# 从固定管线到可编程管线：十段代码入门OpenGL

## 1. 最简单的OpenGL应用程序

有兴趣点进来阅读此文的同学，想必都已经安装了PyOpenGL，如果没有，请运行pip install pyopengl命令立即安装。使用windows的同学，也可以照此安装，不过可能会遇到因为freeglut.dll缺失或版本错误导致的glutInit函数不可用问题。点击[“Python模块仓库”](https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopengl)下载适合自己的版本，直接安装.[whl](https://so.csdn.net/so/search?q=whl&spm=1001.2101.3001.7020)文件，也许是windows平台上更稳妥的选择。

**demo\_01.py**

#!/usr/bin/env python3

"""最简单的OpenGL应用程序"""

from OpenGL.GL import \* # 导入核心库GL，该库所有函数均以gl为前缀

from OpenGL.GLUT import \* # 导入工具库GLUT，该库所有函数均以glut为前缀

def draw():

"""绘制模型"""

glClear(GL\_COLOR\_BUFFER\_BIT) # 清除缓冲区

glBegin(GL\_TRIANGLES) # 开始绘制三角形

glColor(1.0, 0.0, 0.0) # 设置当前颜色为红色

glVertex(0.0, 1.0, 0.0) # 设置第1个顶点

glColor(0.0, 1.0, 0.0) # 设置当前颜色为绿色

glVertex(-1.0, -1.0, 0.0) # 设置第2个顶点

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glVertex(1.0, -1.0, 0.0) # 设置第3个顶点

glEnd() # 结束绘制三角形

glFlush() # 输出缓冲区

if \_\_name\_\_ == "\_\_main\_\_":

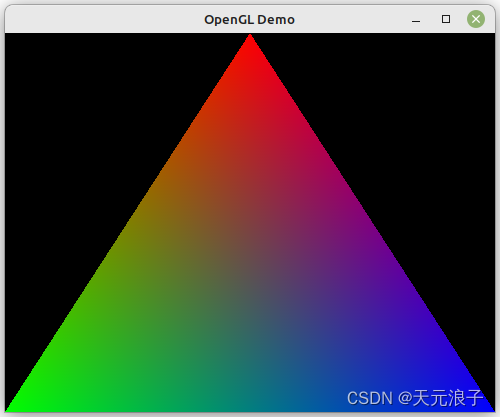
glutInit() # 1. 初始化glut库

glutCreateWindow('OpenGL Demo') # 2. 创建glut窗口

glutDisplayFunc(draw) # 3. 绑定模型绘制函数

glutMainLoop() # 4. 进入glut主循环

这段代码使用OpenGL的GLUT库构建了一个最简单的OpenGL应用程序——绘制三角形，运行界面截图如下。



最简单的OpenGL应用程序（demo\_01.py）

代码前两行导入GL库和GLUT库——前者是OpenGL的核心库，实现绘图功能，后者是不依赖于窗口平台的OpenGL工具库，用于构建窗口程序。尽管OpenGL包含了多个库，最常用的其实只有三个库，除了这段代码用到了GL和GLUT，还有一个实用库GLU，接下来就会用到。每个OpenGL库都有大量的函数，但库和函数命名规则非常合理，每个函数都以小写的库名作为前缀，代码读起来一目了然，非常清晰。

使用GLUT创建OpenGL应用程序非常简单，就像代码中展示的那样，只需要四步——当然，前提是先准备好绘图函数，也就是上面代码中的draw函数。

这段代码中的draw函数演示了线段的绘制流程。OpenGL可以绘制点、线、面等十种基本图元，每种图元的顶点、颜色以及法向量、[纹理](https://so.csdn.net/so/search?q=%E7%BA%B9%E7%90%86&spm=1001.2101.3001.7020)坐标等，都必须包含在glBegin函数和glEnd函数之间，而glBegin函数的参数就是十种基本图元之一。

**OpenGL基本图元速查表**

| **参数** | **说明** |
| --- | --- |
| GL\_POINTS | 绘制一个或多个顶点，顶点之间是相互独立的 |
| GL\_LINES | 绘制线段，每两个顶点对应一条单独的线段。若顶点数为奇数，则忽略最后一个 |
| GL\_LINE\_STRIP | 绘制连续线段，各个顶点依次顺序连接 |
| GL\_LINE\_LOOP | 绘制闭合的线段，各个顶点依次顺序连接，最后首尾相接 |
| GL\_POLYGON | 绘制凸多边形，多边形的边缘决不能相交 |
| GL\_TRIANGLES | 绘制一个或多个三角形，每三个顶点对应一个三角形。若顶点个数不是三的倍数，则忽略多余的顶点 |
| GL\_TRIANGLE\_STRIP | 绘制连续三角形，每个顶点与其前面的两个顶点构成下一个三角形 |
| GL\_TRIANGLE\_FAN | 绘制多个三角形组成的扇形，每个顶点与其前面的顶点和第一个顶点构成下一个三角形 |
| GL\_QUADS | 绘制一个或多个四边形，每四个顶点对应一个四角形。若顶点个数不是四的倍数，则忽略多余的顶点 |
| GL\_QUAD\_STRIP | 绘制连续四边形，每一对顶点与其前面的一对顶点构成下一个四角形 |

借助于图形，更容易理解基本图元的差异和用法。懒得画图了，在网上随便找了一张，顶点顺序也许和读者的习惯并不一致，但在逻辑上是没有问题的。

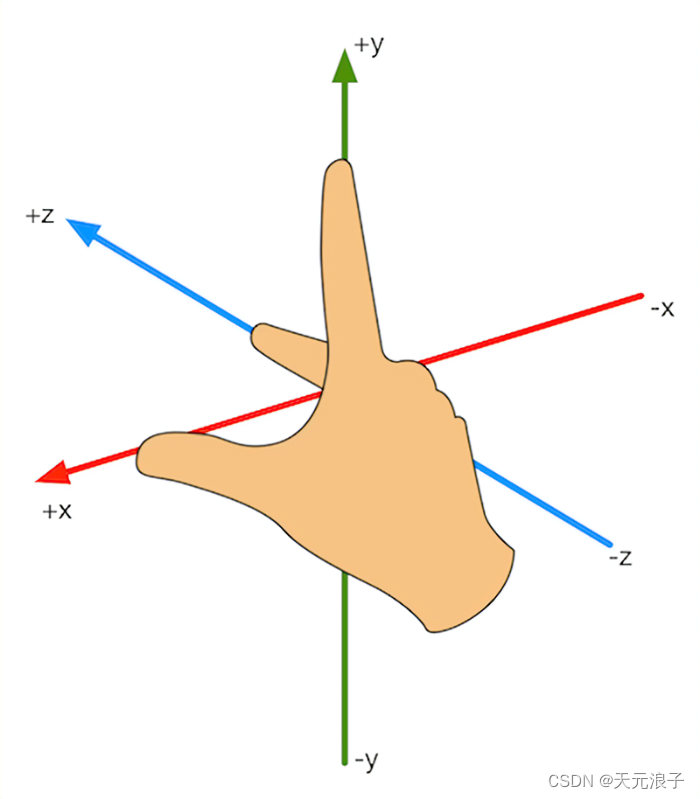
OpenGL基本图元顶点顺序示意图

举个例子：在glBegin函数和glEnd函数之间有6个顶点，序号分别是0～5。如果绘制独立三角面（GL\_TRIANGLES），则顶点(0, 1, 2)构成一个三角形，顶点(3, 4, 5)构成一个三角形，总共2个三角形；如果绘制带状三角面（GL\_TRIANGLE\_STRIP），则有4个三角形，分别是顶点(0, 1, 2)、顶点(2, 1, 3)、顶点(2, 3, 4)和顶点(4, 3, 5)；如果绘制扇面（GL\_TRIANGLE\_FAN），同样有4个三角形，分别是顶点(0, 1, 2)、顶点(0, 2, 3)、顶点(0, 3, 4)和顶点(0, 4, 5)。

绘制三角形、四边形、多边形时，顶点的顺序决定了哪个面是图元的正面——正反面的意义不仅仅是确定法向量的方向，还为固定管线中的面裁剪和可编程管线中的弃用片元提供了依据。在OpenGL系统中默认逆时针顺序构建的面是正面，使用glFrontFace函数可以重新设置。

## 2. 视点系统和投影矩阵

运行第一段代码，改变窗口大小及宽高比，仔细观察就会发现三角形的宽高比例会随窗口宽高比例变化而变化，但是 x轴和 y轴的显示区间始终都是(-1, 1)，x 轴从屏幕的左侧指向右侧，y轴从屏幕的底部指向上方，z轴垂直于屏幕。OpenGL使用右手系坐标，参考下图，不难想象出 z 轴的方向是从屏幕里面指向外面的。



右手系坐标示意图

接下来有一个问题需要思考：如何看到第一段代码展示的三角形的背面呢？无论怎样调整眼睛和屏幕的相对关系，看到的始终是平面的2D效果。设想一下，把有深度的屏幕（默认 z轴就是深度轴）视为一个舞台场景，将一架相机放入场景中，改变相机的位置和拍摄角度，不就可以看到想要的效果了吗？

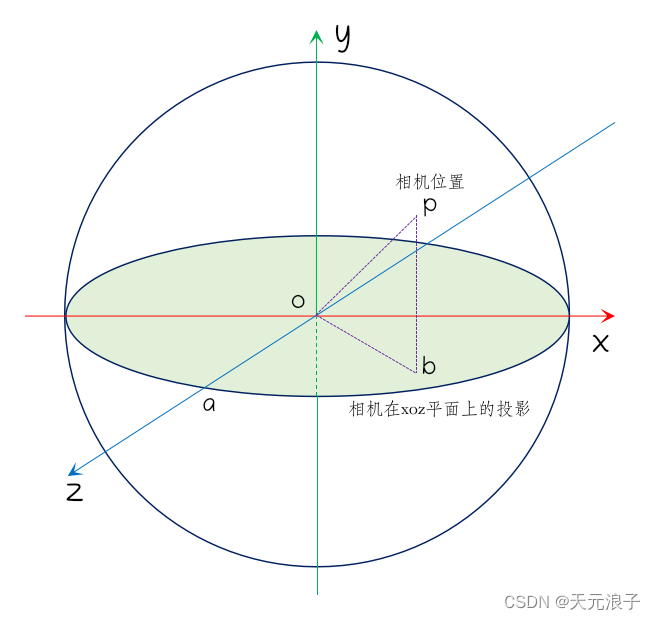
那么，如何定义场景中的这一架相机呢？现实生活中使用相机拍照，大致有以下三个要素。

1. 相机位置：决定了相机和拍摄目标之间的空间相对关系
2. 拍摄角度：相机的姿态，包括横置、竖置、平视、俯视、仰视等
3. 镜头焦距：决定视野宽窄，焦距越小，视野越宽

在3D场景中的相机与之类似，只不过是用相机和拍摄目标之间的距离、相机的方位角和高度角替代了相机位置和拍摄角度，三要素变成了四要素。

1. 相机和拍摄目标之间的距离（代码中简写为dist）
2. 方位角（代码中简写为azim）
3. 高度角（代码中简写为elev）
4. 水平视野角度（代码中简写为fovy）

在一个OpenGL空间直角坐标系中，点 a为 z轴正半轴上一点，假设相机位置在空间点 p处，点 b为点 p在 x o z 平面上的投影，那么线段 o p 的长度就是相机和坐标原点之间的距离dist， ∠ a o b 即为方位角azim，∠ p o b 即为高度角elev。相机绕 y 轴逆时针旋转时，方位角正向增加；相机向 y 轴正方向升高时，高度角正向增加。



相机的方位角和高度角

至此，一个完整的视点系统就建立起来了。视点系统对应着一个矩阵，相机方位角、高度角以及距离的变化就是改变这个矩阵，这个矩阵叫做视点矩阵（View Matrix）。视点矩阵是玩转OpenGL必须要理解的三个矩阵之一，另外两个是投影矩阵（Projection Matrix）和模型矩阵（Model Matrix），三个矩阵合称MVP矩阵。喜欢篮球或足球的话，很容易记住这个组合——MVP，最有价值球员。

**demo\_02.py**

#!/usr/bin/env python3

"""视点系统和投影矩阵"""

import numpy as np

from OpenGL.GL import \*

from OpenGL.GLU import \*

from OpenGL.GLUT import \*

dist = 5.0 # 全局变量：相机与坐标原点之间的距离

azim = 0.0 # 全局变量：方位角

elev = 0.0 # 全局变量：高度角

fovy = 40.0 # 全局变量：水平视野角度

near = 2.0 # 全局变量：最近对焦距离

far = 1000.0 # 全局变量：最远对焦距离

cam = (0.0, 0.0, 5.0) # 全局变量：相机位置

csize = (800, 600) # 全局变量：窗口大小

aspect = csize[0]/csize[1] # 全局变量：窗口宽高比

mouse\_pos = = (0,0)   # 全局变量：鼠标位置，不能用None会报错

def click(btn, state, x, y):

"""鼠标按键和滚轮事件函数"""

global mouse\_pos

if (btn == 0 or btn == 2) and state == 0: # 左键或右键被按下

mouse\_pos = (x, y) # 记录鼠标位置

glutPostRedisplay() # 更新显示

def drag(x, y):

"""鼠标拖拽事件函数"""

global mouse\_pos, azim, elev, cam

dx, dy = x-mouse\_pos[0], y-mouse\_pos[1] # 计算鼠标拖拽距离

mouse\_pos = (x, y) # 更新鼠标位置

azim = azim - 180\*dx/csize[0] # 计算方位角

elev = elev + 90\*dy/csize[1] # 计算高度角

d = dist \* np.cos(np.radians(elev))

x\_cam = d\*np.sin(np.radians(azim))

y\_cam = dist\*np.sin(np.radians(elev))

z\_cam = d\*np.cos(np.radians(azim))

cam = [x\_cam, y\_cam, z\_cam] # 更新相机位置

glutPostRedisplay() # 更新显示

def reshape(w, h):

"""改变窗口大小事件函数"""

global csize, aspect

csize = (w, h) # 保存窗口大小

aspect = w/h if h > 0 else 1e4 # 更新窗口宽高比

glViewport(0, 0, w, h) # 设置视口

glutPostRedisplay() # 更新显示

def draw():

"""绘制模型"""

glClear(GL\_COLOR\_BUFFER\_BIT) # 清除缓冲区

glMatrixMode(GL\_PROJECTION) # 操作投影矩阵

glLoadIdentity() # 将投影矩阵设置为单位矩阵

gluPerspective(fovy, aspect, near, far) # 生成透视投影矩阵

gluLookAt(\*cam, \*[0,0,0], \*[0,1,0]) # 设置视点矩阵

glBegin(GL\_TRIANGLES) # 开始绘制三角形

glColor(1.0, 0.0, 0.0) # 设置当前颜色为红色

glVertex(0.0, 1.0, 0.0) # 设置顶点

glColor(0.0, 1.0, 0.0) # 设置当前颜色为绿色

glVertex(-1.0, -1.0, 0.0) # 设置顶点

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glVertex(1.0, -1.0, 0.0) # 设置顶点

glEnd() # 结束绘制三角形

glBegin(GL\_LINES) # 开始绘制线段

glColor(1.0, 0.0, 1.0) # 设置当前颜色为紫色

glVertex(0.0, 0.0, -1.0) # 设置线段顶点（z轴负方向）

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glVertex(0.0, 0.0, 1.0) # 设置线段顶点（z轴正方向）

glEnd() # 结束绘制线段

glFlush() # 执行缓冲区指令

if \_\_name\_\_ == "\_\_main\_\_":

glutInit() # 1. 初始化glut库

glutInitWindowSize(\*csize) # 2.1 设置窗口大小

glutCreateWindow('OpenGL Demo') # 2.2 创建glut窗口

glutDisplayFunc(draw) # 3.1 绑定模型绘制函数

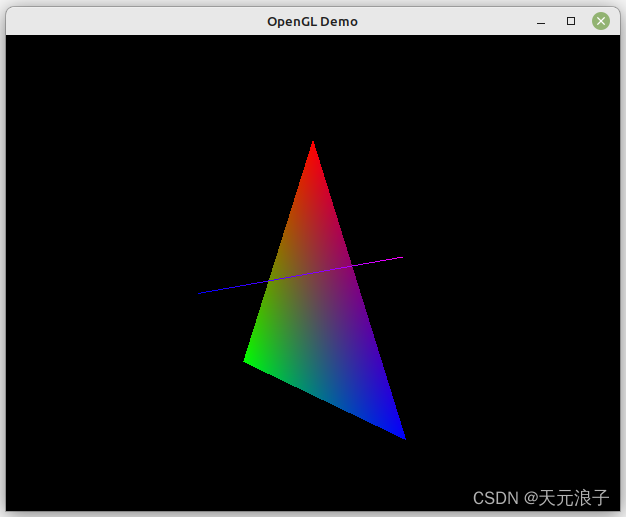
glutReshapeFunc(reshape) # 3.2 绑定窗口大小改变事件函数

glutMouseFunc(click) # 3.3 绑定鼠标按键

glutMotionFunc(drag) # 3.4 绑定鼠标拖拽事件函数

glutMainLoop() # 4. 进入glut主循环

这段代码依旧是绘制了一个三角形，外加一条和 z 轴重合的线段。现在拖拽鼠标改变相机位置就能看到三角形的背面了，并且改变窗口大小时，三角形的宽高比例不再随着窗口的宽高比例的改变而改变。运行界面截图如下。



视点系统和投影矩阵（demo\_02.py）

前面说过OpenGL有三个最常用的库，第一段代码用到了GL库和GLUT库，这段代码用到了了第三个库——GLU库中的两个函数，一个是gluPerspective，用来生成透视投影矩阵，一个是gluLookAt，用来设置视点矩阵。

生成投影矩阵需要4个参数，分别是fovy（相机水平视野角度）、aspect（窗口宽高比），以及near（最近对焦距离）和far（最远对焦距离）。near和far是相对于相机的距离，距离相机小于near或者大于far的目标都不会显示在场景中。

生成视点矩阵需要9个参数，分成3组，分别是相机位置、视点坐标系统原点和指向相机上方的单位向量。相机位置不言自明，视点坐标系统原点可以理解为相机的对焦点，代码中直接使用了坐标系原点，即相机的焦点始终对准坐标系原点。指向相机上方的单位向量说的是相机的姿态——是横置还是竖置抑或是旋转180°，这是暂时固定指向 y轴正方向，后续代码会做相应处理。

## 3. 深度缓冲区和深度测试

在讲述视点矩阵和投影矩阵时，我有意回避了很多复杂的概念，目的是帮助初学者快速理解代码——有了代码，才能在实践中深入理解OpenGL中那些复杂的概念。不过，简化也带来了一些副作用，比如：

1. 三角形无法遮住其背后的线段，没有实现深度遮挡
2. 不能缩放视野，无法抵近观察三角形和线段
3. 使用了很多全局变量，读着吃力，维护起来也很困难，难以实现代码复用

第1个问题中的深度遮挡，简单点说就是前面的模型遮挡后面的模型，复杂点说，对于半透明模型在渲染时还要考虑先后顺序。在3D绘图中实现深度遮挡，只需要开启深度测试并设置深度测试函数的参数就可以了。在此之前，需要先在GLUT的显示模式中启用深度缓冲区。这些工作看似繁琐，其实只需要3行代码。

那么，深度缓冲区究竟是什么？为什么要做深度测试呢？所谓深度缓冲区，就是一块内存区域，存储了屏幕上每个像素点对应的模型上的点和相机的距离，也就是深度值，值越大，离摄像机就越远。渲染像素化后的模型时，要对每个点做深度测试，判断一个点的深度和该点所对应的屏幕上的点的缓存深度谁前谁后，据此决定该点是否被遮挡。深度测试由深度测试函数实现，但需要用户提供测试规则。常用的测试规则见下表。为便于理解，这里我故意混淆了点和片元的概念，待了解了着色器之后，读者自会辨查。

**OpenGL深度测试规则速查表**

| **规则** | **说明** |
| --- | --- |
| GL\_ALWAYS | 总是通过深度测试 |
| GL\_NEVER | 总是不通过深度测试 |
| GL\_LESS | 片元深度值小于缓冲的深度值时通过测试 |
| GL\_EQUAL | 片元深度值等于缓冲区的深度值时通过测试 |
| GL\_LEQUAL | 片元深度值小于等于缓冲区的深度值时通过测试 |
| GL\_GREATER | 片元深度值大于缓冲区的深度值时通过测试 |
| GL\_NOTEQUAL | 片元深度值不等于缓冲区的深度值时通过测试 |
| GL\_GEQUAL | 片元深度值大于等于缓冲区的深度值时通过测试 |

第2个问题，关于缩放视野，可以用改变相机水平视野角度解决，就像拍照时使用变焦镜头拉近拉远一样，变焦就是改变了镜头的视野角度。如果相机没有变焦功能，那就只能走进或远离拍摄目标，也就是改变相机和拍摄目标之间的距离，同样可以缩放视野。

第3个问题，可以用面向对象编程（OOP）的方式解决。下面的代码基于GLUT库封装了一个3D场景类，启用了深度检测，封装了相机拖拽、视野缩放、相机姿态还原等鼠标操作，还支持自定义高度轴——建筑和空间领域更习惯用 z轴作为高度轴。用户只要在派生类中重写draw函数，就可以方便地复用这段代码了。

**demo\_03.py**

#!/usr/bin/env python3

"""深度缓冲区和深度测试"""

import numpy as np

from OpenGL.GL import \*

from OpenGL.GLU import \*

from OpenGL.GLUT import \*

class BaseScene:

"""基于OpenGl.GLUT的三维场景类"""

def \_\_init\_\_(self, \*\*kwds):

"""构造函数"""

self.csize = kwds.get('size', (960, 640)) # 画布分辨率

self.bg = kwds.get('bg', [0.0, 0.0, 0.0]) # 背景色

self.haxis = kwds.get('haxis', 'y').lower() # 高度轴

self.oecs = kwds.get('oecs', [0.0, 0.0, 0.0]) # 视点坐标系ECS原点

self.near = kwds.get('near', 2.0) # 相机与视椎体前端面的距离

self.far = kwds.get('far', 1000.0) # 相机与视椎体后端面的距离

self.fovy = kwds.get('fovy', 40.0) # 相机水平视野角度

self.dist = kwds.get('dist', 5.0) # 相机与ECS原点的距离

self.azim = kwds.get('azim', 0.0) # 方位角

self.elev = kwds.get('elev', 0.0) # 高度角

self.aspect = self.csize[0]/self.csize[1] # 画布宽高比

self.cam = None # 相机位置

self.up = None # 指向相机上方的单位向量

self.\_update\_cam\_and\_up() # 计算相机位置和指向相机上方的单位向量

self.left\_down = False # 左键按下

self.mouse\_pos = (0, 0) # 鼠标位置

# 保存相机初始姿态（视野、方位角、高度角和距离）

self.home = {'fovy':self.fovy, 'azim':self.azim, 'elev':self.elev, 'dist':self.dist}

def \_update\_cam\_and\_up(self, oecs=None, dist=None, azim=None, elev=None):

"""根据当前ECS原点位置、距离、方位角、仰角等参数，重新计算相机位置和up向量"""

if not oecs is None:

self.oecs = [\*oecs,]

if not dist is None:

self.dist = dist

if not azim is None:

self.azim = (azim+180)%360 - 180

if not elev is None:

self.elev = (elev+180)%360 - 180

up = 1.0 if -90 <= self.elev <= 90 else -1.0

azim, elev = np.radians(self.azim), np.radians(self.elev)

d = self.dist \* np.cos(elev)

if self.haxis == 'z':

azim -= 0.5 \* np.pi

self.cam = [d\*np.cos(azim)+self.oecs[0], d\*np.sin(azim)+self.oecs[1], self.dist\*np.sin(elev)+self.oecs[2]]

self.up = [0.0, 0.0, up]

else:

self.cam = [d\*np.sin(azim)+self.oecs[0], self.dist\*np.sin(elev)+self.oecs[1], d\*np.cos(azim)+self.oecs[2]]

self.up = [0.0, up, 0.0]

def reshape(self, w, h):

"""改变窗口大小事件函数"""

self.csize = (w, h)

self.aspect = self.csize[0]/self.csize[1] if self.csize[1] > 0 else 1e4

glViewport(0, 0, self.csize[0], self.csize[1])

glutPostRedisplay()

def click(self, btn, state, x, y):

"""鼠标按键和滚轮事件函数"""

if btn == 0: # 左键

if state == 0: # 按下

self.left\_down = True

self.mouse\_pos = (x, y)

else: # 弹起

self.left\_down = False

elif btn == 2 and state ==1: # 右键弹起，恢复相机初始姿态

self.\_update\_cam\_and\_up(dist=self.home['dist'], azim=self.home['azim'], elev=self.home['elev'])

self.fovy = self.home['fovy']

elif btn == 3 and state == 0: # 滚轮前滚

self.fovy \*= 0.95

elif btn == 4 and state == 0: # 滚轮后滚

self.fovy += (180 - self.fovy) / 180

glutPostRedisplay()

def drag(self, x, y):

"""鼠标拖拽事件函数"""

dx, dy = x - self.mouse\_pos[0], y - self.mouse\_pos[1]

self.mouse\_pos = (x, y)

azim = self.azim - (180\*dx/self.csize[0]) \* (self.up[2] if self.haxis == 'z' else self.up[1])

elev = self.elev + 90\*dy/self.csize[1]

self.\_update\_cam\_and\_up(azim=azim, elev=elev)

glutPostRedisplay()

def prepare(self):

"""GL初始化后、开始绘制前的预处理。可在派生类中重写此方法"""

pass

def draw(self):

"""绘制模型。可在派生类中重写此方法"""

glBegin(GL\_TRIANGLES) # 开始绘制三角形

glColor(1.0, 0.0, 0.0) # 设置当前颜色为红色

glVertex(0.0, 1.0, 0.5) # 设置顶点

glColor(0.0, 1.0, 0.0) # 设置当前颜色为绿色

glVertex(-1.0, -1.0, 0.5) # 设置顶点

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glVertex(1.0, -1.0, 0.5) # 设置顶点

glEnd() # 结束绘制三角形

glBegin(GL\_TRIANGLES) # 开始绘制三角形

glColor(1.0, 0.0, 0.0) # 设置当前颜色为红色

glVertex(0.0, 1.0, -0.5) # 设置顶点

glColor(0.0, 1.0, 0.0) # 设置当前颜色为绿色

glVertex(-1.0, -1.0, -0.5) # 设置顶点

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glVertex(1.0, -1.0, -0.5) # 设置顶点

glEnd() # 结束绘制三角形

def render(self):

"""重绘事件函数"""

# 清除屏幕及深度缓存

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

# 设置投影矩阵

glMatrixMode(GL\_PROJECTION) # 操作投影矩阵

glLoadIdentity() # 将投影矩阵设置为单位矩阵

gluPerspective(self.fovy, self.aspect, self.near, self.far) # 生成透视投影矩阵

# 设置视点矩阵

gluLookAt(\*self.cam, \*self.oecs, \*self.up)

# 设置模型矩阵

glMatrixMode(GL\_MODELVIEW) # 操作模型矩阵

glLoadIdentity() # 将模型矩阵设置为单位矩阵

# 绘制模型

self.draw()

# 交换缓冲区

glutSwapBuffers()

def show(self):

"""显示"""

glutInit() # 初始化glut库

sw, sh = glutGet(GLUT\_SCREEN\_WIDTH), glutGet(GLUT\_SCREEN\_HEIGHT)

left, top = (sw-self.csize[0])//2, (sh-self.csize[1])//2

glutInitWindowSize(\*self.csize) # 设置窗口大小

glutInitWindowPosition(left, top) # 设置窗口位置

glutCreateWindow('Data View Toolkit') # 创建窗口

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_ALPHA | GLUT\_DEPTH) # 设置显示模式

glClearColor(\*self.bg, 1.0) # 设置背景色

glEnable(GL\_DEPTH\_TEST) # 开启深度测试

glDepthFunc(GL\_LEQUAL) # 设置深度测试函数的参数

self.prepare() # GL初始化后、开始绘制前的预处理

glutDisplayFunc(self.render) # 绑定重绘事件函数

glutReshapeFunc(self.reshape) # 绑定窗口大小改变事件函数

glutMouseFunc(self.click) # 绑定鼠标按键和滚轮事件函数

glutMotionFunc(self.drag) # 绑定鼠标拖拽事件函数

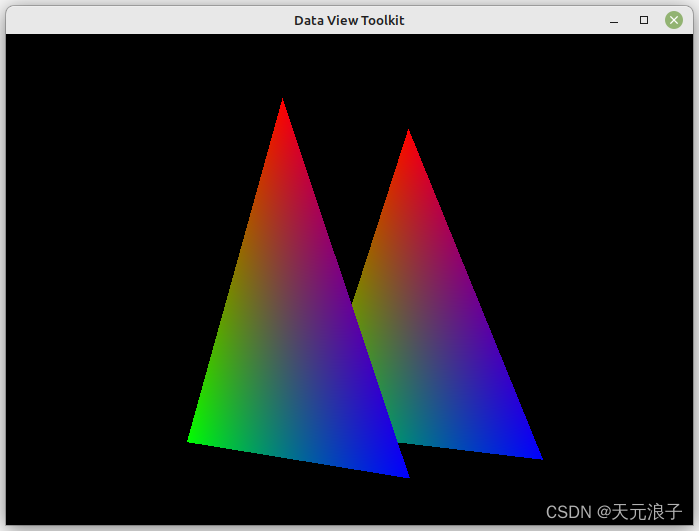
glutMainLoop() # 进入glut主循环

if \_\_name\_\_ == '\_\_main\_\_':

app = BaseScene(haxis='y')

app.show()

这段代码绘制了一前一后两个完全相同的三角形，拽拽鼠标、滚动鼠标滚轮，可以改变相机位置、缩放视野，视觉上呈现出近大远小、前面遮挡后面的效果。



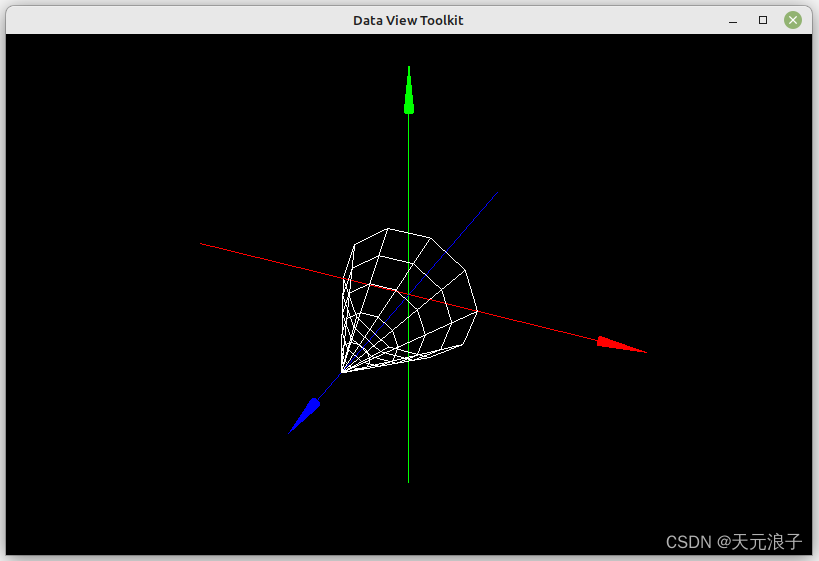
深度缓冲区和深度测试（demo\_03.py）

不同于前两段代码将draw函数绑定为GLUT的重绘函数，这段代码将重绘函数命名为render，draw函数退化成一个纯粹的绘图函数，并由render函数调用。在调用draw函数之前，render函数先清除屏幕及深度缓存（显示模式设置中已经启用了深度缓存），再根据当前的相机视野宽度、视口宽高比设置投影矩阵（P矩阵），最后根据当前的相机位置等参数设置视点矩阵（V矩阵）；在调用draw函数之后，render函数调用GLUT库的交换前后缓冲区（显示模式设置中已经启用了双缓冲区）函数glutSwapBuffers以更新显示内容。

另外这段代码中还隐藏了一个未曾使用的prepare方法，该方法在OpenGL初始化后、重绘函数render被绑定之前被调用，用于数据预处理，用户可在派生类中重写此方法。

## 4. 模型的旋转和平移

下图是继承前一段代码的BsaeScene类并重写draw方法之后的运行结果，场景中以红绿蓝三色显示了带有锥形箭头的xyz轴以及一个用线框表示的白色锥形。



模型的旋转和平移（demo\_04.py）

这几个圆锥是如何绘制的呢？仔细观察就会发现，圆锥的锥尖部分是由多个三角形组成的，这些三角形都以锥尖作为共同的顶点——这不就是扇形图元吗？使用GL\_TRISNGLE\_FAN就可以绘制出来。锥体部分则是带状四边形图元，使用GL\_QUAD\_STRIP也可以绘制出来。这么分析下来，绘制圆锥似乎并不困难，不过要计算出空间中任意位置的锥形的全部顶点并非易事，因此GLUT库提供了很多函数来绘制圆锥以及其他常见的几何形体，比如立方体、圆柱体、球体等，每种几何体的绘制函数又有线框几何体函数和实心几何体函数之分。上图中的白色圆锥就是用线框圆锥体函数绘制的，三个坐标轴上的圆锥则是实心圆锥体函数绘制的。

**GLUT自带的几何体绘制函数速查表**

| **参数** | **说明** |
| --- | --- |
| glutWireCube() | 线框立方体，参数：边长 |
| glutSolidCube() | 实线立方体，参数：边长 |
| glutWireSphere() | 线框球体，参数：半径、环Z轴分片数、沿z轴分片数 |
| glutSolidSphere() | 实心球体，参数：半径、环Z轴分片数、沿z轴分片数 |
| glutWireCone() | 线框圆锥体，参数：锥底半径，锥高，环Z轴分片数、沿z轴分片数 |
| glutSolidCone() | 实心圆锥体，参数：锥底半径，锥高，环Z轴分片数、沿z轴分片数 |
| glutWireCylinder() | 线框圆柱体，参数：半径，高，环Z轴分片数、沿z轴分片数 |
| glutSolidCylinder() | 实心圆柱体，参数：半径，高，环Z轴分片数、沿z轴分片数 |
| glutWireTorus() | 线框圆环，参数：圆半径，环半径，圆分片数、环分片数 |
| glutSolidTorus() | 实心圆环，参数：圆半径，环半径，圆分片数、环分片数 |
| glutWireDodecahedron() | 线框正十二面体，半径为3的平方根 |
| glutSolidDodecahedron() | 实心正十二面体，半径为3的平方根 |
| glutWireOctahedron() | 线框正八面体，半径为1 |
| glutSolidOctahedron() | 实心正八面体，半径为1 |
| glutWireTetrahedron() | 线框正四面体，半径为3的平方根 |
| glutSolidTetrahedron() | 实心正四面体，半径为3的平方根 |
| glutWireIcosahedron() | 线框的正二十面体，半径为1 |
| glutSolidIcosahedron() | 实心的正二十面体，半径为1 |
| glutWireRhombicDodecahedron() | 线框斜十二面体，半径为1 |
| glutSolidRhombicDodecahedron() | 实心斜十二面体 |
| glutWireSierpinskiSponge () | 线框迭代海绵体，参数：迭代次数，中心位置，大小 |
| glutSolidSierpinskiSponge() | 实心迭代海绵体，参数：迭代次数，中心位置，大小 |
| glutWireTeapot() | 线框茶壶，参数：大小 |
| glutSolidTeapot() | 实心茶壶：大小 |

然而，不管是线框圆锥是实体圆锥，锥底中心都在坐标系原点，锥尖都在 z 轴的正半轴上，圆锥绘制函数只接受锥底半径、锥高、环绕 z 轴的分片数和沿着 z 轴的分片数等4个参数。那么问题来了：坐标轴上锥形箭头的锥底圆心都不在坐标系原点，它们是怎么绘制出来的呢？这时候模型矩阵（M矩阵）闪亮登场了。

数学告诉我们，空间中一个刚体经过一系列旋转和平移后一定可以到达另一个位置，旋转和平移被称作几何变换，一系列几何变换可以用一个矩阵描述，这个矩阵就是模型矩阵。以 x轴的圆锥为例：先绘制一个锥底位于原点的实体圆锥，再绕 y轴逆时针旋转90°，此时锥尖朝向 x的正方向，接下来平移圆锥至目标位置即可。

**demo\_04.py**

#!/usr/bin/env python3

"""模型的旋转和平移"""

from OpenGL.GL import \*

from OpenGL.GLUT import \*

from demo\_03 import BaseScene

class App(BaseScene):

"""由BaseScene派生的3D应用程序类"""

def draw(self):

"""绘制模型"""

glBegin(GL\_LINES) # 开始绘制线段

glColor(1.0, 0.0, 0.0) # 设置当前颜色为红色

glVertex(-1.0, 0.0, 0.0) # 设置线段顶点（x轴负方向）

glVertex(0.8, 0.0, 0.0) # 设置线段顶点（x轴正方向）

glColor(0.0, 1.0, 0.0) # 设置当前颜色为绿色

glVertex(0.0, -1.0, 0.0) # 设置线段顶点（y轴负方向）

glVertex(0.0, 0.8, 0.0) # 设置线段顶点（y轴正方向）

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glVertex(0.0, 0.0, -1.0) # 设置线段顶点（z轴负方向）

glVertex(0.0, 0.0, 0.8) # 设置线段顶点（z轴正方向）

glEnd() # 结束绘制线段

glColor(1.0, 1.0, 1.0) # 设置当前颜色为白色

glutWireCone(0.3, 0.6, 10, 5) # 绘制圆锥（线框）

glPushMatrix() # 保存当前矩阵

glRotate(90, 0, 1, 0) # 绕向量(0,1,0)即y轴旋转90度

glTranslate(0, 0, 0.8) # x轴变z轴，向z轴正方向平移

glColor(1.0, 0.0, 0.0) # 设置当前颜色为红色

glutSolidCone(0.02, 0.2, 30, 30) # 绘制圆锥（实体）

glPopMatrix() # 恢复保存的矩阵

glPushMatrix() # 保存当前矩阵

glRotate(-90, 1, 0, 0) # 绕向量(1,0,0)即x轴旋转-90度

glTranslate(0, 0, 0.8) # y轴变z轴，向z轴正方向平移

glColor(0.0, 1.0, 0.0) # 设置当前颜色为绿色

glutSolidCone(0.02, 0.2, 30, 30) # 绘制圆锥（实体）

glPopMatrix() # 恢复保存的矩阵

glPushMatrix() # 保存当前矩阵

glTranslate(0, 0, 0.8) # 向z轴正方向平移

glColor(0.0, 0.0, 1.0) # 设置当前颜色为蓝色

glutSolidCone(0.02, 0.2, 30, 30) # 绘制圆锥（实体）

glPopMatrix() # 恢复保存的矩阵

if \_\_name\_\_ == '\_\_main\_\_':

app = App(hax='y')

app.show()

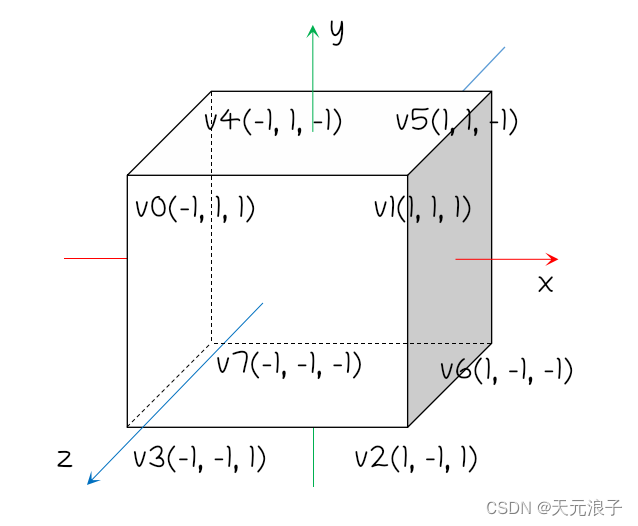
代码中的App类是BaseScene的派生类，只重写了draw方法。在draw方法中绘制了多个模型，仅当被绘制的模型需要旋转或平移时，才启用模型矩阵。启用模型矩阵时，需要先保存所有矩阵的当前状态，模型绘制完成后，还要恢复保存的所有矩阵。模型平移函数glTranslate的三个参数分别是模型在 x y z三个轴上的偏移量。模型旋转函数glRotate的第一个参数是旋转角度，其后三个参数是旋转轴向量。旋转角度的正负遵从右手定则，即右手拇指指向旋转轴向量的方向，其余四指轻握，手指方向即使旋转角度的正方向。

## 5. VBO和顶点混合数组

使用GLUT构建的窗口程序，每当窗口移动或改变大小，以及鼠标操作时，都会触发重绘事件，从而引发重绘函数被调用。每次重绘时，draw函数里面的顶点总是从CPU传送到GPU。实际应用OpenGL绘制三维模型，往往需要处理数以万计的顶点，有时甚至是百万级、千万级，如果也用前面的方法传送顶点数据，势必消耗大量的CPU资源。因此，通常不会在绘制函数里面传送这些数据，而是在绘制之前，将这些数据提前传送到GPU。绘制函数每次绘制时，只需要从GPU的缓存中取出数据即可，极大地提高了效率。这个机制的实现，依赖于顶点缓冲区对象（Vertex Buffer Object），简称VBO。

需要说明一下，这里提到的顶点数据，不仅指顶点的坐标，还包括顶点颜色、法向量、纹理坐标，以及顶点索引。谈论一个模型，顶点坐标是必不可少的，顶点颜色无需解释，想要呈现光照效果的话顶点法向量是必需的，使用纹理贴图的话还需要提供顶点的纹理坐标。那么，顶点索引是什么呢？

假如一个六面体的8个顶点的编号及坐标如下所示。要用四边形图元绘制这个六面体的话，每个顶点需要重复3次，变成了24个顶点；如果用三角形图元绘制，则有12个三角形，共计36个顶点。



六面体8个顶点的序号

怎样才能在不增加顶点数量的前提下，描述图元的顶点关系呢？这就是顶点索引要解决的问题。顶点索引是一个整型数组，用各个顶点在顶点数组内的索引号来描述三角形或四边形的顶点顺序。比如顶点索引数组[0, 3, 2, 1, 1, 2, 6, 5]，表示以逆时针顺序描述的上面这个六面体的前面和右侧的两个四边形。

在VBO保存的顶点数据集，除了顶点信息外，还可以包含颜色、法向量、纹理坐标等数据，这就是顶点混合数组。假定我们在上面的顶点集中增加每个顶点的颜色，可以写成这样：

vertices = np.array([

[0.0, 0.5, 1.0, -1, 1, 1], # c0-v0

[0.5, 1.0, 0.0, 1, 1, 1], # c1-v1

[1.0, 0.0, 0.5, 1, -1, 1], # c2-v2

[0.0, 1.0, 0.5, -1, -1, 1], # c3-v3

[0.5, 0.0, 1.0, -1, 1, -1], # c4-v4

[1.0, 0.5, 0.0, 1, 1, -1], # c5-v5

[0.0, 1.0, 1.0, 1, -1, -1], # c6-v6

[1.0, 1.0, 0.0, -1, -1, -1] # c7-v7

], dtype=np.float32)

四边形索引数组是这样的（写成二维数组是为了阅读方便也可以，OpenGL会自动将二维数组转成一维）：

indices\_cube = np.array([

[0, 3, 2, 1], # v0-v3-v2-v1 (front)

[4, 0, 1, 5], # v4-v0-v1-v5 (top)

[3, 7, 6, 2], # v3-v7-v6-v2 (bottom)

[7, 4, 5, 6], # v7-v4-v5-v6 (back)

[1, 2, 6, 5], # v1-v2-v6-v5 (right)

[4, 7, 3, 0] # v4-v7-v3-v0 (left)

], dtype=np.int32)

顶点的数据类型是np.float32，索引的数据类型是np.int32，根据数据类型的不同，VBO也分为有两种类型：顶点VBO和索引VBO。创建VBO的核心代码如下：

from OpenGL.arrays import vbo

vbo\_vertices = vbo.VBO(vertices) # 顶点VBO

vbo\_indices = vbo.VBO(indices, target=GL\_ELEMENT\_ARRAY\_BUFFER) # 索引VBO

VBO数组创建后，重绘时使用glInterleavedArrays函数可以从顶点混合数组中根据混合数组的类型分离出顶点坐标数据以及颜色、法向量和纹理坐标等数据。OpenGL支持的混合数组类型总共有14个种，详见下表。

**OpenGL支持的顶点混合数组类型速查表**

| **参数** | **说明** |
| --- | --- |
| GL\_V2F | 2个float32型的顶点坐标 |
| GL\_V3F | 3个float32型的顶点坐标 |
| GL\_C4UB\_V2F | 4通道8位无符号整型颜色 + 2个32位浮点型的顶点坐标 |
| GL\_C4UB\_V3F | 4通道8位无符号整型颜色 + 3个32位浮点型的顶点坐标 |
| GL\_C3F\_V3F | 3通道32位浮点型颜色 + 3个32位浮点型的顶点坐标 |
| GL\_N3F\_V3F | 3个32位浮点型法向量 + 3个32位浮点型的顶点坐标 |
| GL\_C4F\_N3F\_V3F | 4通道32位浮点型颜色 + 3个32位浮点型法向量 + 3个32位浮点型的顶点坐标 |
| GL\_T2F\_V3F | 2个32位浮点型纹理坐标 + 3个32位浮点型的顶点坐标 |
| GL\_T4F\_V4F | 4个32位浮点型纹理坐标 + 4个32位浮点型的顶点坐标 |
| GL\_T2F\_C4UB\_V3F | 2个32位浮点型纹理坐标 + 4通道8位无符号整型颜色 + 3个32位浮点型的顶点坐标 |
| GL\_T2F\_C3F\_V3F | 2个32位浮点型纹理坐标 + 3通道32位浮点型颜色 + 3个32位浮点型的顶点坐标 |
| GL\_T2F\_N3F\_V3F | 2个32位浮点型纹理坐标 + 3个32位浮点型法向量 + 3个32位浮点型的顶点坐标 |
| GL\_T2F\_C4F\_N3F\_V3F | 2个32位浮点型纹理坐标 + 4通道32位浮点型颜色 + 3个32位浮点型法向量 + 3个32位浮点型的顶点坐标 |
| GL\_T4F\_C4F\_N3F\_V4F | 4个32位浮点型纹理坐标 + 4通道32位浮点型颜色 + 3个32位浮点型法向量 + 4个32位浮点型的顶点坐标 |

每次重绘时需要先绑定顶点VBO和索引VBO（如果存在的话），再使用glInterleavedArrays分理出顶点、颜色、法线等数据。如果有索引VBO，则使用glDrawElements函数绘制图元，否则，使用glDrawArrays函数绘制图元。glDrawElements函数的第2个参数是索引数组的长度，glDrawArrays函数的第3个参数是顶点个数，这两处是最容易产生错误的地方。

**demo\_05.py**

#!/usr/bin/env python3

"""顶点缓冲区对象和顶点混合数组"""

import numpy as np

from OpenGL.GL import \*

from OpenGL.arrays import vbo

from demo\_03 import BaseScene

class App(BaseScene):

"""由BaseScene派生的3D应用程序类"""

def \_\_init\_\_(self, \*\*kwds):

"""构造函数"""

BaseScene.\_\_init\_\_(self, \*\*kwds) # 调用基类构造函数

self.models = list() # 模型列表，保存模型的VBO

def prepare(self):

"""顶点数据预处理"""

# 三角形颜色、顶点数组

vertices\_triangle = np.array([

[1.0, 0.0, 0.0, 0.0, 1.2, 0.0], # c0-v0

[0.0, 1.0, 0.0, -1.2, -1.2, 0.0], # c1-v1

[0.0, 0.0, 1.0, 1.2, -1.2, 0.0], # c2-v2

], dtype=np.float32)

# 六面体颜色顶点数组

vertices\_cube = np.array([

[0.0, 0.5, 1.0, -1, 1, 1], # c0-v0

[0.5, 1.0, 0.0, 1, 1, 1], # c1-v1

[1.0, 0.0, 0.5, 1, -1, 1], # c2-v2

[0.0, 1.0, 0.5, -1, -1, 1], # c3-v3

[0.5, 0.0, 1.0, -1, 1, -1], # c4-v4

[1.0, 0.5, 0.0, 1, 1, -1], # c5-v5

[0.0, 1.0, 1.0, 1, -1, -1], # c6-v6

[1.0, 1.0, 0.0, -1, -1, -1] # c7-v7

], dtype=np.float32)

# 六面体索引数组

indices\_cube = np.array([

[0, 3, 2, 1], # v0-v1-v2-v3 (front)

[4, 0, 1, 5], # v4-v5-v1-v0 (top)

[3, 7, 6, 2], # v3-v2-v6-v7 (bottom)

[7, 4, 5, 6], # v5-v4-v7-v6 (back)

[1, 2, 6, 5], # v1-v5-v6-v2 (right)

[4, 7, 3, 0] # v4-v0-v3-v7 (left)

], dtype=np.int32)

vs\_triangle = vbo.VBO(vertices\_triangle)

vs\_cube = vbo.VBO(vertices\_cube)

idx\_cube = vbo.VBO(indices\_cube, target=GL\_ELEMENT\_ARRAY\_BUFFER)

self.models.append({

'gltype': GL\_TRIANGLES, # 图元类型

'atype': GL\_C3F\_V3F, # 混合数组类型

'vbo\_vs': vs\_triangle, # 顶点vbo

'n\_vs': len(vs\_triangle), # 顶点数量

'vbo\_idx': None, # 索引vbo

'n\_idx': None # 索引长度

})

self.models.append({

'gltype': GL\_QUADS, # 图元类型

'atype': GL\_C3F\_V3F, # 混合数组类型

'vbo\_vs': vs\_cube, # 顶点vbo

'n\_vs': len(vs\_cube), # 顶点数量

'vbo\_idx': idx\_cube, # 索引vbo

'n\_idx': indices\_cube.size # 索引长度

})

def draw(self):

"""绘制模型。可在派生类中重写此方法"""

for m in self.models:

m['vbo\_vs'].bind() # 绑定当前顶点VBO

glInterleavedArrays(m['atype'], 0, None) # 根据数组类型分离顶点混合数组

if m['vbo\_idx']: # 如果存在索引VBO

m['vbo\_idx'].bind() # 帮顶当前索引VBO

glDrawElements(m['gltype'], m['n\_idx'], GL\_UNSIGNED\_INT, None)

m['vbo\_idx'].unbind() # 解绑当前索引VBO

else:

glDrawArrays(m['gltype'], 0, m['n\_vs'])

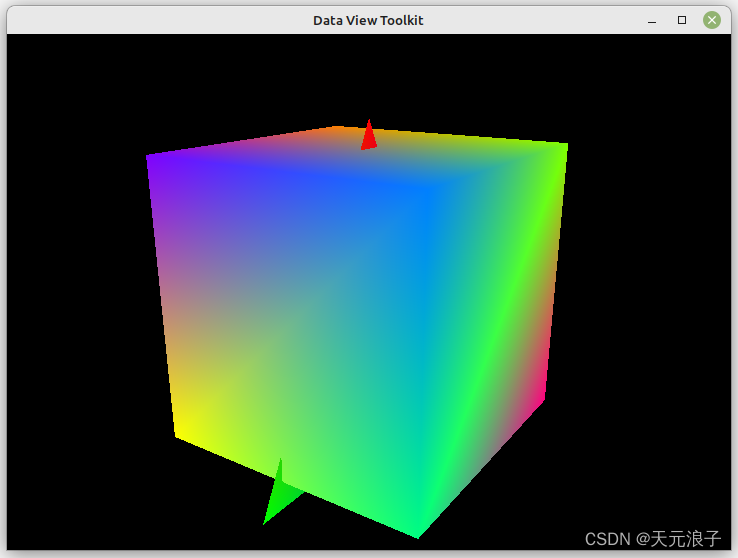
m['vbo\_vs'].unbind() # 解绑当前顶点VBO

if \_\_name\_\_ == '\_\_main\_\_':

app = App(hax='y', azim=-30, elev=10)

app.show()

这段代码绘制了一个位于xoy平面上的三角形和一个中心点位于坐标系原点的六面体，三角形没有使用顶点索引，六面体给出了8个顶点，使用顶点索引生成了6个四边形。



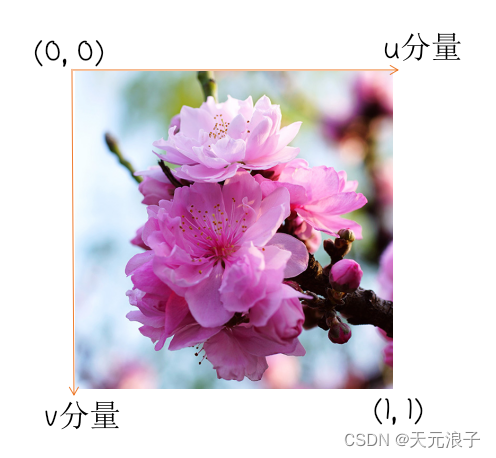
VBO和顶点混合数组（demo\_05.py）

对于初学者来说，阅读这段代码最大的困惑也许是纠结于顶点混合数组分离出来的数据去哪儿了。如果把OpenGL视为一条全自动的生产线，顶点混合数组就像是生产原料，操作者只需要把原料投放到生产线上，接下来这些原料会自动被切割、分捡，传送到下一道工序。这样一想，是不是就不再纠结了呢？事实上OpenGL的确是一条生产线，通常被称为渲染管线。到目前为止，我所描述的这条管线还是一条“固定管线”——所有的顶点数据一旦被投放到这条管线上，就只能按照内部固化的渲染流程走下去，开发者没有任何手段干预这个流程。

## 6. 纹理映射和纹理坐标

如果用GL\_QUADS方法绘制一个四角面，需要提供4个顶点的坐标和颜色，那么这个四角面上其他点的颜色如何控制呢？换个说法，怎样把一张图片贴到一个四角面上呢？这就要用到纹理映射和纹理坐标了。

对于OpenGl来说，纹理就是在GPU中创建的一组位置关系保持相对固定的颜色集合，比如将一个图像的数据传输到GPU，就是一个2D纹理对象。将一个2D纹理对象的各个颜色按照它们的相对位置关系投射到四角面上，就是纹理映射。需要说明的是，纹理对象不止有2D的，还是1D的、3D的、数组的等多种形式，2D纹理对象也不是只能映射到四角面上。纹理对象更像是一个颜色仓库，只要告诉它需要哪一层、哪一行、哪一列，就可以取到这个位置对应的颜色，颜色的位置就是纹理坐标。



2D纹理坐标示意图

2D纹理对象的纹理坐标是由u分量和v分量组成的二元组，u分量和v分量可以理解为水平方向和垂直方向。u分量和v分量的值域是[0,1]闭区间，但如果纹理对象铺贴方式是可重复的，那么超出值域范围的纹理坐标是有效的，比如2.1等价于0.1。传统上把左下角视为纹理坐标的原点，现在有更多的人喜欢用左上角作为原点，这二者仅仅是习惯的不同，并无优劣或血统差异。

**demo\_06.py**

#!/usr/bin/env python3

"""纹理映射和纹理坐标"""

import numpy as np

from PIL import Image

from OpenGL.GL import \*

from OpenGL.arrays import vbo

from demo\_03 import BaseScene

class App(BaseScene):

"""由BaseScene派生的3D应用程序类"""

def \_\_init\_\_(self, \*\*kwds):

"""构造函数"""

BaseScene.\_\_init\_\_(self, \*\*kwds) # 调用基类构造函数

self.models = list() # 模型列表，保存模型的VBO

def create\_texture\_2d(self, texture\_file):

"""创建纹理对象"""

im = np.array(Image.open(texture\_file))

im\_h, im\_w = im.shape[:2]

im\_mode = GL\_LUMINANCE if im.ndim == 2 else (GL\_RGB, GL\_RGBA)[im.shape[-1]-3]

tid = glGenTextures(1)

glBindTexture(GL\_TEXTURE\_2D, tid)

if (im.size/im\_h)%4 == 0:

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 4)

else:

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1)

glTexImage2D(GL\_TEXTURE\_2D, 0, im\_mode, im\_w, im\_h, 0, im\_mode, GL\_UNSIGNED\_BYTE, im)

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR) # 缩小滤波器：线性滤波

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR) # 放大滤波器：线性滤波

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT) # S方向铺贴方式：重复

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT) # T方向铺贴方式：重复

glGenerateMipmap(GL\_TEXTURE\_2D)

glBindTexture(GL\_TEXTURE\_2D, 0)

return tid

def prepare(self):

"""顶点数据预处理"""

vertices = np.array([

[0, 0, -1, 1, 1], # t0-v0

[1, 0, 1, 1, 1], # t1-v1

[1, 1, 1, -1, 1], # t2-v2

[0, 1, -1, -1, 1], # t3-v3

[1, 0, -1, 1, -1], # t4-v4

[0, 0, 1, 1, -1], # t5-v5

[0, 1, 1, -1, -1], # t6-v6

[1, 1, -1, -1, -1] # t7-v7

], dtype=np.float32)

indices = np.array([

[0, 3, 2, 1], # v0-v1-v2-v3 (front)

[5, 6, 7, 4], # v5-v4-v7-v6 (back)

], dtype=np.int32)

vbo\_vs = vbo.VBO(vertices)

vbo\_idx = vbo.VBO(indices, target=GL\_ELEMENT\_ARRAY\_BUFFER)

texture = self.create\_texture\_2d('res/flower.jpg')

glEnable(GL\_TEXTURE\_2D)

self.texture = self.create\_texture\_2d('res/flower.jpg')

self.models.append({

'gltype': GL\_QUADS, # 图元类型

'atype': GL\_T2F\_V3F, # 混合数组类型

'vbo\_vs': vbo\_vs, # 顶点vbo

'n\_vs': len(vertices), # 顶点数量

'vbo\_idx': vbo\_idx, # 索引vbo

'texture': texture, # 纹理对象

'ttype': GL\_TEXTURE\_2D # 纹理类型

})

def draw(self):

"""绘制模型。可在派生类中重写此方法"""

for m in self.models:

if m.get('texture'): # 如果当前模型使用了纹理

glBindTexture(m['ttype'], m['texture']) # 绑定该纹理

m['vbo\_vs'].bind() # 绑定当前顶点VBO

glInterleavedArrays(m['atype'], 0, None) # 根据数组类型分离顶点混合数组

if m['vbo\_idx']: # 如果存在索引VBO

m['vbo\_idx'].bind() # 帮顶当前索引VBO

glDrawElements(m['gltype'], m['vbo\_idx'].size//4, GL\_UNSIGNED\_INT, None)

m['vbo\_idx'].unbind() # 解绑当前索引VBO

else:

glDrawArrays(m['gltype'], 0, m['n\_vs'])

m['vbo\_vs'].unbind() # 解绑当前顶点VBO

if m.get('texture'): # 如果当前模型使用了纹理

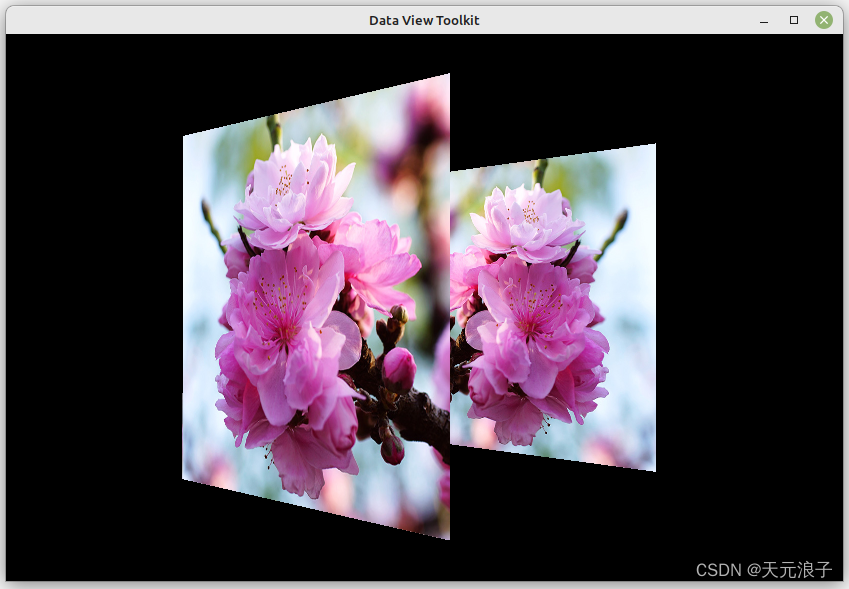
glBindTexture(m['ttype'], m['texture']) # 解绑该纹理

if \_\_name\_\_ == '\_\_main\_\_':

app = App(hax='y')

app.show()

这段代码仍然是继承BaseScene类，重写了prepare方法和draw方法，添加了create\_texture\_2d方法。传入一个图像文件名，create\_texture\_2d方法就返回一个2D纹理对象。因为每个顶点对应的纹理坐标已经在顶点混合数组中发送到了GPU，draw函数在重绘时，只需要绑定对应的2D纹理对象即可。代码中用到了一个图片文件，读者可以任意替换成自己喜欢的本地图像文件。



纹理映射和纹理坐标（demo\_06.py）

有必要重点说一下创建纹理对象时的滤波方式设置。像素化后的模型和纹理对象的点不是一一对应的，如果纹理图片很大，模型在屏幕上占据的像素数很少，或者反过来，纹理图片很小而模型在屏幕上占据的像素数很多，此时就需要选择合适的放大滤波方式和缩小滤波方式，否则相机或模型移动时就会产生抖晃感。上面的代码中放大滤波器和缩小滤波器都使用了是线性滤波（GL\_LINEAR），如果纹理图片的分辨率较高，请尝试将缩小滤波器改为GL\_LINEAR\_MIPMAP\_NEAREST或者GL\_NEAREST\_MIPMAP\_LINEAR等方式，以消除干扰纹理或抖晃。

## 7. 光照和法向量计算

喜欢摄影的人都知道，摄影是用光的艺术，光影语言是摄影艺术最好的呈现方式。其实，3D渲染又何尝不是如此呢？没有灯光的模型没有灵性，立体感和真实感都会大打折扣。下面这段代码绘制了一个圆球和一个六面体，一束自下而上的偏绿色光线照射到圆球上，一束自上而下的略微昏暗的光线照射到六面体上。

**demo\_07.py**

#!/usr/bin/env python3

"""光照和法向量计算"""

import numpy as np

from OpenGL.GL import \*

from OpenGL.GLUT import \*

from demo\_03 import BaseScene

class App(BaseScene):

"""由BaseScene派生的3D应用程序类"""

def \_\_init\_\_(self, \*\*kwds):

"""构造函数"""

BaseScene.\_\_init\_\_(self, \*\*kwds) # 调用基类构造函数

self.models = list() # 模型列表，保存模型的VBO

def prepare(self):

"""初始化灯光"""

glLightfv(GL\_LIGHT0, GL\_AMBIENT, (0.8,1.0,0.8,1.0)) # 光源中的环境光

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, (0.8,0.8,0.8,1.0)) # 光源中的散射光

glLightfv(GL\_LIGHT0, GL\_SPECULAR, (0.2,0.2,0.2,1.0)) # 光源中的反射光

glLightfv(GL\_LIGHT0, GL\_POSITION, (2.0,-20.0,5.0,1.0)) # 光源位置

glLightfv(GL\_LIGHT1, GL\_AMBIENT, (0.5,0.5,0.5,1.0)) # 光源中的环境光

glLightfv(GL\_LIGHT1, GL\_DIFFUSE, (0.3,0.3,0.3,1.0)) # 光源中的散射光

glLightfv(GL\_LIGHT1, GL\_SPECULAR, (0.2,0.2,0.2,1.0)) # 光源中的反射光

glLightfv(GL\_LIGHT1, GL\_POSITION, (-2.0,20.0,-1.0,1.0)) # 光源位置

def draw(self):

"""绘制模型。可在派生类中重写此方法"""

glPushAttrib(GL\_ALL\_ATTRIB\_BITS) # 保存当前全部环境设置

glEnable(GL\_LIGHTING) # 启用光照

glEnable(GL\_LIGHT0) # 开启的灯光0：光线偏绿色，从下向上照射

glPushMatrix() # 保存当前矩阵

glTranslate(-1, 0, 0) # 平移

glutSolidSphere(1, 180, 90) # 绘制球

glPopMatrix() # 恢复保存的矩阵

glPopAttrib() # 恢复当前全部环境设置

glPushAttrib(GL\_ALL\_ATTRIB\_BITS) # 保存当前全部环境设置

glEnable(GL\_LIGHTING) # 启用光照

glEnable(GL\_LIGHT1) # 开启的灯光1：稍暗淡，从上向下照射

glPushMatrix() # 保存当前矩阵

glTranslate(1, 0, 0) # 平移

glutSolidCube(1) # 绘制六面体

glPopMatrix() # 恢复保存的矩阵

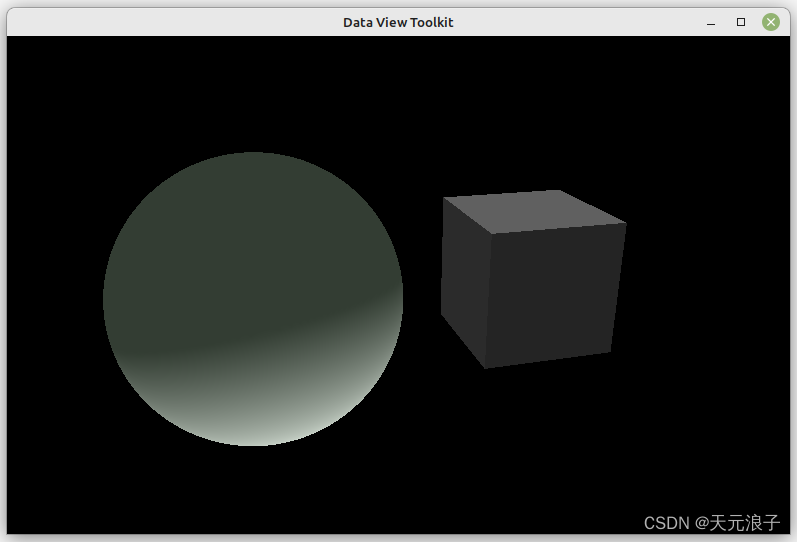
glPopAttrib() # 恢复当前全部环境设置

if \_\_name\_\_ == '\_\_main\_\_':

app = App(hax='y')

app.show()

两个模型的光照效果如下图所示。



光照和法向量计算（demo\_07.py）

常识告诉我们，物体表面呈现出的颜色和亮度是由反射光线决定，而反射——不管是镜面反射还是漫反射，都遵从光的反射定律：入射光线、反射光线、法线在同一平面上；反射光线和入射光线分居在法线的两侧；反射角等于入射角。 在反射面和入射光线已知的条件下，法线计算成为计算物体表面亮度和颜色的前提。

法线是垂直于反射面的向量，因此也被称为法向量。如何计算一个平面的法向量呢？最简单的办法就是用该平面上两个不同的向量叉乘，结果就是法向量。

>>> import numpy as np

>>> np.cross((1,0,0), (0,1,0))

>>> array([0, 0, 1])

以右手系空间直角坐标系为例，假定 x o y平面是反射面，向量(1, 0, 0)和(0, 1, 0)——即 x x*x* 轴和 y y*y* 轴，它们叉乘的结果是(0, 0, 1)，也就是 z轴。

## 8. 最简单的着色器程序

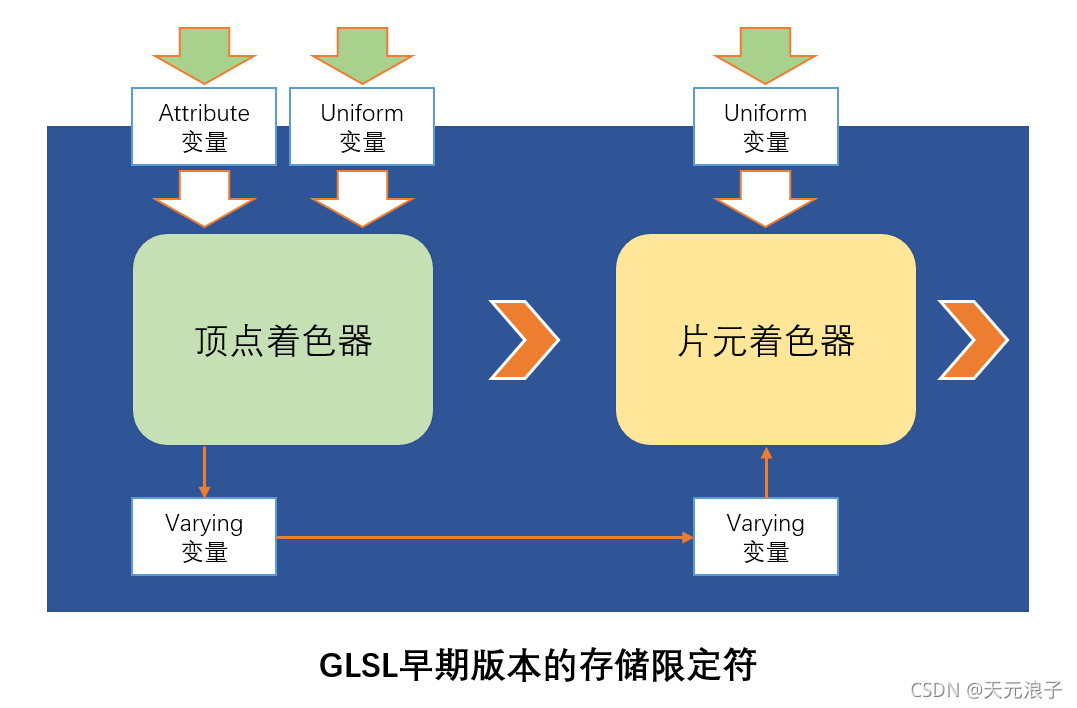
尽管OpenGL的渲染管线很长，有些环节是可选的，但有两个重要环节是不可或缺的：一是顶点渲染，通过变换、差值、滤波等将顶点变成图元形状；二是像素渲染，将图元形状像素化。对于固定管线而言，顶点渲染和像素渲染的流程是固定的、不可干预的，而可编程管线则允许开发者利用着色器自行设计渲染流程，赋予了开发者更大的灵活度和创造性。

顶点渲染需要顶点着色器，像素渲染需要片元着色器。着色器一般使用GLSL语言编写，经过编译生成着色器程序，运行在GPU上。**顶点着色器程序每次只处理一个顶点，同样的，片元着色器也是逐片元处理的**。自从我理解这句话之后，才真正理解了可编程管线的意义。

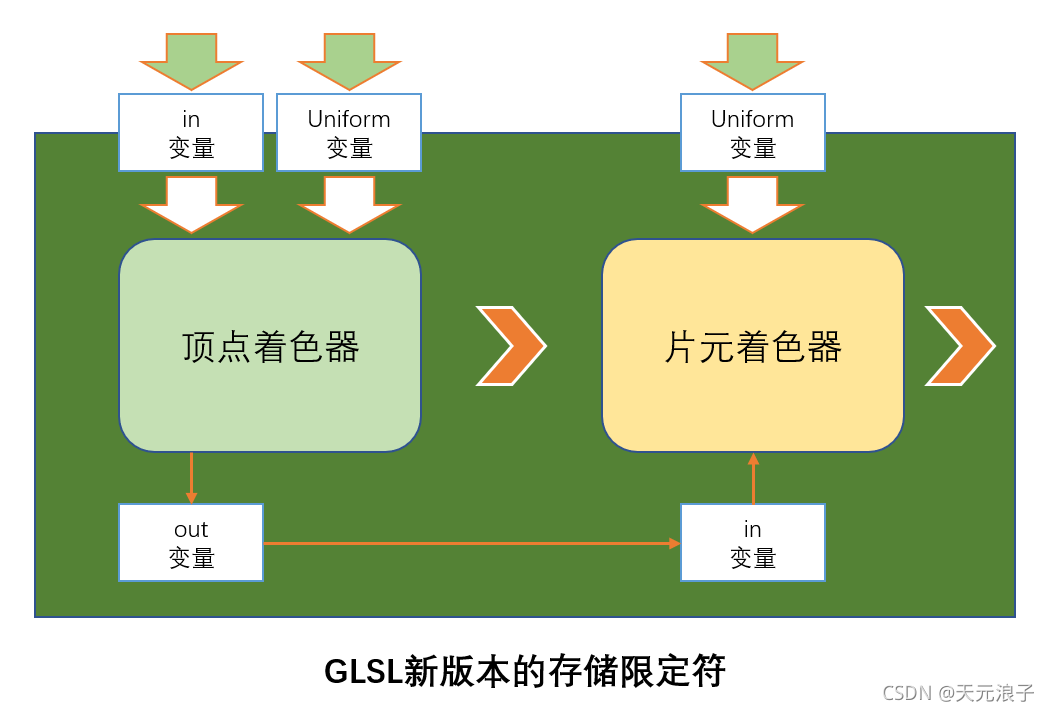
假如一个模型有1万个顶点，则顶点着色器程序需要运行1万次，每次输入的顶点数据（包括法向量、纹理坐标等）是各不相同的。但是也有在这1万运行中保持不变的数据，比如在该模型的此次渲染期间，投影矩阵、视点矩阵不会改变，大多数情况下，灯光参数也不会改变。

对于一个模型的一次渲染，在GLSL140版本之前，将顶点着色器程序每次运行时都会改变的数据称之为attribute，将每次运行保持不变的数据称之为uniform，将着色器之间顺序传递的数据称之为varying——这就是所谓的存储限定符。

我们可以向着色器程序传入attribute类型的数据和uniform类型的数据，但不能传入varying类型的数据。下图描述了attribute、uniform和varying的作用和用法。



也许是为了强化渲染管线数据的数据输入输出概念，自GLSL140版本开始，存储限定符attribute和varying不再被推荐使用，取而代之的是in和out。显然，in和out与attribute和varying并不是一一对应的关系。



了解了上述概念后，学习GLSL语言就非常容易了。GLSL语言不是本文的重点，有兴趣的读者可以参考我的写的[《OpenGL着色器语言GLSL资料汇编》](https://xufive.blog.csdn.net/article/details/121437277)一文。

**demo\_08.py**

#!/usr/bin/env python3

"""最简单的着色器程序"""

import numpy as np

from OpenGL.GL import \*

from OpenGL.GLUT import \*

from OpenGL.arrays import vbo

from OpenGL.GL import shaders

PROGRAM = None

VERTICES = None

COLORS = None

def prepare():

"""准备模型数据"""

global PROGRAM, VERTICES, COLORS

vshader\_src = """

#version 330 core

in vec4 a\_Position;

in vec4 a\_Color;

out vec4 v\_Color;

void main() {

gl\_Position = a\_Position; // gl\_Position是内置变量

v\_Color = a\_Color;

}

"""

fshader\_src = """

#version 330 core

in vec4 v\_Color;

void main() {

gl\_FragColor = v\_Color; // gl\_FragColor是内置变量

}

"""

vshader = shaders.compileShader(vshader\_src, GL\_VERTEX\_SHADER)

fshader = shaders.compileShader(fshader\_src, GL\_FRAGMENT\_SHADER)

PROGRAM = shaders.compileProgram(vshader, fshader)

VERTICES = vbo.VBO(np.array([[0, 1, 0], [-1, -1, 0], [1, -1, 0]], dtype=np.float32))

COLORS = vbo.VBO(np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]], dtype=np.float32))

def draw():

"""绘制模型"""

glClear(GL\_COLOR\_BUFFER\_BIT) # 清除缓冲区

glUseProgram(PROGRAM)

v\_loc = glGetAttribLocation(PROGRAM, 'a\_Position')

VERTICES.bind()

glVertexAttribPointer(v\_loc, 3, GL\_FLOAT, GL\_FALSE, 3\*4, VERTICES)

glEnableVertexAttribArray(v\_loc)

VERTICES.unbind()

c\_loc = glGetAttribLocation(PROGRAM, 'a\_Color')

COLORS.bind()

glVertexAttribPointer(c\_loc, 3, GL\_FLOAT, GL\_FALSE, 3\*4, COLORS)

glEnableVertexAttribArray(c\_loc)

COLORS.unbind()

glDrawArrays(GL\_TRIANGLES, 0, 3)

glUseProgram(0)

glFlush() # 执行缓冲区指令

if \_\_name\_\_ == "\_\_main\_\_":

glutInit() # 1. 初始化glut库

glutCreateWindow('OpenGL Demo') # 2. 创建glut窗口

prepare() # 3. 生成着色器程序、顶点数据集、颜色数据集

glutDisplayFunc(draw) # 4. 绑定模型绘制函数

glutMainLoop() # 5. 进入glut主循环

和本文开篇第一段代码绘制三角形一样，这段代码绘制了一个大小相同、颜色一致的三角形，不同之处在于第一段代码使用固定管线，这段代码使用了着色器。prepare函数中的顶点着色器源码和片元着色器源码堪称最简单的着色器程序了。

## 9. 着色器中的MVP矩阵

前面曾经说过，模型矩阵（Model Matrix）、视点矩阵（View Matrix）、投影矩阵（Projection Matrix）合称MVP矩阵，深刻理解和使用MVP矩阵是玩转OpenGL的必要条件。

**demo\_09.py**

#!/usr/bin/env python3

"""着色器中的MVP矩阵"""

import numpy as np

from PIL import Image

from OpenGL.GL import \*

from OpenGL.GLUT import \*

from OpenGL.arrays import vbo

from OpenGL.GL import shaders

from demo\_03 import BaseScene

class Scene(BaseScene):

"""由BaseScene派生的三维场景类"""

def \_\_init\_\_(self, \*\*kwds):

"""构造函数"""

BaseScene.\_\_init\_\_(self, \*\*kwds) # 调用基类构造函数

self.mmat = np.eye(4, dtype=np.float32) # 模型矩阵

self.vmat = np.eye(4, dtype=np.float32) # 视点矩阵

self.pmat = np.eye(4, dtype=np.float32) # 投影矩阵

def get\_vmat(self):

"""返回视点矩阵"""

camX, camY, camZ = self.cam

oecsX, oecsY, oecsZ = self.oecs

upX, upY, upZ = self.up

f = np.array([oecsX-camX, oecsY-camY, oecsZ-camZ], dtype=np.float64)

f /= np.linalg.norm(f)

s = np.array([f[1]\*upZ - f[2]\*upY, f[2]\*upX - f[0]\*upZ, f[0]\*upY - f[1]\*upX], dtype=np.float64)

s /= np.linalg.norm(s)

u = np.cross(s, f)

return np.array([

[s[0], u[0], -f[0], 0],

[s[1], u[1], -f[1], 0],

[s[2], u[2], -f[2], 0],

[- s[0]\*camX - s[1]\*camY - s[2]\*camZ,

- u[0]\*camX - u[1]\*camY - u[2]\*camZ,

f[0]\*camX + f[1]\*camY + f[2]\*camZ, 1]

], dtype=np.float32)

def get\_pmat(self):

"""返回投影矩阵"""

right = np.tan(np.radians(self.fovy/2)) \* self.near

left = -right

top = right/self.aspect

bottom = left/self.aspect

rw, rh, rd = 1/(right-left), 1/(top-bottom), 1/(self.far-self.near)

return np.array([

[2 \* self.near \* rw, 0, 0, 0],

[0, 2 \* self.near \* rh, 0, 0],

[(right+left) \* rw, (top+bottom) \* rh, -(self.far+self.near) \* rd, -1],

[0, 0, -2 \* self.near \* self.far \* rd, 0]

], dtype=np.float32)

def create\_texture\_2d(self, texture\_file):

"""创建纹理对象"""

im = np.array(Image.open(texture\_file))

im\_h, im\_w = im.shape[:2]

im\_mode = GL\_LUMINANCE if im.ndim == 2 else (GL\_RGB, GL\_RGBA)[im.shape[-1]-3]

tid = glGenTextures(1)

glBindTexture(GL\_TEXTURE\_2D, tid)

if (im.size/im\_h)%4 == 0:

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 4)

else:

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1)

glTexImage2D(GL\_TEXTURE\_2D, 0, im\_mode, im\_w, im\_h, 0, im\_mode, GL\_UNSIGNED\_BYTE, im)

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR)

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR)

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT)

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT)

glGenerateMipmap(GL\_TEXTURE\_2D)

glBindTexture(GL\_TEXTURE\_2D, 0)

return tid

def prepare(self):

"""准备模型数据"""

vshader\_src = """

#version 330 core

in vec4 a\_Position;

in vec2 a\_Texcoord;

uniform mat4 u\_ProjMatrix;

uniform mat4 u\_ViewMatrix;

uniform mat4 u\_ModelMatrix;

out vec2 v\_Texcoord;

void main() {

gl\_Position = u\_ProjMatrix \* u\_ViewMatrix \* u\_ModelMatrix \* a\_Position;

v\_Texcoord = a\_Texcoord;

}

"""

fshader\_src = """

#version 330 core

in vec2 v\_Texcoord;

uniform sampler2D u\_Texture;

void main() {

gl\_FragColor = texture2D(u\_Texture, v\_Texcoord);

}

"""

vs = np.array([

[-1, 1, 1], [ 1, 1, 1], [ 1, -1, 1], [-1, -1, 1],

[-1, 1, -1], [ 1, 1, -1], [ 1, -1, -1], [-1, -1, -1]

], dtype=np.float32)

indices = np.array([

0, 3, 2, 1, # v0-v1-v2-v3 (front)

4, 0, 1, 5, # v4-v5-v1-v0 (top)

3, 7, 6, 2, # v3-v2-v6-v7 (bottom)

7, 4, 5, 6, # v5-v4-v7-v6 (back)

1, 2, 6, 5, # v1-v5-v6-v2 (right)

4, 7, 3, 0 # v4-v0-v3-v7 (left)

], dtype=np.int32)

vertices = vs[indices]

texcoord = np.array([

[0,0], [0,1], [1,1], [1,0], [0,0], [0,1], [1,1], [1,0], [0,0], [0,1], [1,1], [1,0],

[0,0], [0,1], [1,1], [1,0], [0,0], [0,1], [1,1], [1,0], [0,0], [0,1], [1,1], [1,0]

], dtype=np.float32)

vshader = shaders.compileShader(vshader\_src, GL\_VERTEX\_SHADER)

fshader = shaders.compileShader(fshader\_src, GL\_FRAGMENT\_SHADER)

self.program = shaders.compileProgram(vshader, fshader)

self.vertices = vbo.VBO(vertices)

self.texcoord = vbo.VBO(texcoord)

self.texture = self.create\_texture\_2d('res/flower.jpg')

def draw(self):

"""绘制模型。可在派生类中重写此方法"""

glUseProgram(self.program)

loc = glGetAttribLocation(self.program, 'a\_Position')

self.vertices.bind()

glVertexAttribPointer(loc, 3, GL\_FLOAT, GL\_FALSE, 3\*4, self.vertices)

glEnableVertexAttribArray(loc)

self.vertices.unbind()

loc = glGetAttribLocation(self.program, 'a\_Texcoord')

self.texcoord.bind()

glVertexAttribPointer(loc, 2, GL\_FLOAT, GL\_FALSE, 2\*4, self.texcoord)

glEnableVertexAttribArray(loc)

self.texcoord.unbind()

loc = glGetUniformLocation(self.program, 'u\_ProjMatrix')

glUniformMatrix4fv(loc, 1, GL\_FALSE, self.get\_pmat(), None)

loc = glGetUniformLocation(self.program, 'u\_ViewMatrix')

glUniformMatrix4fv(loc, 1, GL\_FALSE, self.get\_vmat(), None)

loc = glGetUniformLocation(self.program, 'u\_ModelMatrix')

glUniformMatrix4fv(loc, 1, GL\_FALSE, self.mmat, None)

loc = glGetUniformLocation(self.program, 'u\_Texture')

glActiveTexture(GL\_TEXTURE0)

glBindTexture(GL\_TEXTURE\_2D, self.texture)

glUniform1i(loc, 0)

glDrawArrays(GL\_QUADS, 0, 24)

glUseProgram(0)

def render(self):

"""重绘事件函数"""

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT) # 清除屏幕及深度缓存

self.draw() # 绘制模型

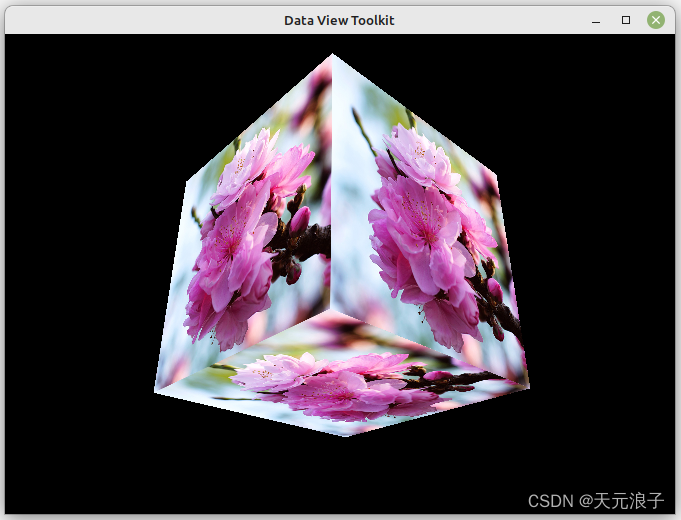
glutSwapBuffers() # 交换缓冲区

if \_\_name\_\_ == '\_\_main\_\_':

app = Scene(haxis='y')

app.show()

这段代码从BaseScene类派生了一个名为Scene的新类，新类增加了计算视点矩阵的方法get\_vmat和计算投影矩阵的方法get\_pmat，重写了prepare方法、render方法和draw方法。每次渲染时，draw方法都会根据当前状态重新计算V矩阵和P矩阵，与M矩阵（一个固定不变的4阶单位矩阵）一起被送到着色器程序



着色器中的MVP矩阵（demo\_09.py）

如果要对模型做旋转平移操作，只需要在每次渲染前更新Scene类的模型矩阵Scene.mmat即可。更进一步，如果绑定GLUT的idle事件，利用空闲时间触发重绘事件，还可以让模型或相机动起来。不过，计算模型的旋转、平移和缩放矩阵，以及让模型动起来，超出了本文的范围，有闲了我会另写一篇详细阐述。

## 10. 着色器中的漫反射、镜面反射和高光计算

物体表面反射的光线进入我们的眼睛，使得我们看见了这个物体。进入我们眼睛的光通量越大，物体就越清晰明亮。物体表面的反射又分为漫反射和镜面反射，二者都遵从光的反射定律。

如果物体表面比较粗糙，各处的漫反射光线都可能进入我们的眼睛，漫反射的强度了决定物体表面的亮度，我们用漫反射系数diffuse来表示漫反射的强度，0表示没有漫反射，1表示最强漫反射。

如果物体表面足够平滑，就会产生镜面反射。当我们的眼睛位于镜面反射的路径上且面对光线反射方向，就会在光线入射点处看到较强镜面反射光，其他位置则是相对较弱的漫反射光。镜面反射的强度决定了亮度区域的亮度，我们用镜面反射系数specular来表示镜面反射的强度，0表示没有镜面反射，1表示最强镜面反射。另外，相同的光照环境下，材质不同高光区域大小也不尽相同。我们用高光系数shiny来影响高光区域大小，高光系数大于0，数值越大，高光区域越小，一般超过2000以后就几乎看不到高光了。

另外，还要考虑材质的透光性，透光性决定了模型背面的亮度。我们用pellucid来表示材质的透光性，0表示没有不透光，1表示和正面亮度一致。

**demo\_10.py**

#!/usr/bin/env python3

"""着色器中的漫反射、镜面反射和高光计算"""

import numpy as np

from OpenGL.GL import \*

from OpenGL.GLUT import \*

from OpenGL.arrays import vbo

from OpenGL.GL import shaders

from demo\_09 import Scene

class App(Scene):

"""由BaseScene派生的3D应用程序类"""

def prepare(self):

"""准备模型数据"""

vshader\_src = """

#version 330 core

in vec4 a\_Position;

in vec3 a\_Normal;

in vec2 a\_Texcoord;

uniform mat4 u\_ProjMatrix;

uniform mat4 u\_ViewMatrix;

uniform mat4 u\_ModelMatrix;

uniform vec3 u\_CamPos;

out vec2 v\_Texcoord;

out vec3 v\_Normal;

out vec3 v\_CamDir;

void main() {

gl\_Position = u\_ProjMatrix \* u\_ViewMatrix \* u\_ModelMatrix \* a\_Position;

v\_Texcoord = a\_Texcoord;

mat4 NormalMatrix = transpose(inverse(u\_ModelMatrix)); // 法向量矩阵

v\_Normal = normalize(vec3(NormalMatrix \* vec4(a\_Normal, 1.0))); // 重新计算模型变换后的法向量

v\_CamDir = normalize(u\_CamPos - vec3(u\_ModelMatrix \* a\_Position)); // 从当前顶点指向相机的向量

}

"""

fshader\_src = """

#version 330 core

in vec2 v\_Texcoord;

in vec3 v\_Normal;

in vec3 v\_CamDir;

uniform sampler2D u\_Texture;

uniform vec3 u\_LightDir; // 定向光方向

uniform vec3 u\_LightColor; // 定向光颜色

uniform vec3 u\_AmbientColor; // 环境光颜色

uniform float u\_Shiny; // 高光系数，非负数，数值越大高光点越小

uniform float u\_Specular; // 镜面反射系数，0~1之间的浮点数，影响高光亮度

uniform float u\_Diffuse; // 漫反射系数，0~1之间的浮点数，影响表面亮度

uniform float u\_Pellucid; // 透光系数，0~1之间的浮点数，影响背面亮度

void main() {

vec3 lightDir = normalize(-u\_LightDir); // 光线向量取反后单位化

vec3 middleDir = normalize(v\_CamDir + lightDir); // 视线和光线的中间向量

vec4 color = texture2D(u\_Texture, v\_Texcoord);

float diffuseCos = u\_Diffuse \* max(0.0, dot(lightDir, v\_Normal)); // 光线向量和法向量的内积

float specularCos = u\_Specular \* max(0.0, dot(middleDir, v\_Normal)); // 中间向量和法向量内积

if (!gl\_FrontFacing)

diffuseCos \*= u\_Pellucid; // 背面受透光系数影响

if (diffuseCos == 0.0)

specularCos = 0.0;

else

specularCos = pow(specularCos, u\_Shiny);

vec3 scatteredLight = min(u\_AmbientColor + u\_LightColor \* diffuseCos, vec3(1.0)); // 散射光

vec3 reflectedLight = u\_LightColor \* specularCos; // 反射光

vec3 rgb = min(color.rgb \* (scatteredLight + reflectedLight), vec3(1.0));

gl\_FragColor = vec4(rgb, color.a);

}

"""

# 生成地球的顶点，东西方向和南北方向精度为2°

rows, cols, r = 90, 180, 1

gv, gu = np.mgrid[0.5\*np.pi:-0.5\*np.pi:complex(0,rows), 0:2\*np.pi:complex(0,cols)]

xs = r \* np.cos(gv)\*np.cos(gu)

ys = r \* np.cos(gv)\*np.sin(gu)

zs = r \* np.sin(gv)

vs = np.dstack((xs, ys, zs)).reshape(-1, 3)

vs = np.float32(vs)

# 生成三角面的索引

idx = np.arange(rows\*cols).reshape(rows, cols)

idx\_a, idx\_b, idx\_c, idx\_d = idx[:-1,:-1], idx[1:,:-1], idx[:-1, 1:], idx[1:,1:]

indices = np.int32(np.dstack((idx\_a, idx\_b, idx\_c, idx\_c, idx\_b, idx\_d)).ravel())

# 生成法向量

primitive = vs[indices]

a = primitive[::3]

b = primitive[1::3]

c = primitive[2::3]

normal = np.repeat(np.cross(b-a, c-a), 3, axis=0)

idx\_arg = np.argsort(indices)

rise = np.where(np.diff(indices[idx\_arg])==1)[0]+1

rise = np.hstack((0, rise, len(indices)))

tmp = np.zeros((rows\*cols, 3), dtype=np.float32)

for i in range(rows\*cols):

tmp[i] = np.sum(normal[idx\_arg[rise[i]:rise[i+1]]], axis=0)

normal = tmp.reshape(rows,cols,-1)

normal[:,0] += normal[:,-1]

normal[:,-1] = normal[:,0]

normal[0] = normal[0,0]

normal[-1] = normal[-1,0]

# 生成纹理坐标

u, v = np.linspace(0, 1, cols), np.linspace(0, 1, rows)

texcoord = np.float32(np.dstack(np.meshgrid(u, v)).reshape(-1, 2))

# 生成着色器程序

vshader = shaders.compileShader(vshader\_src, GL\_VERTEX\_SHADER)

fshader = shaders.compileShader(fshader\_src, GL\_FRAGMENT\_SHADER)

self.program = shaders.compileProgram(vshader, fshader)

# 创建VBO和纹理对象

self.vertices = vbo.VBO(vs)

self.normal = vbo.VBO(normal)

self.texcoord = vbo.VBO(texcoord)

self.indices = vbo.VBO(indices, target=GL\_ELEMENT\_ARRAY\_BUFFER)

self.n = len(indices)

self.texture = self.create\_texture\_2d('res/earth.jpg')

# 设置光照参数

self.light\_dir = np.array([-1, 1, 0], dtype=np.float32) # 光线照射方向

self.light\_color = np.array([1, 1, 1], dtype=np.float32) # 光线颜色

self.ambient = np.array([0.2, 0.2, 0.2], dtype=np.float32) # 环境光颜色

self.shiny = 50 # 高光系数

self.specular = 1.0 # 镜面反射系数

self.diffuse = 0.7 # 漫反射系数

self.pellucid = 0.5 # 透光度

def draw(self):

"""绘制模型。可在派生类中重写此方法"""

glUseProgram(self.program)

loc = glGetAttribLocation(self.program, 'a\_Position')

self.vertices.bind()

glVertexAttribPointer(loc, 3, GL\_FLOAT, GL\_FALSE, 3\*4, self.vertices)

glEnableVertexAttribArray(loc)

self.vertices.unbind()

loc = glGetAttribLocation(self.program, 'a\_Normal')

self.normal.bind()

glVertexAttribPointer(loc, 3, GL\_FLOAT, GL\_FALSE, 3\*4, self.normal)

glEnableVertexAttribArray(loc)

self.normal.unbind()

loc = glGetAttribLocation(self.program, 'a\_Texcoord')

self.texcoord.bind()

glVertexAttribPointer(loc, 2, GL\_FLOAT, GL\_FALSE, 2\*4, self.texcoord)

glEnableVertexAttribArray(loc)

self.texcoord.unbind()

loc = glGetUniformLocation(self.program, 'u\_ProjMatrix')

glUniformMatrix4fv(loc, 1, GL\_FALSE, self.get\_pmat(), None)

loc = glGetUniformLocation(self.program, 'u\_ViewMatrix')

glUniformMatrix4fv(loc, 1, GL\_FALSE, self.get\_vmat(), None)

loc = glGetUniformLocation(self.program, 'u\_ModelMatrix')

glUniformMatrix4fv(loc, 1, GL\_FALSE, self.mmat, None)

loc = glGetUniformLocation(self.program, 'u\_CamPos')

glUniform3f(loc, \*self.cam)

loc = glGetUniformLocation(self.program, 'u\_Texture')

glActiveTexture(GL\_TEXTURE0)

glBindTexture(GL\_TEXTURE\_2D, self.texture)

glUniform1i(loc, 0)

loc = glGetUniformLocation(self.program, 'u\_LightDir')

glUniform3f(loc, \*self.light\_dir)

loc = glGetUniformLocation(self.program, 'u\_LightColor')

glUniform3f(loc, \*self.light\_color)

loc = glGetUniformLocation(self.program, 'u\_AmbientColor')

glUniform3f(loc, \*self.ambient)

loc = glGetUniformLocation(self.program, 'u\_Shiny')

glUniform1f(loc, self.shiny)

loc = glGetUniformLocation(self.program, 'u\_Specular')

glUniform1f(loc, self.specular)

loc = glGetUniformLocation(self.program, 'u\_Diffuse')

glUniform1f(loc, self.diffuse)

loc = glGetUniformLocation(self.program, 'u\_Pellucid')

glUniform1f(loc, self.pellucid)

self.indices.bind()

glDrawElements(GL\_TRIANGLES, self.n, GL\_UNSIGNED\_INT, None)

self.indices.unbind()

glUseProgram(0)

def render(self):

"""重绘事件函数"""

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT) # 清除屏幕及深度缓存

self.draw() # 绘制模型

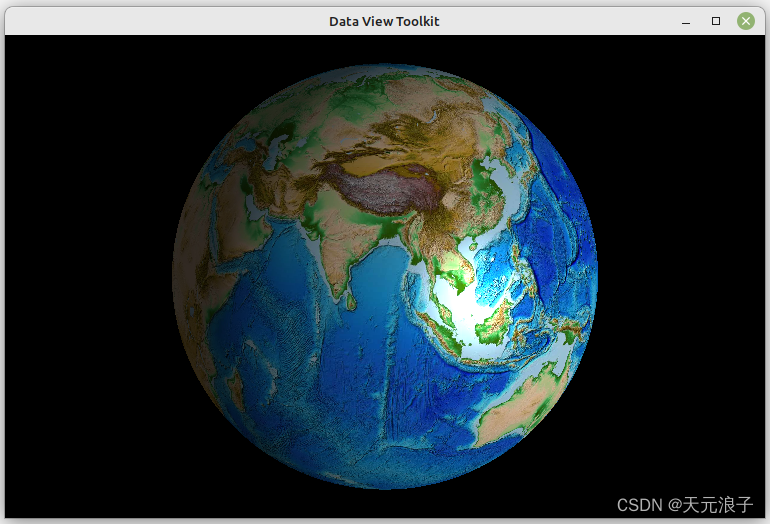
glutSwapBuffers() # 交换缓冲区

if \_\_name\_\_ == '\_\_main\_\_':

app = App(haxis='z')

app.show()

这段代码以 z 轴为高度轴，绘制了一个地球。平行光线（比如阳光）射向(-1,1,0)方向，配合适当的漫反射系数、镜面反射系数和高光系数，地球表面形成了一个高光区域。



着色器中的漫反射、镜面反射和高光计算（demo\_10.py）

着色器中反射强度、高光强度的计算以及光线调制，我已经在代码中做了详尽的说明。对于真正的程序员来说，代码本身就是最好的注释。懂得自然都懂，暂时不明白也没关系，多思考、多练习，慢慢就懂了。书读百遍，其意自现，说的就是这个意思。

**文**