# **第2章 OpenGL图像管线**

OpenGL是整合软硬件的多平台2D和3D图形API。

在硬件方面，OpenGL提供了一个多级图形管线，可以使用一种名为GLSL的语言进行部分编程。

软件方面，OpenGL的API是用C语言编写的，因此API调用直接兼容 C和C++。使用C++时，程序员编写在CPU上运行的（编译后的）代码并包含OpenGL调用。当一个C++程序包OpenGL调用时，我们将其称为C++/OpenGL应用程序。

C++/OpenGL应用程序的一个重要任务是将程序员的GLSL代码安装到GPU上。

基于C++的图形应用大致如图2.1所示

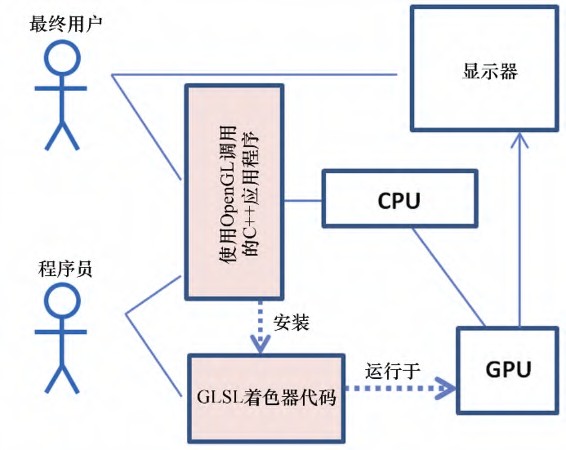


图2.1 基于C++的图形应用概览

在我们后面的代码中，一部分用C++编码，进行OpenGL调用；另一部分是GLSL。C++/OpenGL应用程序、GLSL模块和硬件一起用来生成3D 图形输出。当应用完成之后，最终用户直接与C++应用程序进行交互。

GLSL是与OpenGL兼容的专用着色器语言，因此我们C++/OpenGL应用代码之外，需要用GLSL写着色器代码

## 2.1OpenGL管线

现代3D图形编程会使用管线的概念，在管线中。将3D场景转换成2D图形的过程被分割成许多步骤。

图2.2展示了OpenGL图形管线简化后的概览（并未展示所有阶段， 仅包含我们要学习的主要阶段）。C++/OpenGL应用发送图形数据到顶点着色器，随着管线处理，最终生成在显示器上显示的像素点

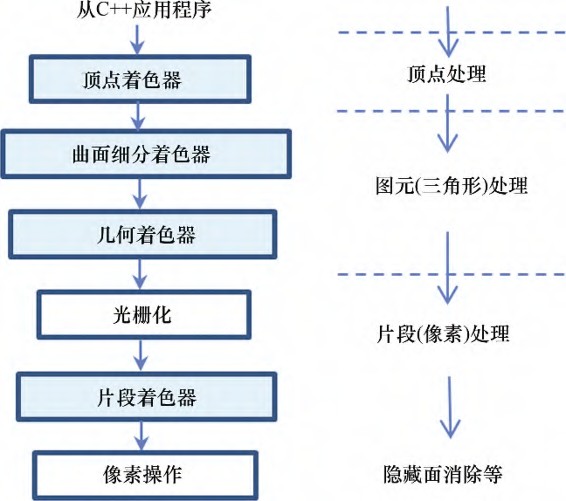


图2.2 OpenGL管线概览

用灰色阴影表示的阶段（顶点着色器、曲面细分着色器、几何着色器、片段着色器）可以用GLSL进行编程。将GLSL程序载入这些着色器阶段也是C++/OpenGL程序的责任之一，其过程如下。

（1）首先使用C++获取GLSL着色器代码，既可以从文件中读取，也可以硬编码在字符串中。

（2）接下来创建OpenGL着色器对象并将GLSL着色器代码加载进着色器对象。

（3）最后，用OpenGL命令编译并连接着色器对象，并将它们安装进GPU。

在实践中，一般至少要提供顶点着色器和片段着色器阶段的GLSL 代码，而曲面细分着色器和几何着色器阶段是可选的。

### 2.1.1 C++/OpenGL应用程序

GLFW库包含了GLFWwindow类，我们可以在其上进行3D场景绘制。在我们尝试编写着色器之前，先写一个简单的C++/OpenGL程序，创建一个GLFWwindow实例并为其设置背景色。这个过程根本用不到着色器！其代码如程序2.1所示。

将会用到的main()函数一样。其中重要的操作有：

（a）初始化GLFW 库；

（b）实例化GLFWwindow；

（c）初始化GLEW库；

（d）调用一次 init()函数；

（e）重复调用display()函数。

#include <GL\glew.h>

#include <GLFW\glfw3.h>

#include <iostream>

using namespace std;

void init(GLFWwindow\* window) { }

void display(GLFWwindow\* window, double currentTime) {

    /\*OpenGL调用glClearColor\*/

    glClearColor(1.0, 0.0, 0.0, 1.0);//指定清除背景时用的颜色值,(1,0,0,1)代表RGB值中的红色（末尾的1表示不透明度）

    glClear(GL\_COLOR\_BUFFER\_BIT);//使用OpenGL调用glClear(GL\_COLOR\_BUFFER\_BIT)，实际使用红色对颜色缓冲区进行填充

}

int main(void) {

    if (!glfwInit()) { exit(EXIT\_FAILURE); }//GLFW库初始化

    /\*WindowHints指定了机器必须与 OpenGL版本4.3兼容（“主版本号”=4，“次版本号”=3）\*/

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

    /\*创建GLFW窗口,指定了窗口的宽、高（以像素为单位)以及窗口顶部的标题

    （这里没有用到的另外两个参数设为NULL，分别用来允许全屏显示以及资源共享。\*/

    GLFWwindow\* window = glfwCreateWindow(600, 600, "Chapter 2 - program 1", NULL, NULL);

    glfwMakeContextCurrent(window);//创建GLFW窗口并不会自动将它与当前 OpenGL上下文关联起来——因此我们需要调用 glfwMakeContextCurrent()

    if (glewInit() != GLEW\_OK) { exit(EXIT\_FAILURE); }//GLEW库初始化

    glfwSwapInterval(1);//glfwSwapInterval()和glfwSwapBuffers()用来开启垂直同步（Vsync）,GLFW窗口默认是双缓冲的

    init(window);

    //渲染循环，用来反复调用display() 方法

    while (!glfwWindowShouldClose(window)) {

        display(window, glfwGetTime());//用glfwGetTime()，它会返回 GLFW初始化之后经过的时间.将当前时间传入了display()调用，这样方便保证动画在不同计算机上以相同速度播放

        glfwSwapBuffers(window);//绘制屏幕

        glfwPollEvents();//处理窗口相关事件（如按键事件）当GLFW探测 到应该关闭窗口的事件（如用户单击了右上角的×）时，循环就会终止

    }

    glfwDestroyWindow(window);//GLFW销毁窗口

    glfwTerminate();//终止运行

    exit(EXIT\_SUCCESS);

}

程序2.1 第一个C++/OpenGL应用程序

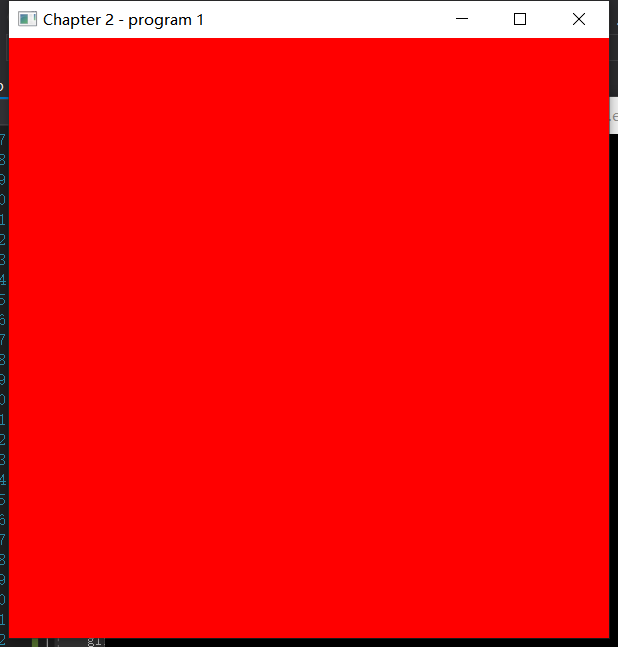


图2.3展示了程序2.1的输出

现在是时候详细看看程序2.1中的OpenGL调用了。首先关注一下这个调用：

glClear(GL\_COLOR\_BUFFER\_BIT);

在这里，调用的OpenGL参考文档中的描述是：

void glClear(GLbitfield mask);

参数中引用了类型为Glbitfield的“GL\_COLOR\_BUFFER\_BIT”。 OpenGL有很多预定义的常量（其中很多是枚举量）。GL\_COLOR\_BUFFER\_BIT引用了包含渲染后像素的颜色缓冲区。OpenGL有多个颜色缓冲区，glClear这个命令会将它们全部清除——用一种被称为“清除色（clear color）”的预定义颜色填充所有缓冲区。

在调用glClear()后紧接着是glClearColor()的调用。 glClearColor()让我们能够指定颜色缓冲区清除后填充的值。这里我们指定了(1,0,0,1)，即RGBA颜色中的红色。

### 2.1.2 顶点着色器和片段着色器

在第一个OpenGL程序中，我们实际上并没有绘制任何东西——仅仅用一种颜色来填充了颜色缓冲区。要真的绘制点什么，我们需要加入顶点着色器和片段着色器。

点、线、三角形。这些简单的东西叫作图元，多数3D模型通常是由许多三角形的图元构成。图元由顶点组成——例如三角形有3个顶点。顶点可以由很多来源产生——从文件读取并由C++/ OpenGL应用载入缓冲区、直接在C++文件中硬编码或者直接在GLSL代码中。

在加载顶点之前，C++/OpenGL应用必须编译并链接合适的GLSL顶点着色器和片段着色器程序，之后将它们载入管线。我们稍后将会看到这些命令。

C++/OpenGL应用同时也负责通知OpenGL构建三角形，通过使用如下OpenGL函数完成：

glDrawArrays(GLenum mode, Glint first, GLsizei count);

**mode**参数是图元的类型——对于三角形我们用GL\_TRIANGLES。

**first**参数表示从哪个顶点开始绘制（通常是顶点0，即第一个顶点）

**count**表示总共要绘制的顶点数。

当调用glDrawArrays()时，管线中的GLSL代码开始执行。现在可以向管线加一些GLSL代码了。

不管它们从何处读入，所有的顶点都会被传入顶点着色器。顶点们会被一个一个地处理，即着色器会对每个顶点执行一次。对拥有很多顶点的大型复杂模型而言，顶点着色器会执行成百上千甚至百万次，这些执行过程通常是并行的。

现在，我们来写一个简单的程序，它仅包含一个顶点，硬编码于顶点着色器中。虽然这不足以让我们画三角形，但是足够画出一个点。为了显示这个点，我们还需要提供片段着色器。为简单起见，我们将这两个着色器程序声明为字符串数组。

#include <GL\glew.h>

#include <GLFW\glfw3.h>

#include <iostream>

using namespace std;

#define numVAOs 1

GLuint renderingProgram;

GLuint vao[numVAOs];

GLuint createShaderProgram() {

    //GLSL顶点着色器代码

    const char \*vshaderSource =

        "#version 430    \n"

        "void main(void) \n"

        "{ gl\_Position = vec4(0.0, 0.0, 0.0, 1.0); }";

    //GLSL片段着色器代码

    const char \*fshaderSource =

        "#version 430    \n"

        "out vec4 color; \n"

        "void main(void) \n"

        "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";

    //基于像素位置决定输出颜色

    /\*

    const char \*fshaderSource =

        "#version 430 \n"

        "out vec4 color; \n"

        "void main(void) \n"

        "{if (gl\_FragCoord.x < 20) color = vec4(1.0, 0.0, 0.0, 1.0); else color = vec4(0.0, 0.0, 1.0, 1.0);}";

    \*/

    GLuint vShader = glCreateShader(GL\_VERTEX\_SHADER); //创建了类型为GL\_VERTEX\_SHADER的着色器

    GLuint fShader = glCreateShader(GL\_FRAGMENT\_SHADER);//创建了类型为GL\_FRAGMENT\_SHADER的着色器

    GLuint vfprogram = glCreateProgram();//创建程序对象

    glShaderSource(vShader, 1, &vshaderSource, NULL);//将GLSL代码从字符串载入空着色器对象中

    glShaderSource(fShader, 1, &fshaderSource, NULL);

    glCompileShader(vShader);//编译各着色器

    glCompileShader(fShader);

    glAttachShader(vfprogram, vShader);//将着色器加入程序对象

    glAttachShader(vfprogram, fShader);

    glLinkProgram(vfprogram);//请求GLSL编译器确保它们的兼容性

    return vfprogram;

}

void init(GLFWwindow\* window) {

    renderingProgram = createShaderProgram();

    glGenVertexArrays(numVAOs, vao);//创建OpenGL要求的VAO

    glBindVertexArray(vao[0]);

}

void display(GLFWwindow\* window, double currentTime) {

    glUseProgram(renderingProgram);//将含有两个已编译着色器 的程序载入OpenGL管线阶段（在GPU上）

    glPointSize(30.0f);//像素范围，注释该句默认为1像素

    glDrawArrays(GL\_POINTS, 0, 1);//启动管线处理过程,原始类型是GL\_POINTS，仅用来显示一个点

}

int main(void) {

    if (!glfwInit()) { exit(EXIT\_FAILURE); }

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

    GLFWwindow\* window = glfwCreateWindow(600, 600, "Chapter 2 - program 2", NULL, NULL);

    glfwMakeContextCurrent(window);

    if (glewInit() != GLEW\_OK) { exit(EXIT\_FAILURE); }

    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {

        display(window, glfwGetTime());

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

    glfwDestroyWindow(window);

    glfwTerminate();

    exit(EXIT\_SUCCESS);

}

程序2.2 着色器画一个点

程序看起来只显示了一个空的窗口（见图2.4）。但仔细观察一下，会发现窗口中央有一个蓝色的点（假设本页印刷精度足够）。 OpenGL中点的默认大小为1像素。

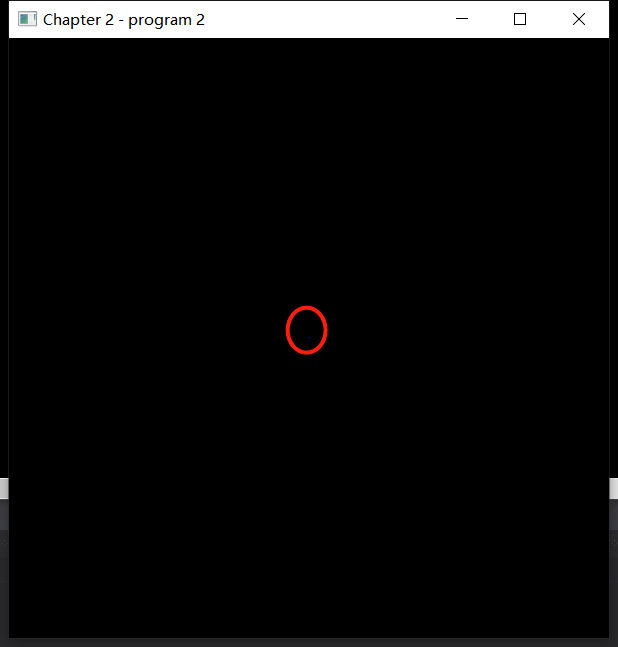


图2.4 程序2.2的输出效果

Gluint：这是由OpenGL提供的“unsigned int”的平台无关简写（许多OpenGL结构体都是整数类型引用）。

init()不再是空函数了，现在它会调用另一个 叫作“createShaderProgram”的函数（我们写的）。 “createShaderProgram”函数先定义了两个字符串vshaderSource和 fshaderSource。之后调用了两次glCreateShader()函数，创建了类型为GL\_VERTEX\_SHADER和GL\_FRAGMENT\_SHADER的两个着色器。

OpenGL创建每个着色器对象（初始值为空）的时候，会返回一个整数ID作为后面引用它的序号——我们的代码将这个ID存入了vShader和fShader变量中。之后，createShaderProgram()调用了glShaderSource()，这个函数用于将GLSL代码从字符串载入空着色器对象中。之后，用

glCompileShader()编译各着色器。

glShaderSource()有4个参数：

参数1：存放着色器的着色器对象

参数2：着色器源代码中的字符串数量

参数3：包含源代码的字符串指针

参数4：最后一个没用到的参数（稍后会在补充章节说明中解释这个参数）。

注意，这两次调用 glCompileShader()时都指明了着色器的源代码字符串数量为“1”，这个参数也会在补充说明中解释。

之后应用程序创建了一个叫作vfProgram的程序对象，并储存指向它的整数ID。OpenGL“程序”对象包含一系列编译过的着色器，这里可以看到使用glCreateProgram()创建程序对象，使用 glAttachShader()将着色器加入程序对象，之后使用glLinkProgram() 来请求GLSL编译器确保它们的兼容性。

如前所见，在init()结束后程序调用了display()。display()函数所做的事情中包含调glUseProgram()，它将含有两个已编译着色器 的程序载入OpenGL管线阶段（在GPU上！）。注意glUseProgram()并没有运行着色器，它只是将着色器加载进硬件。

我们稍后在第4章会看到，一般情况下，这里C++/OpenGL将会准备要发送给管线绘制的模型的顶点集。但是本例中，由于是第一个着色器程序，我们仅仅在顶点着色器中硬编码了一个顶点。因此，本例中 的display()函数接着调用了glDrawArrays()用来启动管线处理过程。 原始类型是GL\_POINTS，仅用来显示一个点。

现在我们来看一下着色器。正如我们所看到的，在C++/OpenGL程序中，它们声明为字符串数组。这是一种笨拙的编程方式，不过在这个超简单的例子中足够了。这个顶点着色器是：

#version 430

void main(void)

{

gl\_Position = vec4(0.0, 0.0, 0.0, 1.0); }

第一行指明了OpenGL版本，这里是4.3。接下来是一个“main”函数（我们后面将会看到，GLSL句法上与C++类似）。所有顶点着色器的主要目标都是将顶点发送给管线（正如之前所说的，它会对每个顶点进行处理）。内置变量gl\_Position用来设置顶点在3D空间的坐标位置，并发送至下一个管线阶段。GLSL数据类型vec4用来存储4元组，适合用来存储坐标，4元组的前3个值分别表示X,Y,Z。第4个值在这里设为1.0（第3章将会学习第4个值的用途）。本例中，顶点坐标硬编码于原点(0,0,0)。

顶点接下来将沿着管线移动到光栅着色器，它们会在这里被转换成像素位置（更精确地说是片段——后面会解释）。最终这些像素（片段）到达片段着色器：

#version 430

out vec4 color; void main(void

)

{

color = vec4(0.0, 0.0, 1.0, 1.0); }

所有片段着色器的目的都是给将要展示的像素赋予RGB颜色。在本例中所指定的输出颜色值(0,0,1)是蓝色（第4个值1.0是不透明度）。

注意这里的“out”标签表明color变量是输出变量。（在顶点着色器中并不是必须给gl\_Position指定“out”标签，因为gl\_Position是预定义的输出变量。）

代码中还有一处我们没有讨论的细节，即init()函数中的最后两行。它们看起来可能有些神秘。我们在第4章中将会看到，当准备将数据集发送给管线时是以缓冲区形式发送的。这些缓冲区最后都会被存入顶点数组对象（Vertex Array Object，VAO）中。 在本例中，我们向顶点着色器中硬编码了一个点，因此我们不需要任何缓冲区。但是，即使应用程序完全没有用到任何缓冲区，OpenGL仍然需要在使用着色器的时候至少有一个创建好的VAO，所以这两行用来创建OpenGL要求的VAO。

最后的问题就是从顶点着色器出来的顶点是如何变成片段着色器中的像素的。回忆一下图2.2中，在顶点处理和像素处理中间存在着光栅化阶段。正是在这个阶段中图元（如点或三角形）转换成了像素集合。OpenGL中默认点的大小为1像素，这就是为什么我们的单点最终渲染成了单个像素。

让我们将下面的命令加入display()函数中，就放在调用 glDrawArrays()之前：

glPointSize(30.0f);

现在，当光栅化阶段从顶点着色器收到顶点时，它会为一个大小是30像素的点设置像素颜色值。输出的结果展示在图2.5中

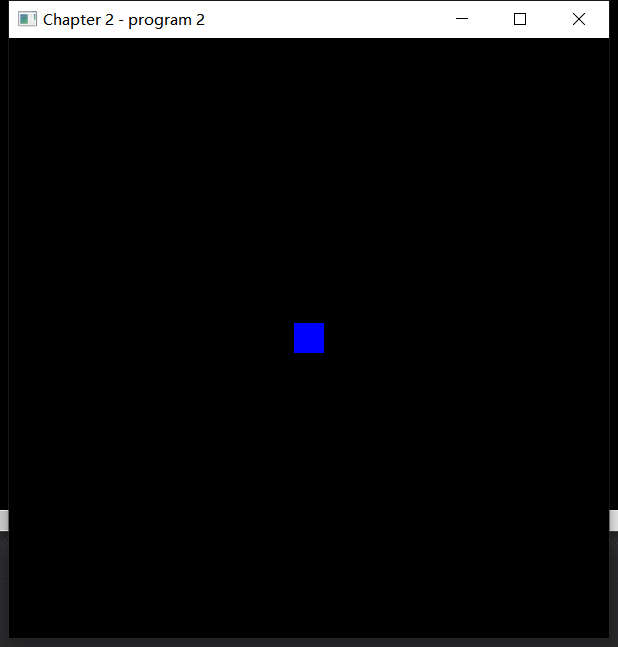


图2.5 改变glPointSize

让我们继续观察剩下的OpenGL管线。

### 2.1.3 曲面细分着色器

我们在第12章中介绍曲面细分。可编程曲面细分阶段是最近加入 OpenGL（在4.0版中）的功能。它提供了一个曲面细分着色器用以生成大量三角形，通常是网格形式。同时也提供一些可以以各种方式操作这些三角形的工具。例如，程序员可能需要以图2.6展示的方式操作一个曲面细分过的三角形网格。

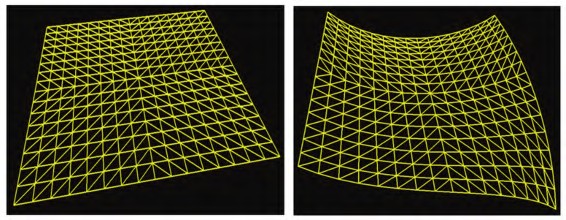


图2.6 曲面细分着色器生成的网格

当在简单形状上需要很多顶点时，曲面细分着色器就能发挥作用了，如在方形区域或曲面上。稍后我们会看到，它在生成复杂地形时也很有用。对于这种情况，有时用GPU中的曲面细分着色器在硬件里生成三角形网格比在C++中生成要高效得多。

### 2.1.4 几何着色器

我们在第13章中介绍了几何着色器阶段。顶点着色器赋予程序员一次操作一个顶点的能力（“按顶点”处理），片段着色器（稍后会看到）允许一次操作一个像素（“按片段”处理），几何着色器赋予了一次操作一个图元的能力（“按图元”处理）。

回顾前文提到最通用的图元是三角形。当我们到达几何着色器阶段时，管线肯定已经完成了将顶点组合为三角形的过程（这个过程叫作图元组装）。接下来几何着色器会让程序员可以同时访问每个三角形的所有顶点。

按图元处理有很多用途，既可以让图元变形，比如拉伸或者缩小，还可以删除一些图元，从而在渲染的物体上产生“洞”——这是一种将简单模型转化为复杂模型的方法。

几何着色器也提供了生成额外图元的方法。这些方法也打开了通过转换简单模型而得到复杂模型的大门。几何着色器有一种有趣的用法，就是在物体上增加表面纹理，如凸起、鳞甚至“毛发”。考虑图 2.7所示的简单环面（本书后面会介绍如何生成它）。环面的表面由上百个三角形构成。如果我们用几何着色器对每个三角形外面增加一个额外的三角形，就会得到如图2.8所示的结果。这个“鳞环面”如果是从C++/OpenGL应用程序那边从零建模生成，代价就大了。



图2.7 环面模型

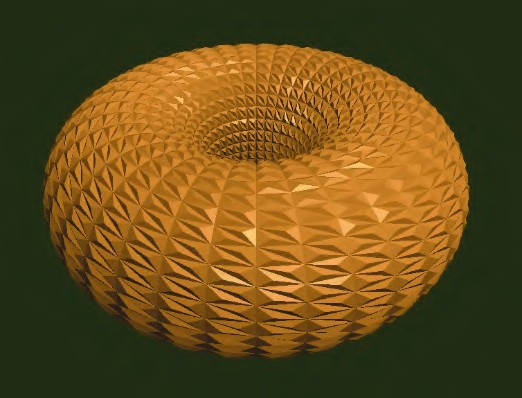


图2.8 几何着色器修改后的环面

在曲面细分阶段已经给程序员同时访问模型中所有顶点的能力后，再提供一个按图元运算的着色器阶段可能看起来有点多余。它们的区别是，曲面细分只在非常少数情况下提供了这个能力——尤其在模型是由曲面细分器生成的三角形网格时。它并没有提供同时访问所有顶点，即任何从C++用缓冲区传来的顶点的能力。

### 2.1.5 光栅化

最终，我们3D世界中的点、三角形、颜色等全都需要展现在一个 2D显示器上。这个2D屏幕由光栅——矩形像素阵列组成。

当3D物体光栅化后，OpenGL将物体中的图元（通常是三角形）转化为片段。片段拥有关于像素的信息。光栅化过程确定了用以显示3个顶点所确定的三角形的所有像素需要绘制的位置。

光栅化过程开始时先对三角形的每对顶点进行插值。插值过程可以通过选项调节。就目前而言，使用图2.9所示的简单的线性插值就够了。原本的3个顶点标记为红色（见彩插）。

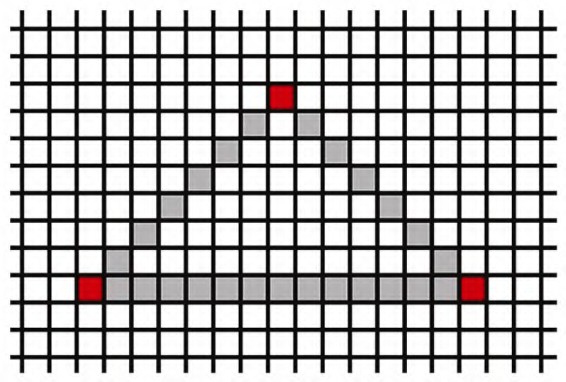


图2.9 光栅化（步骤1）

如果光栅化过程到此为止，那么呈现出的图像将会是线框模型。 呈现线框模型也是OpenGL中的一个选项。通过在display()函数中 glDrawArrays()调用之前添加如下命令：

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

如果2.1.4小节中的环面使用了这行额外代码，它将会看起来如图 2.10所示。

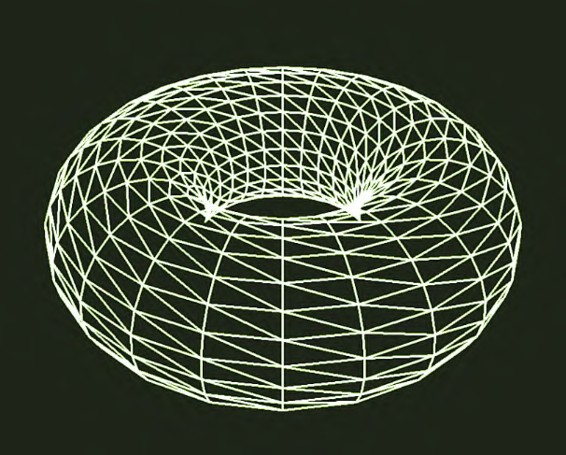


图2.10 使用线框模型渲染的环面

如果我们不加入之前的那一行代码（或者我们在其中使用GL\_FILL 而非GL\_LINE），插值过程将会继续沿着光栅线填充三角形的内部，如图2.11所示。当应用于环面时会产生一个完全光栅化的“实心”环 面，如图2.12（左）所示。请注意，在这种情况下，环面的整体形状和曲率不明显——这是因为我们没有包括任何纹理或照明技术，因此 它看起来是“平”的。图2.12（右）是同样的“平”环面叠加了线框模型。前面图2.7所示的环面包括了照明效果，因此更清晰地显示了环面的形状。我们将在第7章学习照明。

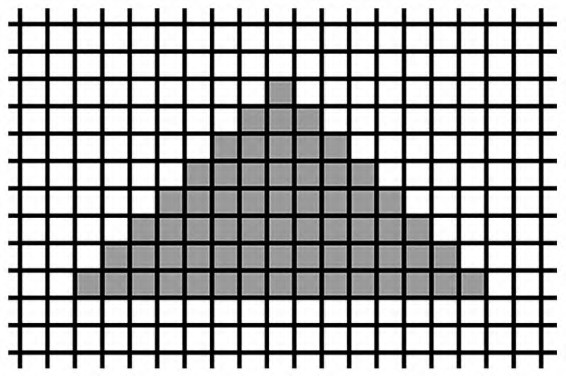


图2.11 完全光栅化的三角形

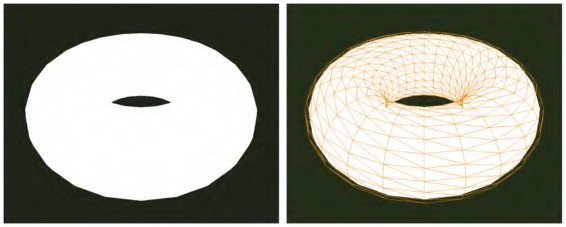


图2.12 环面的完全光栅化图元渲染（左）和使用线框叠加（右）

在本章后面我们将看到，光栅化不仅可以对像素插值。任何顶点着色器输出的变量和片段着色器的输入变量都可以基于对应的像素进行插值。我们将会使用该功能生成平滑的颜色渐变，实现真实光照以及许多其他效果。

### 2.1.6 片段着色器

如前所述，片段着色器用于为光栅化的像素指定颜色。我们已经在程序2.2中看到了片段着色器示例。在程序2.2中，片段着色器仅将输出硬编码为特定值，从而为每个输出的像素赋予相同的颜色。不过 GLSL为我们提供了其他计算颜色的方式，用以表现无穷的创造力。

一个简单的例子就是基于像素位置决定输出颜色。回忆我们在顶点着色器中，顶点的输出坐标使用了预定义变量gl\_Position。在片段着色器中，同样有一个变量让程序员可以访问输入片段的坐标，叫作gl\_FragCoord。我们可以通过修改程序2.2中的片段着色器，让它使用gl\_FragCoord（在本例中通过GLSL属性选择语法引用它的X坐标）基于位置设置每个像素的颜色，如：

#version 430

out vec4 color; void main(void)

{ if (gl\_FragCoord.x < 200) color = vec4(1.0, 0.0, 0.0, 1.0); else

color = vec4(0.0, 0.0, 1.0, 1.0);

}

如果我们像在2.1.2小节末尾那样增大GL\_PointSize，渲染的点的像素颜色将会以坐标变化——坐标小于200时是红色，否则就是蓝色，如图2.13所示（我没做出来）。

    //基于像素位置决定输出颜色

    const char \*fshaderSource =

        "#version 430 \n"

        "out vec4 color; \n"

        "void main(void) \n"

        "{if (gl\_FragCoord.x < 20) color = vec4(1.0, 0.0, 0.0, 1.0); else color = vec4(0.0, 0.0, 1.0, 1.0);}";

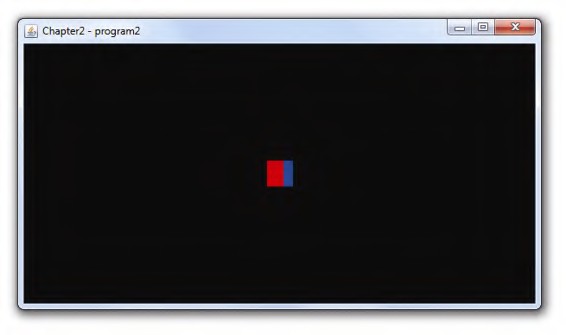


图2.13 片段着色器颜色变化

### 2.1.7 像素操作

当我们在display()方法中使用glDrawArrays()命令绘制场景中的物体时，我们通常期望前面的物体挡住后面的物体。这也可以推广到物体自身，我们通常期望看到物体的正面对着我们，而不是背对我们。

为了实现这个效果，我们需要隐藏面消除（Hidden Surface Removal，HSR）。基于场景需要的不同效果OpenGL可以进行一系列不同的HSR操作。虽然这个阶段不可编程，但是理解它的工作原理是非常重要的。我们不仅需要正确地配置它，之后还需要在给场景添加阴影时对它进行进一步操作。

OpenGL通过精巧地协调两个缓冲区完成隐藏面消除：颜色缓冲区（我们之前讨论过）和深度缓冲区（也叫作Z缓冲、Z-buffer）。这两个缓冲区都和光栅的大小相同——即对于屏幕上每个像素，在两个缓冲区都各有一个对应条目。

当绘制场景中的各种对象时，片段着色器会生成像素颜色。像素颜色会存放在颜色缓冲区中——颜色缓冲区最终会被写入屏幕。当多个对象占据颜色缓冲区中的相同像素时，必须根据哪个对象最接近观察者来确定保留哪个像素颜色。

隐藏面消除按照如下步骤完成。

（1）在每个场景渲染前，深度缓冲区全部初始化为表示最大深度的值。

（2）当像素颜色由片段着色器输出时，计算它到观察者的距离。

（3）如果距离小于深度缓冲区存储的值（对当前像素），那么用当前像素颜色替换颜色缓冲区中的颜色，同时用当前距离替换深度缓冲区中的值，否则抛弃当前像素。

这个过程叫作Z-Buffer算法，如图2.14所示。

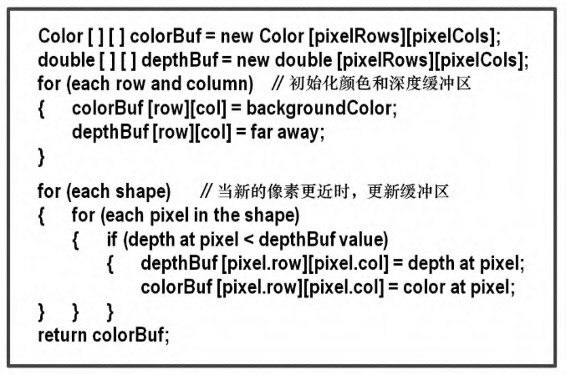


图2.14 Z-buffer算法

## 2.2检测OpenGL和GLSL错误

编译和运行GLSL代码与普通编码的过程不同，GLSL编译发生在 C++运行时。另外一个复杂的地方是GLSL代码并没有运行在CPU中（它运行在GPU上），因此操作系统不总是能够捕获OpenGL运行时的错误。 以上这些使得调试变得很困难，因为常常很难检测着色器是否失败， 以及为什么失败。

程序2.3展示了用于捕获和显示GLSL错误的模块。其中GLSL函数 glGetShaderiv()和glGetProgramiv()用于提供有关编译过的GLSL着色器和程序的信息。还有之前程序2.2中的createShaderProgram()函数，不过加入了错误检测的调用。程序2.3包含如下3个实用程序。

checkOpenGLError：检查OpenGL错误标志，即是否发生OpenGL错 误。 printShaderLog：当GLSL编译失败时，显示OpenGL日志内容。 printProgramLog：当GLSL链接失败时，显示OpenGL日志内容。

checkOpenGLError()既用于检测GLSL编译错误，又用于检测 OpenGL运行时的错误，因此我们强烈建议在整个C++/OpenGL应用程序开发过程中使用它。例如，在之前的程序2.2中，对于 glCompileShader()和glLinkProgram()的调用很容易用程序2.3的代码进行加强，来确认所有的拼写错误和编译错误都能被捕获到，同时报告其原因。

用这些工具很重要的另一个原因是，GLSL错误并不会导致C++程序崩溃。因此，除非程序员通过步进找到错误发生的点，否则调试会非常困难。

#include <GL\glew.h>

#include <GLFW\glfw3.h>

#include <iostream>

using namespace std;

#define numVAOs 1

GLuint renderingProgram;

GLuint vao[numVAOs];

void printShaderLog(GLuint shader) {//当GLSL编译失败时，显示OpenGL日志内容

    int len = 0;

    int chWrittn = 0;

    char \*log;

    glGetShaderiv(shader, GL\_INFO\_LOG\_LENGTH, &len);

    if (len > 0) {

        log = (char \*)malloc(len);

        glGetShaderInfoLog(shader, len, &chWrittn, log);

        cout << "Shader Info Log: " << log << endl;

        free(log);

    }

}

void printProgramLog(int prog) {//当GLSL链接失败时，显示OpenGL日志内容

    int len = 0;

    int chWrittn = 0;

    char \*log;

    glGetProgramiv(prog, GL\_INFO\_LOG\_LENGTH, &len);

    if (len > 0) {

        log = (char \*)malloc(len);

        glGetProgramInfoLog(prog, len, &chWrittn, log);

        cout << "Program Info Log: " << log << endl;

        free(log);

    }

}

/\*

检查OpenGL错误标志，即是否发生OpenGL错误

checkOpenGLError()既用于检测GLSL编译错误，又用于检测 OpenGL运行时的错误，因此我们强烈建议在整个C++/OpenGL应用程序开发过程中使用它

\*/

bool checkOpenGLError() {//

    bool foundError = false;

    int glErr = glGetError();

    while (glErr != GL\_NO\_ERROR) {

        cout << "glError: " << glErr << endl;

        foundError = true;

        glErr = glGetError();

    }

    return foundError;

}

GLuint createShaderProgram() {

    GLint vertCompiled;

    GLint fragCompiled;

    GLint linked;

    const char \*vshaderSource =

        "#version 430    \n"

        "void main(void) \n"

        "{ gl\_Position = vec4(0.0, 0.0, 0.0, 1.0); }";

    const char \*fshaderSource =

        "#version 430    \n"

        "out vec4 color; \n"

        "void main(void) \n"

        "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";

    GLuint vShader = glCreateShader(GL\_VERTEX\_SHADER);

    GLuint fShader = glCreateShader(GL\_FRAGMENT\_SHADER);

    GLuint vfprogram = glCreateProgram();

    glShaderSource(vShader, 1, &vshaderSource, NULL);

    glShaderSource(fShader, 1, &fshaderSource, NULL);

    // 捕获编译着色器时的错误

    glCompileShader(vShader);

    checkOpenGLError();

    glGetShaderiv(vShader, GL\_COMPILE\_STATUS, &vertCompiled);

    if (vertCompiled == 1) {

        cout << "vertex compilation success" << endl;

    }

    else {

        cout << "vertex compilation failed" << endl;

        printShaderLog(vShader);

    }

    glCompileShader(fShader);

    checkOpenGLError();

    glGetShaderiv(fShader, GL\_COMPILE\_STATUS, &fragCompiled);//提供有关编译过的GLSL着色器和程序的信息

    if (fragCompiled == 1) {

        cout << "fragment compilation success" << endl;

    }

    else {

        cout << "fragment compilation failed" << endl;

        printShaderLog(fShader);

    }

    // 捕获链接着色器时的错误

    glAttachShader(vfprogram, vShader);

    glAttachShader(vfprogram, fShader);

    glLinkProgram(vfprogram);

    glLinkProgram(vfprogram);

    checkOpenGLError();

    glGetProgramiv(vfprogram, GL\_LINK\_STATUS, &linked);//提供有关编译过的GLSL着色器和程序的信息

    if (linked == 1) {

        cout << "linking succeeded" << endl;

    }

    else {

        cout << "linking failed" << endl;

        printProgramLog(vfprogram);

    }

    return vfprogram;

}

void init(GLFWwindow\* window) {

    renderingProgram = createShaderProgram();

    glGenVertexArrays(numVAOs, vao);

    glBindVertexArray(vao[0]);

}

void display(GLFWwindow\* window, double currentTime) {

    glUseProgram(renderingProgram);

    glPointSize(30.0f);

    glDrawArrays(GL\_POINTS, 0, 1);

}

int main(void) {

    if (!glfwInit()) { exit(EXIT\_FAILURE); }

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

    GLFWwindow\* window = glfwCreateWindow(600, 600, "Chapter 2 - program 3", NULL, NULL);

    glfwMakeContextCurrent(window);

    if (glewInit() != GLEW\_OK) { exit(EXIT\_FAILURE); }

    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {

        display(window, glfwGetTime());

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

    glfwDestroyWindow(window);

    glfwTerminate();

    exit(EXIT\_SUCCESS);

}

}

程序2.3 用以捕获GLSL错误的模块

还有一些其他用于推测着色器代码运行时错误成因的技巧。着色器运行时错误的常见结果是输出屏幕上完全空白，根本没有输出。即使是着色器中的一个小拼写错误也可能导致这种结果，这样就很难断定是哪个管线阶段发生了错误。没有任何输出的情况下，找到错误的 成因就像大海捞针。

其中一种有用的技巧就是暂时将片段着色器换成程序2.2中的片段着色器。回忆程序2.2中，片段着色器仅输出一个特定颜色——例如蓝色。如果后来的输出中的几何形状正确（但是全是蓝色），那么顶点着色器应该是正确的，错误应该发生在片段着色器。如果输出的仍然是空白屏幕，那错误很可能发生在管线的更早期，譬如顶点着色器。

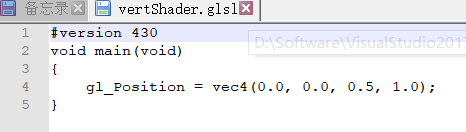
在附录C中，我们展示了另一种有用的调试工具，叫作Nsight，适用于特定型号Nvidia显卡的机器。

## 2.3 从文件读取GLSL源代码

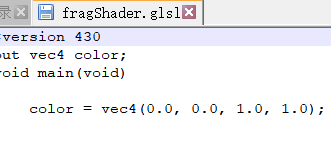
到此为止，GLSL着色器代码已经内联存储在字符串中了。当程序变得更复杂时，这么做就不实际了。我们应当将我们的着色器代码存在文件中并读入它们。

读入文本文件是基础C++技能，我们在此就不赘述了。但是，为实用起见，用于读取着色器的代码readShaderSource()在程序2.4中提供。它读取着色器文本文件并返回一个字符串数组，其中每个字符串是文件中的一行文本。然后根据读入的行数确定该数组的大小。注意，createShaderProgram()在这里替换了程序2.2中的版本。在本例中，顶点着色器和片段着色器代码现在分别放在文本文件 “vertShader.glsl”和“fragShader.glsl”中。

vertShader.glsl



fragShader.glsl



#include <GL\glew.h>

#include <GLFW\glfw3.h>

#include <iostream>

#include <string>

#include <iostream>

#include <fstream>

using namespace std;

#define numVAOs 1

GLuint renderingProgram;

GLuint vao[numVAOs];

string readFile(const char \*filePath) {

    string content;

    ifstream fileStream(filePath, ios::in);

    string line = "";

    while (!fileStream.eof()) {

        getline(fileStream, line);

        content.append(line + "\n");

    }

    fileStream.close();

    return content;

}

GLuint createShaderProgram() {

    GLuint vShader = glCreateShader(GL\_VERTEX\_SHADER);

    GLuint fShader = glCreateShader(GL\_FRAGMENT\_SHADER);

    GLuint vfprogram = glCreateProgram();

    string vertShaderStr = readFile("vertShader.glsl");

    string fragShaderStr = readFile("fragShader.glsl");

    const char \*vertShaderSrc = vertShaderStr.c\_str();

    const char \*fragShaderSrc = fragShaderStr.c\_str();

    glShaderSource(vShader, 1, &vertShaderSrc, NULL);

    glShaderSource(fShader, 1, &fragShaderSrc, NULL);

    glCompileShader(vShader);

    glCompileShader(fShader);

    glAttachShader(vfprogram, vShader);

    glAttachShader(vfprogram, fShader);

    glLinkProgram(vfprogram);

    return vfprogram;

}

void init(GLFWwindow\* window) {

    renderingProgram = createShaderProgram();

    glGenVertexArrays(numVAOs, vao);

    glBindVertexArray(vao[0]);

}

void display(GLFWwindow\* window, double currentTime) {

    glUseProgram(renderingProgram);

    glPointSize(30.0f);

    glDrawArrays(GL\_POINTS, 0, 1);

}

int main(void) {

    if (!glfwInit()) { exit(EXIT\_FAILURE); }

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

    GLFWwindow\* window = glfwCreateWindow(600, 600, "Chapter 2 - program 4", NULL, NULL);

    glfwMakeContextCurrent(window);

    if (glewInit() != GLEW\_OK) { exit(EXIT\_FAILURE); }

    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {

        display(window, glfwGetTime());

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

    glfwDestroyWindow(window);

    glfwTerminate();

    exit(EXIT\_SUCCESS);

}

程序2.4 从文件读取GLSL源文件

## 2.4 从顶点构建对象

最终我们想要绘制的不止是一个单独的点，而是想要绘制由很多顶点组成的对象。本书的大部分章节将会致力于这一主题。现在我们从一个简单的例子开始——我们将会定义3个顶点，并用它们绘制一个三角形，如图2.15所示。

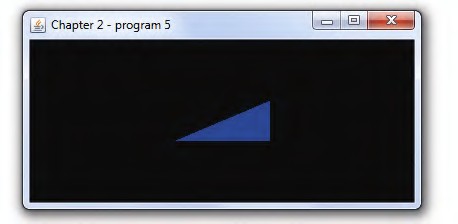
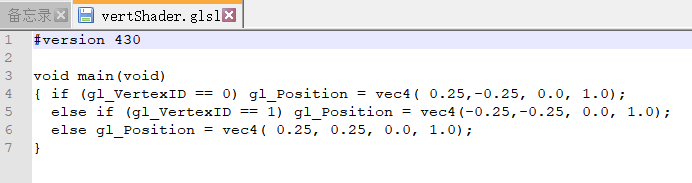


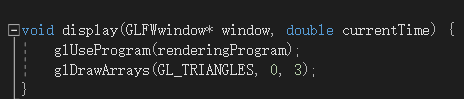
图2.15 绘制简单三角形

我们可以通过对程序2.2（事实上是从文件读入着色器的程序 2.4）进行两个小改动来实现绘制三角形：

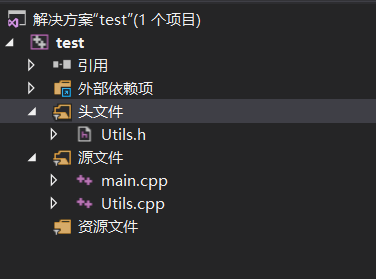
（a）修改顶点着色器，以便将3个不同的点输出到后续的管线阶段；



（b）修改glDrawArrays() 调用，指定3个顶点。



在C++/OpenGL应用程序中［特别是在glDrawArrays()调用中］我们指定了GL\_TRIANGLES（而非GL\_POINTS），同时也指定了管线中有3个顶点。这样顶点着色器会在每个迭代运行3遍，内置变量 gl\_VertexID会自增（初始值为0）。通过检测gl\_VertexID的值，着色器设计为可以在每次运行时输出不同的点。前面说到这3个点之后会经过光栅化阶段，生成一个填充过的三角形。程序的改动显示在程序2.5 中（余下的代码与之前在程序2.4中的相同）。



记得添加这个Utils.cpp和h文件

#include <GL\glew.h>

#include <GLFW\glfw3.h>

#include <iostream>

#include <string>

#include <iostream>

#include <fstream>

#include "Utils.h"

using namespace std;

#define numVAOs 1

GLuint renderingProgram;

GLuint vao[numVAOs];

void init(GLFWwindow\* window) {

    renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");

    glGenVertexArrays(numVAOs, vao);

    glBindVertexArray(vao[0]);

}

void display(GLFWwindow\* window, double currentTime) {

    glUseProgram(renderingProgram);

    glDrawArrays(GL\_TRIANGLES, 0, 3);//改动在这里

}

int main(void) {

    if (!glfwInit()) { exit(EXIT\_FAILURE); }

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

    GLFWwindow\* window = glfwCreateWindow(600, 600, "Chapter 2 - program 5", NULL, NULL);

    glfwMakeContextCurrent(window);

    if (glewInit() != GLEW\_OK) { exit(EXIT\_FAILURE); }

    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {

        display(window, glfwGetTime());

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

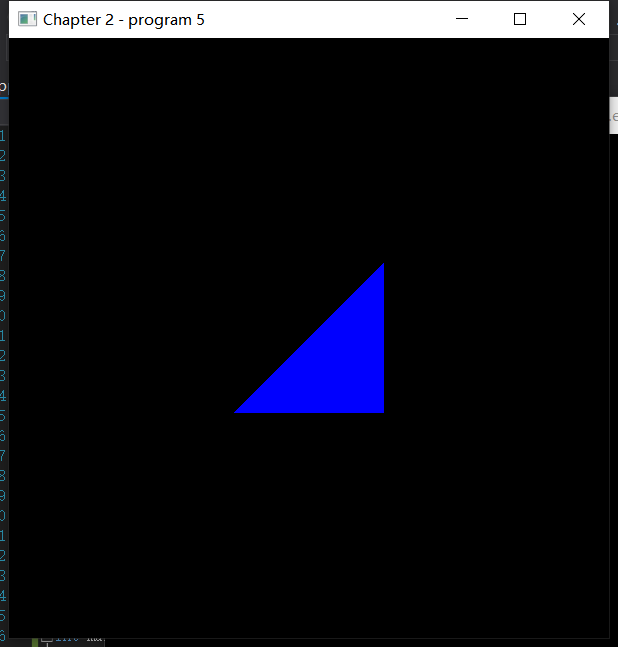
    glfwDestroyWindow(window);

    glfwTerminate();

    exit(EXIT\_SUCCESS);

}

程序2.5 绘制三角形



## 2.5 场景动画

本书中的很多技术可以用于动画。当场景中的物体移动或改变时，场景会被重复渲染以实时反映这些改动。

回顾2.1.1小节中，我们构建的main()函数只调用了init()一次， 之后就重复调用display()。因此虽然前面所有的例子看起来都是绘制的场景，但实际上main()函数中的循环会让它们一次又一次地绘制。

因此，main()函数的结构已经可以支持动画了。我们只需要设计 display()函数来随时间改变绘制的东西。场景的每一次绘制都叫作一帧，调用display()的频率叫作帧率。在程序逻辑中移动的速率可以通过自前一帧到目前经过的时间来控制（这就是为什么我们会将 “currentTime”作为display()函数的参数）。

程序2.6中展示了动画示例。我们使用了程序2.5中的三角形，并给它加入了先向右，再向左，往复移动的动画。在本例中，我们不考虑经过的时间，因此三角形的移动或快或慢，基于运行计算机的处理速度。在未来的示例中，我们将会使用经过的时间来确保无论在什么配置的计算机上运行，动画都保持以同样的速度播放。

在程序2.6中，程序的display()方法维持一个变量用于偏移三角形的X轴位置。每当display()调用时，它的值都会改变（因此每帧都不同）。同时每当它到达1.0或者−1.0时，都会改变方向。在x中的值会被复制到顶点着色器的“offset”变量中。执行这个复制的机制叫作Uniform变量（统一变量），稍后我们会在第4章中学习它。目前不必了解统一变量的细节。现在，只需要注意C++/OpenGL应用程序先调用glGetUniformLocation()获取指向“offset”变量的指针，之后调用glProgramUniform1f()将x的值复制给offset。之后顶点着色器会将offset加给所绘制三角形的X坐标。注意，每次调用display()时背景都会被清除，以避免三角形移动时留下一串轨迹。图2.16展示了3个时间点显示的图像（当然，书中的静态图是无法展示移动的）。

#include <GL\glew.h>

#include <GLFW\glfw3.h>

#include <iostream>

#include <string>

#include <iostream>

#include <fstream>

#include "Utils.h"

using namespace std;

#define numVAOs 1

GLuint renderingProgram;

GLuint vao[numVAOs];

GLuint offsetLoc;

float x = 0.0f;

float inc = 0.01f;

void init(GLFWwindow\* window) {

    renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");

    glGenVertexArrays(numVAOs, vao);

    glBindVertexArray(vao[0]);

}

void display(GLFWwindow\* window, double currentTime) {

    glClear(GL\_DEPTH\_BUFFER\_BIT);

    glClearColor(0.0, 0.0, 0.0, 1.0);

    glClear(GL\_COLOR\_BUFFER\_BIT);

    glUseProgram(renderingProgram);

    x += inc;

    if (x > 1.0f) inc = -0.01f;

    if (x < -1.0f) inc = 0.01f;

    offsetLoc = glGetUniformLocation(renderingProgram, "offset");//获取指向“offset”变量的指针

    glProgramUniform1f(renderingProgram, offsetLoc, x);//将x的值复制给offset

    glDrawArrays(GL\_TRIANGLES, 0, 3);

}

int main(void) {

    if (!glfwInit()) { exit(EXIT\_FAILURE); }

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

    glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

    GLFWwindow\* window = glfwCreateWindow(600, 600, "Chapter 2 - program 6", NULL, NULL);

    glfwMakeContextCurrent(window);

    if (glewInit() != GLEW\_OK) { exit(EXIT\_FAILURE); }

    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {

        display(window, glfwGetTime());

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

    glfwDestroyWindow(window);

    glfwTerminate();

    exit(EXIT\_SUCCESS);

}

程序2.6 简单动画示例

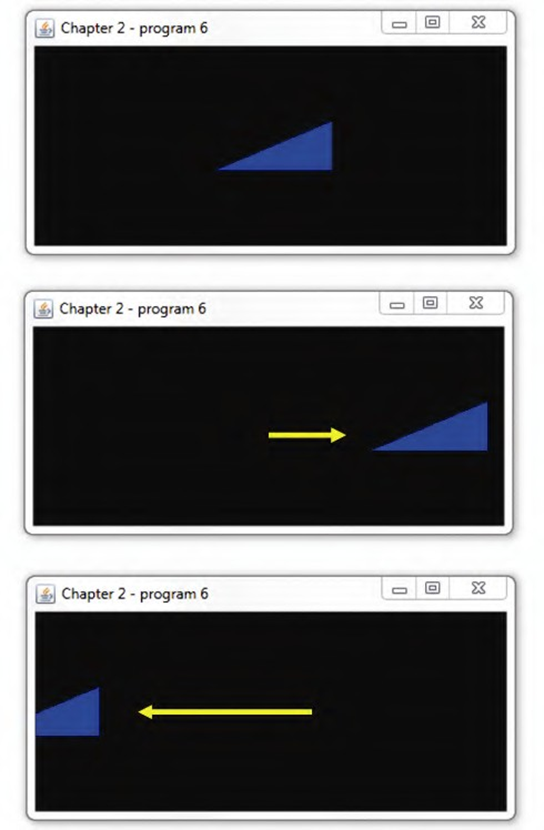


图2.16 移动的三角形动画

注意，除了添加三角形动画代码之外，我们还在display()函数的开头添加了这行代码：

glClear(GL\_DEPTH\_BUFFER\_BIT);

虽然在本例中并不是必需的，我们仍然把它加在这里，同时它会在之后的大多数应用程序中存在。回忆2.1.7小节中讨论的，隐藏面消除需要同时用到颜色缓冲区和深度缓冲区。当我们后面渐渐地开始绘制更复杂的3D场景时，每帧初始化（清除）深度缓冲区就是必要的，尤其是对于动画场景，要确保深度对比不会受旧的深度数据影响。从前面的例子中可以明显看出，清除深度缓冲区的命令与清除颜色缓冲 区的命令基本相同。

## 2.6 C++代码文件结构

目前为止，我们的所有C++/OpenGL应用程序代码都放在同一个叫作“main.cpp”的文件中，GLSL着色器代码放在“vertShader.glsl” 和“fragShader.glsl”文件中。我们承认在main.cpp中塞进很多应用代码不是最佳实践，但我们在本书中采用这个约定，以便于在每个例子中，哪个文件包含这个例子中主要的C++/OpenGL代码这件事都很清楚。在本教材中，主要的C++/OpenGL文件总是叫作“main.cpp”。在 实践中，应用程序当然应该模块化，以适当对应应用的各功能。

但是，当我们继续学习时，我们会遇到一些情况。在这些情况下，我们会创建一些实用的模块，并在不同的应用程序中使用。当时机适当，我们会将这些模块分离到单独的文件中以便重用。例如，稍后我们会定义一个Sphere类。这个类会在很多例子中用到，因此它会分到它自己的文件（Sphere.cpp和Sphere.h）中。

相似地，当我们遇到需要重用函数的时候，我们会把它们放进 “Utils.cpp”（与“Utils.h”关联）。我们已经看到好几个适合放 进“Utils.cpp”的函数了：2.2节中描述的错误检测模块和2.3节中描述的用来读入GLSL着色器的函数。后者非常适合重载，如 “createShaderProgram()”可以对应用中所有可能的管线着色器组合进行定义：

* GLuint Utils::createShaderProgram(const char \*vp, const char

\*fp)

* GLuint Utils::createShaderProgram(const char \*vp, const char

\*gp, const char \*fp)

* GLuint Utils::createShaderProgram(const char \*vp, const char

\*tCS, const char\* tES, const char \*fp)

* GLuint Utils::createShaderProgram(const char \*vp, const char

\*tCS, const char\* tES, const char \*gp, const char \*fp)

以上列出的第一个条目支持仅使用了顶点着色器和片段着色器的程序。第二个支持使用了顶点着色器、几何着色器和片段着色器的情况。第三个支持用了顶点着色器、曲面细分着色器和片段着色器的情况。第四个支持用了顶点着色器、曲面细分着色器、几何着色器和片 段着色器的情况。每个条目中，接受的参数都包含着色器代码的GLSL 文件路径。例如，如下调用使用了其中一个重载函数，以编译并链接包含顶点着色器和片段着色器的管线。编译链接后的程序被放在变量“renderingProgram”中：

renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");

这些createShaderProgram()实现都可以在随书附赠的配套资源中找到（在“Utils.cpp”文件中），同时它们都包含了2.2节中的错误处理。它们并没有什么新内容，只是用这种方式组织以便使用。随着我们继续向前推进本书，会有更多相似的函数加入Utils.cpp。我们强烈鼓励读者阅读配套资源中的Utils.cpp文件，甚至有需要时可在其中加入函数。配套资源中的程序是根据学习本书的方法构建的，因此了解它们的结构应当有助加强自己对书中内容的理解。

我们对于在“Utils.cpp”文件中的函数，都以静态函数进行实现，因此不需要实例化Utils类。基于正在开发的系统架构，读者可能 会倾向于使用实例方法甚至独立函数实现它们。

我们所有的着色器文件都使用“.glsl”后缀

# **补充说明**

在本章中，还有很多我们没有讨论到的OpenGL管线细节。我们略过了许多内部阶段，同时完全省略了纹理的处理。我们在本章的目标是，对后面要用来编码的框架有尽可能简单的整体印象。当我们继续学习时，会学到更多的细节。同时我们也推迟了展示曲面细分着色器和几何着色器的代码。在之后的章节中，我们会构建一套完整的系统，来展现如何为每个阶段编写实际的着色器。

对于如何组织场景动画代码，尤其是线程管理，有着更复杂的方法。有的语言中的库，如JOGL和LWJGL（对于Java）会提供一些支持动画的类。我们鼓励对于设计特定应用渲染循环（或者“游戏循环”） 感兴趣的读者去读一些在游戏引擎设计上更加专业的图书[NY14]，同时 跟踪在gamedev.net [GD17]上的讨论。

我们在glShaderSource()命令上注释了一个细节。它的第四个参数指定了一个“长度数组”，其中包括给定着色器程序中每行代码的 字符串的整数长度。如果这个参数被设为null，像我们之前那样， OpenGL将会自动从以null结尾的字符串中构建这个数组。因此我们特地确保所有我们传给glShaderSource()的字符串都是以null结尾的[通过在createShaderProgram()中调用c\_str()函数]。实际中通常也会遇 到手动构建这些数组而非传入null的应用程序。

在本书中，读者可能多次想要了解OpenGL某些方面的数值限制。例如，程序员可能需要知道几何着色器可以生成的最大输出数，或者可以为渲染点指定的最大尺寸。这些值中很多都依赖于实现，即在不同的机器上是不同的。OpenGL提供了通过使用glGet()指令来获取这些值的机制。基于查询的参数的不同类型，glGet()指令也有着不同的形式。例如，查询点的尺寸的最大值时，如下调用会将最小值和最大值

（基于运行机器上的OpenGL实现）放入名为“size”的数组中的前两 个元素。

glGetFloatv(GL\_POINT\_SIZE\_RANGE, size)

这类查询有很多。更多示例参见OpenGL参考文档[OP16]。

在本章中，我们尝试在每次OpenGL调用时，描述其各个参数。当我们向后推进时，这么做就会显得冗余，因此当我们觉得描述参数只会妨碍理解时，就不会描述该参数。这是因为很多OpenGL函数有大量 与我们示例无关的参数。必要时读者应当使用OpenGL文档来获取参数详情。