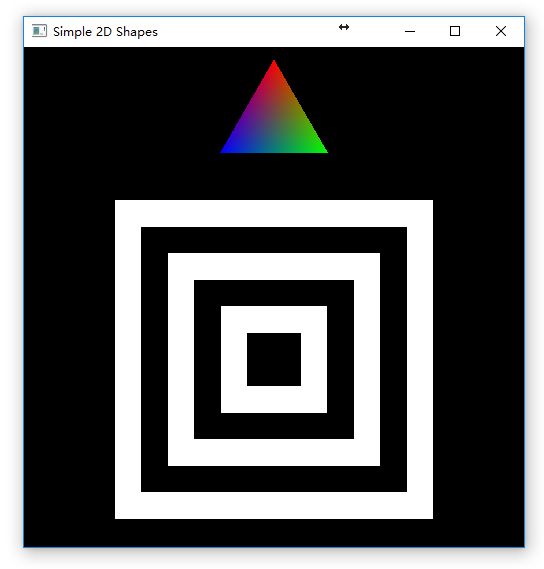
虽然片元着色器与顶点着色器属于两个完全不同类型的着色器，但大部分的代码都很类似。这里的in vec3 color代表了将顶点着色器中输出的color作为该片元着色器输入数据，这样便把两个不同着色阶段的数据连接了起来。最终片元着色器把fColor对应的数值输出，在这里也就是片元所对应的颜色值。

在OpenGL中的颜色是通过RGBA空间表示，因此最后用1.0的完全不透明alpha值，将RGB颜色向量转换为四维RGBA向量。在片元着色器中重点的内容就是设定片元的颜色，而在这里便决定了图元的最终颜色。

1. 课堂练习
2. 执行实验1.2之后，可以得到下图的结果：



1. 通过修改给定代码中生成三角形和生成正方形的函数：generateTrianglePoints和generateSquarePoints，得到下边的效果**【今后的实验中我们会在代码里需要做实验写代码的地方加上@TODO的符号】**：



1. 具体修改内容：
   1. Task-1：在main.cpp中完善generateTrianglePoints函数，生成三角形上的每个点。
   2. Task-2：在main.cpp 中完善generateSquarePoints函数，生成正方形上的每个点。
2. 相关资料

有关着色器和vao等概念的详细介绍可以查看：

<https://learnopengl-cn.github.io/01%20Getting%20started/05%20Shaders/>

1. 函数小结
2. void glGenVertexArrays(GLsizei n, GLuint \*arrays);

返回n个未使用的对象名到数组arrays中，用作顶点数组对象。

1. void glBindVertexArray(GLuint array);

如果输入的变量是由glGenVertexArrays()所返回的，它将创建一个新的顶点数组对象并且与其名称关联。

1. void glGenBuffers(GLsizei n, GLuint \*buffers);

返回那个当前未使用缓存对象名称，并保存到buffers数组中。

1. void glBindBuffer(GLenum target, GLuint buffer);

指定当前激活的缓存对象为target类型的buffer。

1. void glBufferData (GLenum target, GLsizeiptr size, const GLvoid \*data, GLenum usage)

在内存中分配size个存储单元，用于存储数据或者索引。对于顶点的属性数据，target为GL\_ARRAY\_BUFFER，size表示存储数据的总数量。data是为了初始化缓存对象时内存中的指针，不初始化则为NULL。usage用于设置分配数据之后的读取和写入方式。

1. void glBufferSubData (GLenum target, GLintptr offset, GLsizei size, const void \*data);

与glBufferData ()的区别是offset指定了绑定数据的偏移。

1. void glVertexAttribPointer (GLuint index, Glint size, GLsizei size, GLenum type, GLboolean normalized, GLsizei stride const GLvoid \*pointer);

设置index位置对应的数据值。pointer表示缓存对象中从起始位置开始计算的数组数据的偏移值。size表示每个顶点需要更新的分量数目。type指定了数组中每个元素的数据类型。normalized设置顶点数据在存储前是否需要进行归一化。stride是数组中每两个元素之间的大小偏移，为0则是紧密地封装在一起。

1. void glEnableVertexAttribArray (GLuint index);

设置启用与index索引相关联的顶点数组。

1. void glDrawArrays (GLenum mode, Glint first Glsizei count);

使用当前绑定的顶点数组元素来建立一系列的几何图元，起始位置为first，数量为count。mode设置了构建图元的类型。

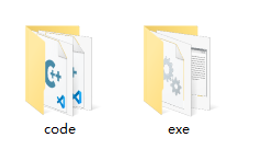
1. void generateTrianglePoints(glm::vec2 vertices[], glm::vec3 colors[], int startVertexIndex)

从startVertexIndex的索引位置开始，在vertices数组存入三角形顶点的坐标信息，在colors数组中存入颜色信息。

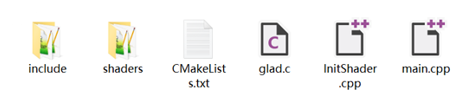
1. void generateSquarePoints(glm::vec2 vertices[], glm::vec3 colors[], int squareNumber, int startVertexIndex)

从startVertexIndex的索引位置开始，在vertices数组存入正方形顶点的坐标信息，在colors数组中存入颜色信息，squareNumber代表正方形的数量。

1. 完成课堂练习后提交**【请各位务必严格按照以下要求执行】**
2. **程序代码**：本次实验提供参考代码，**程序运行窗口标题设为：“学号\_姓名\_实验1.2”**（用你自己的学号和姓名。如果出现乱码，可能是因为代码文件的字符格式问题，实验提供的文件字符编码为utf-8，改成GBK格式就不会出现乱码了）。最终提交代码中与实验内容相关部分必须写注释。
3. **上传格式**：按上述要求完成实验，一并提交源代码压缩包，压缩包名称为“学号\_姓名\_实验1.2”。
   1. 提交文件包括：源代码压缩包，命名格式均为“学号\_姓名\_实验1.2”。
   2. 源代码压缩包内要求有两个文件夹，一个为代码文件夹，命名为“code”，一个为可执行文件夹，命名为“exe”。



* 1. 代码文件夹中只能包含代码和代码需要用到的资源文件（比如纹理图片、模型），其他由编辑器或者编译器创建项目时候生成的文件全部都不要加上，不清楚的同学可以询问助教。



* 1. 可执行文件夹中，只包含可执行文件以及执行所需的动态库文件和资源文件等，要求可以直接点击该程序就可正常执行。



1. **截止时间：2024年09月24日 23:59分**