**写给 python 程序员的 OpenGL 教程**

**文章目录**

* [1 预备知识](https://blog.csdn.net/xufive/article/details/86565130#1__2)
  + [1.1 坐标系](https://blog.csdn.net/xufive/article/details/86565130#11__10)
  + [1.2 投影](https://blog.csdn.net/xufive/article/details/86565130#12__24)
  + [1.3 视景体](https://blog.csdn.net/xufive/article/details/86565130#13__33)
  + [1.4 视口](https://blog.csdn.net/xufive/article/details/86565130#14__38)
  + [1.5 视点](https://blog.csdn.net/xufive/article/details/86565130#15__43)
  + [1.6 OpenGL 变换](https://blog.csdn.net/xufive/article/details/86565130#16_OpenGL__52)
* [2 安装 pyopengl](https://blog.csdn.net/xufive/article/details/86565130#2__pyopengl_61)
* [3 OpenGL 库及函数简介](https://blog.csdn.net/xufive/article/details/86565130#3_OpenGL__76)
  + [3.1 OpenGL 核心库 GL](https://blog.csdn.net/xufive/article/details/86565130#31_OpenGL__GL_87)
  + [3.2 OpenGL 实用库 GLU](https://blog.csdn.net/xufive/article/details/86565130#32_OpenGL__GLU_119)
  + [3.3 OpenGL 工具库 GLUT](https://blog.csdn.net/xufive/article/details/86565130#33_OpenGL__GLUT_140)
  + [3.4 Windows 专用库 WGL](https://blog.csdn.net/xufive/article/details/86565130#34_Windows__WGL_158)
* [4 开始 OpenGL 的奇幻之旅](https://blog.csdn.net/xufive/article/details/86565130#4__OpenGL__173)
  + [4.1 OpenGL 基本图形的绘制](https://blog.csdn.net/xufive/article/details/86565130#41_OpenGL__174)
    - [4.1.1 设置颜色](https://blog.csdn.net/xufive/article/details/86565130#411__175)
    - [4.1.2 设置顶点](https://blog.csdn.net/xufive/article/details/86565130#412__190)
    - [4.1.3 绘制基本图形](https://blog.csdn.net/xufive/article/details/86565130#413__197)
  + [4.2 第一个 OpenGL 程序](https://blog.csdn.net/xufive/article/details/86565130#42__OpenGL__214)
  + [4.3 设置初始显示模式](https://blog.csdn.net/xufive/article/details/86565130#43__297)
  + [4.4 初始化画布](https://blog.csdn.net/xufive/article/details/86565130#44__316)
  + [4.5 清除屏幕及深度缓存](https://blog.csdn.net/xufive/article/details/86565130#45__325)
  + [4.5 设置投影](https://blog.csdn.net/xufive/article/details/86565130#45__330)
  + [4.6 设置视点](https://blog.csdn.net/xufive/article/details/86565130#46__351)
  + [4.7 设置视口](https://blog.csdn.net/xufive/article/details/86565130#47__361)
  + [4.8 设置模型视图](https://blog.csdn.net/xufive/article/details/86565130#48__367)
  + [4.9 捕捉鼠标事件、键盘事件和窗口事件](https://blog.csdn.net/xufive/article/details/86565130#49__375)
  + [4.10 综合应用](https://blog.csdn.net/xufive/article/details/86565130#410__416)
  + [4.11 小结](https://blog.csdn.net/xufive/article/details/86565130#411__653)
* [5 加速渲染](https://blog.csdn.net/xufive/article/details/86565130#5__656)
  + [5.1 创建顶点缓冲区对象（VBO）](https://blog.csdn.net/xufive/article/details/86565130#51_VBO_667)
  + [5.2 分离顶点混合数组](https://blog.csdn.net/xufive/article/details/86565130#52__722)
  + [5.3 使用顶点缓冲区对象（VBO）](https://blog.csdn.net/xufive/article/details/86565130#53_VBO_751)
* [6 致谢](https://blog.csdn.net/xufive/article/details/86565130#6__762)

**1 预备知识**

[OpenGL](https://so.csdn.net/so/search?q=OpenGL&spm=1001.2101.3001.7020) 是 Open Graphics Library 的简写，意为“开放式图形库”，是用于渲染 2D、3D 矢量图形的跨语言、跨平台的应用程序编程接口（API）。OpenGL 不是一个独立的平台，因此，它需要借助于一种编程语言才能被使用。C / C++ / python / java 都可以很好支持 OpengGL，我当然习惯性选择 python 语言。

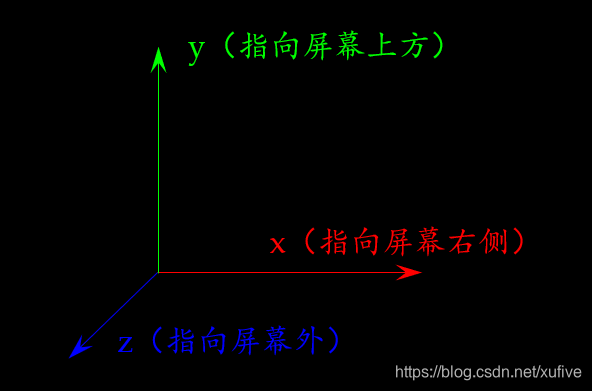
如果读者是 python 程序员，并且了解 [numpy](https://so.csdn.net/so/search?q=numpy&spm=1001.2101.3001.7020)，接下来的阅读应该不会有任何障碍；否则，我建议先花半小时学习一下 python 语言。关于 numpy，可以参考我的另一篇博文[《数学建模三剑客MSN》](https://blog.csdn.net/xufive/article/details/52449255)。事实上，我觉得 python 语言近乎于自然语言，只要读者是程序员，即便不熟悉 python，读起来也不会有多大问题。

另外，读者也不必担心数学问题。使用 OpenGL 不需要具备多么高深的数学水平，只要能辅导初中学生的数学作业，就足够用了。

**1.1 坐标系**

在 OpenGL 的世界里，有各式各样的坐标系。随着对 OpenGL 概念的理解，我们至少会接触到六种坐标系，而初始只需要了解其中的三个就足够用了（第一次阅读这段话的时候，只需要了解世界坐标系就可以了）。

* 世界坐标系（World Coordinates）

世界坐标系是右手坐标系，以屏幕中心为原点(0, 0, 0)，且是始终不变的。 

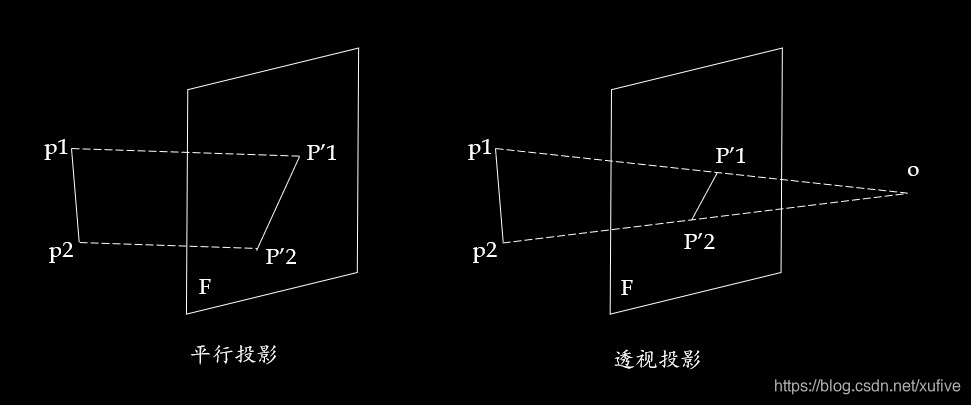
* 视点坐标系（Eye or Camera Coordinates）

视点坐标是以视点为原点，以视线的方向为Z+轴正方向的坐标系。OpenGL 管道会将世界坐标先变换到视点坐标，然后进行裁剪，只有在视线范围（视景体）之内的场景才会进入下一阶段的计算。

* 屏幕坐标系（Window or Screen Coordinates）

OpenGL 的重要功能之一就是将三维的世界坐标经过变换、投影等计算，最终算出它在显示设备上对应的位置，这个位置就称为设备坐标。在屏幕、打印机等设备上的坐标是二维坐标。值得一提的是，OpenGL 可以只使用设备的一部分进行绘制，这个部分称为视区或视口（viewport）。投影得到的是视区内的坐标(投影坐标)，从投影坐标到设备坐标的计算过程就是设备变换了。

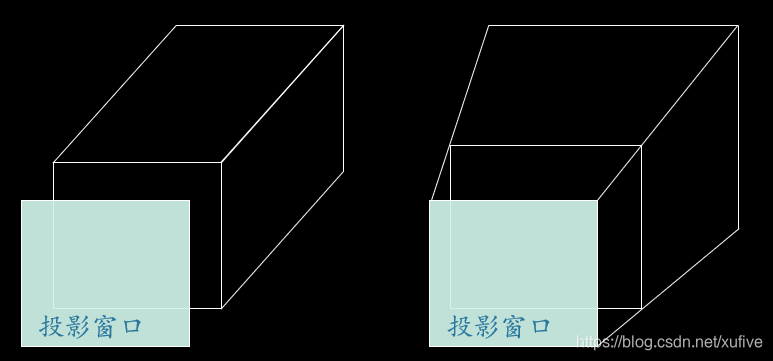
**1.2 投影**

三维场景中的物体最终都会显示在类似屏幕这样的二维观察平面上。将三维物体变为二维图形的变换成为投影变换。最常用的投影有两种：平行投影和透视投影。如下图所示，F 是投影面，p1p2 为三维空间中的一条直线，p’1 和 p’2 分别是 p1 和 p2 在 F 上的投影，虚线表示投影线，O 为投影中心。  


* 平行投影  
  这里所说的平行投影，特指正交平行投影——投影线垂直于投影面。将一个三维点 (x,y,z) 正交平行投影到 xoy 平面上，则投影点坐标为 (x,y,0)。由于平行投影丢弃了深度信息，所以无法产生真实感，但可以保持物体之间相对大小关系不变。
* 透视投影  
  透视投影将投影面置于观察点和投影对象之间，距离观察者越远的物体，投影尺寸越小，投影效果具有真实感，常用于游戏和仿真领域。

**1.3 视景体**

无论是平行投影还是透视投影，投影成像都是在投影面上——我们可以把投影面理解成显示屏幕。世界坐标系描述的三维空间是无限的，投影平面是无限的，但（我们能够看到的）屏幕面积总是有限的，因此在投影变换时，通常只处理能够显示在屏幕上的那一部分三维空间。从无限三维空间中裁切出来的可以显示在屏幕上的部分三维空间，我们称之为视景体。视景体有六个面，分别是左右上下和前后面。

对于平行投影而言，视景体是一个矩形平行六面体；对于透视投影来说，视景体是一个棱台。理解这一点并不难：因为越远处的物体在投影窗口的透视投影越小，也就意味着填满投影窗口需要更大的体量，视景体自然就变成了棱台。  


**1.4 视口**

对于平行投影而言，视口就是由视景体的左右上下四个面围成的矩形，对于透视投影来说，视口就是视景体的前截面在投影窗口上的透视投影。

视口是 OpenGL 中比较重要的概念，现阶段可以简单理解成屏幕（或其他输出设备）。事实上，视口和屏幕是相关但又不相同的，屏幕有固定的宽高比，而视口大小可以由用户自行定义。通常，为了适应不同宽高比的屏幕，在设置视口时，会根据屏幕宽高比调整视景体（增加宽度或高度）。

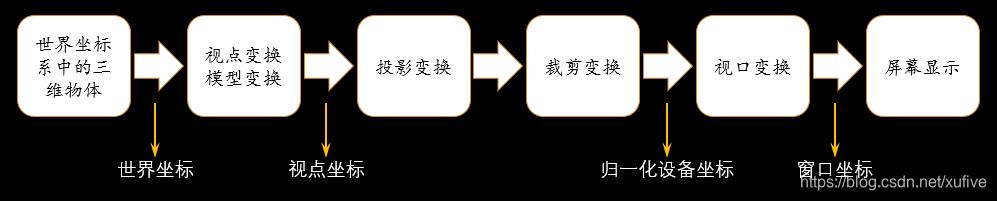
**1.5 视点**

现实生活中，人们看到的三维空间物体的样子取决于观察者站在什么角度去看。这里面包含着三个概念：

* 观察者的位置：眼睛在哪儿？
* 观察者的姿势：站立还是倒立？左侧卧还是右侧卧？
* 观察对象：眼睛盯着哪里？

对应在 OpenGL 中，也有同样的概念，即视点的位置、瞄准方向的参考点，以及（向上的）方向。

**1.6 OpenGL 变换**

下图是三维图形的显示流程。世界坐标系中的三维物体经过视点变换和一系列几何变换（平移、旋转、缩放）之后，坐标系变换为视点坐标系；经过投影和裁剪之后，坐标系变换为归一化设备坐标系；最后经过视口变换显示在屏幕上，相应地，坐标系变成了窗口坐标系。  


* 视点变换：相当于设置视点的位置和方向
* 模型变换：包括平移、旋转、缩放等三种类型
* 裁剪变换：根据视景体定义的六个面（和附加裁剪面）对三维空间裁剪
* 视口变换：将视景体内投影的物体显示在二维的视口平面上

**2 安装 pyopengl**

如果想当然地使用 pip 如下所示安装，可能会有一些麻烦。

pip install pyopengl

当我这样安装之后，运行 OpenGL 代码，得到了这样的错误信息：

NullFunctionError: Attempt to call an undefined function glutInit, check for bool(glutInit) before calling

据说这是因为glut依赖的freeglut.dll默认安装的是32位版本的，而我的操作系统是64位的。建议点击[这里](https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopengl)下载适合自己的版本，直接安装.whl文件。我是这样安装的：

pip install PyOpenGL-3.1.3b2-cp37-cp37m-win\_amd64.whl

**3 OpenGL 库及函数简介**

我第一次接触 OpenGL 的 GL / GLU / GLUT 的时候，一下就被这些长得像孪生兄弟的库名字给整懵圈了，要不是内心强大，也许就跟 OpenGL 说再见了。时间久了才发现，OpenGL 的库及函数命名规则非常合理，便于查找、记忆。

OpenGL函数的命名格式如下：

<库前缀><根命令><可选的参数个数><可选的参数类型>

常见的库前缀有 gl、glu、glut、aux、wgl、glx、agl 等。库前缀表示该函数属于 OpenGL 哪一个开发库。从函数名后面中还可以看出需要多少个参数以及参数的类型。I 代表 int 型，f 代表 float 型，d 代表 double 型，u 代表无符号整型。例如 glColor3f() 表示了该函数属于gl库，参数是三个浮点数。

OpenGL 函数库相关的 API 有核心库(gl)、实用库(glu)、实用工具库(glut)、辅助库(aux)、窗口库(glx、agl、wgl)和扩展函数库等。gl是核心，glu是对gl的部分封装。glut是为跨平台的OpenGL程序的工具包，比aux功能强大。glx、agl、wgl 是针对不同窗口系统的函数。扩展函数库是硬件厂商为实现硬件更新利用OpenGL的扩展机制开发的函数。本文仅对常用的四个库做简单介绍。

**3.1 OpenGL 核心库 GL**

核心库包含有115个函数，函数名的前缀为gl。这部分函数用于常规的、核心的图形处理。此函数由gl.dll来负责解释执行。由于许多函数可以接收不同数以下几类。据类型的参数，因此派生出来的函数原形多达300多个。核心库中的函数主要可以分为以下几类函数：

* 绘制基本几何图元的函数：  
  glBegain()、glEnd()、glNormal\*()、glVertex\*()
* 矩阵操作、几何变换和投影变换的函数：  
  如矩阵入栈函数glPushMatrix()，矩阵出栈函数glPopMatrix()，装载矩阵函数glLoadMatrix()，矩阵相乘函数glMultMatrix()，当前矩阵函数glMatrixMode()和矩阵标准化函数glLoadIdentity()，几何变换函数glTranslate\*()、glRotate\*()和glScale\*()，投影变换函数glOrtho()、glFrustum()和视口变换函数glViewport()
* 颜色、光照和材质的函数：  
  如设置颜色模式函数glColor\*()、glIndex\*()，设置光照效果的函数glLight\*() 、glLightModel\*()和设置材质效果函数glMaterial()
* 显示列表函数：  
  主要有创建、结束、生成、删除和调用显示列表的函数glNewList()、glEndList()、glGenLists()、glCallList()和glDeleteLists()
* 纹理映射函数：  
  主要有一维纹理函数glTexImage1D()、二维纹理函数glTexImage2D()、设置纹理参数、纹理环境和纹理坐标的函数glTexParameter\*()、glTexEnv\*()和glTetCoord\*()
* 特殊效果函数：  
  融合函数glBlendFunc()、反走样函数glHint()和雾化效果glFog\*()
* 光栅化、象素操作函数：  
  如象素位置glRasterPos\*()、线型宽度glLineWidth()、多边形绘制模式glPolygonMode()，读取象素glReadPixel()、复制象素glCopyPixel()
* 选择与反馈函数：  
  主要有渲染模式glRenderMode()、选择缓冲区glSelectBuffer()和反馈缓冲区glFeedbackBuffer()
* 曲线与曲面的绘制函数：  
  生成曲线或曲面的函数glMap\*()、glMapGrid\*()，求值器的函数glEvalCoord\*() glEvalMesh\*()
* 状态设置与查询函数：  
  glGet\*()、glEnable()、glGetError()

**3.2 OpenGL 实用库 GLU**

包含有43个函数，函数名的前缀为glu。OpenGL提供了强大的但是为数不多的绘图命令，所有较复杂的绘图都必须从点、线、面开始。Glu 为了减轻繁重的编程工作，封装了OpenGL函数，Glu函数通过调用核心库的函数，为开发者提供相对简单的用法，实现一些较为复杂的操作。此函数由glu.dll来负责解释执行。OpenGL中的核心库和实用库可以在所有的OpenGL平台上运行。主要包括了以下几种：

* 辅助纹理贴图函数：  
  gluScaleImage() 、gluBuild1Dmipmaps()、gluBuild2Dmipmaps()
* 坐标转换和投影变换函数：  
  定义投影方式函数gluPerspective()、gluOrtho2D() 、gluLookAt()，拾取投影视景体函数gluPickMatrix()，投影矩阵计算gluProject()和gluUnProject()
* 多边形镶嵌工具：  
  gluNewTess()、gluDeleteTess()、gluTessCallback()、gluBeginPolygon()、gluTessVertex()、gluNextContour()、gluEndPolygon()
* 二次曲面绘制工具：  
  主要有绘制球面、锥面、柱面、圆环面gluNewQuadric()、gluSphere()、gluCylinder()、gluDisk()、gluPartialDisk()、gluDeleteQuadric()
* 非均匀有理B样条绘制工具：  
  主要用来定义和绘制Nurbs曲线和曲面，包括gluNewNurbsRenderer()、gluNurbsCurve()、gluBeginSurface()、gluEndSurface()、gluBeginCurve()、gluNurbsProperty()
* 错误反馈工具：  
  获取出错信息的字符串gluErrorString()

**3.3 OpenGL 工具库 GLUT**

包含大约30多个函数，函数名前缀为glut。glut是不依赖于窗口平台的OpenGL工具包，由Mark KLilgrad在SGI编写（现在在Nvidia），目的是隐藏不同窗口平台API的复杂度。函数以glut开头，它们作为aux库功能更强的替代品，提供更为复杂的绘制功能，此函数由glut.dll来负责解释执行。由于glut中的窗口管理函数是不依赖于运行环境的，因此OpenGL中的工具库可以在X-Window, Windows NT, OS/2等系统下运行，特别适合于开发不需要复杂界面的OpenGL示例程序。对于有经验的程序员来说，一般先用glut理顺3D图形代码，然后再集成为完整的应用程序。这部分函数主要包括：

* 窗口操作函数：  
  窗口初始化、窗口大小、窗口位置函数等 glutInit()、glutInitDisplayMode()、glutInitWindowSize()、glutInitWindowPosition()
* 回调函数：  
  响应刷新消息、键盘消息、鼠标消息、定时器函数 GlutDisplayFunc()、glutPostRedisplay()、glutReshapeFunc()、glutTimerFunc()、glutKeyboardFunc()、glutMouseFunc()
* 创建复杂的三维物体：  
  这些和aux库的函数功能相同
* 菜单函数：  
  创建添加菜单的函数 GlutCreateMenu()、glutSetMenu()、glutAddMenuEntry()、glutAddSubMenu() 和 glutAttachMenu()
* 程序运行函数：  
  glutMainLoop()

**3.4 Windows 专用库 WGL**

针对windows平台的扩展，包含有16个函数，函数名前缀为wgl。这部分函数主要用于连接OpenGL和Windows ，以弥补OpenGL在文本方面的不足。 Windows专用库只能用于Windows环境中。这类函数主要包括以下几类：

* 绘图上下文相关函数：  
  wglCreateContext()、wglDeleteContext()、wglGetCurrentContent()、wglGetCurrentDC()、wglDeleteContent()
* 文字和文本处理函数：  
  wglUseFontBitmaps()、wglUseFontOutlines()
* 覆盖层、地层和主平面层处理函数：  
  wglCopyContext()、wglCreateLayerPlane()、wglDescribeLayerPlane()、wglReakizeLayerPlatte()
* 其他函数：  
  wglShareLists()、wglGetProcAddress()

**4 开始 OpenGL 的奇幻之旅**

**4.1 OpenGL 基本图形的绘制**

**4.1.1 设置颜色**

设置颜色的函数有几十个，都是以 glColor 开头，后面跟着参数个数和参数类型。参数可以是 0 到 255 之间的无符号整数，也可以是 0 到 1 之间的浮点数。三个参数分别表示 RGB 分量，第四个参数表示透明度（其实叫不透明度更恰当）。以下最常用的两个设置颜色的方法：

glColor3f(1.0，0.0，0.0) # 设置当前颜色为红色

glColor4f(0.0，1.0，1.0，1.0) # 设置当前颜色为青色，不透明度

glColor3ub(0, 0, 255) # 设置当前颜色为蓝色

glColor 也支持将三个或四个参数以向量方式传递，例如：

glColor3fv([0.0，1.0，0.0]) # 设置当前颜色为绿色

特别提示：OpenGL 是使用状态机模式，颜色是一个状态变量，设置颜色就是改变这个状态变量并一直生效，直到再次调用设置颜色的函数。除了颜色，OpenGL 还有很多的状态变量或模式。在任何时间，都可以查询每个状态变量的当前值，还可以用 glPushAttrib() 或 glPushClientAttrib() 把状态变量的集合保存起来，必要的时候，再用 glPopAttrib() 或 glPopClientAttrib() 恢复状态变量。

**4.1.2 设置顶点**

顶点（vertex）是 OpengGL 中非常重要的概念，描述线段、多边形都离不开顶点。和设置颜色类似，设置顶点的函数也有几十个，都是以 glVertex 开头，后面跟着参数个数和参数类型，同样也支持将多个以向量方式传递。 两个参数的话，分别表示 xy 坐标，三个参数则分别表示 xyz 坐标。如有第四个参数，则表示该点的齐次坐标 w；否则，默认 w=1。至于什么是齐次坐标，显然超出了初中数学的范畴，在此不做探讨。

glVertex2f(1.0，0.5) # xoy平面上的点，z=0

glVertex3f(0.5，1.0，0.0) # 三维空间中的点

**4.1.3 绘制基本图形**

仅仅设置颜色和顶点，并不能画出来什么。我们可以在任何时候改变颜色，但所有的顶点设置，都必须包含在 glBegin() 和 glEnd() 之间，而 glBegin() 的参数则指定了将这些顶点画成什么。以下是 glBegin() 可能的参数选项：

| **参数** | **说明** |
| --- | --- |
| GL\_POINTS | 绘制一个或多个顶点 |
| GL\_LINES | 绘制线段 |
| GL\_LINE\_STRIP | 绘制连续线段 |
| GL\_LINE\_LOOP | 绘制闭合的线段 |
| GL\_POLYGON | 绘制多边形 |
| GL\_TRIANGLES | 绘制一个或多个三角形 |
| GL\_TRIANGLE\_STRIP | 绘制连续三角形 |
| GL\_TRIANGLE\_FAN | 绘制多个三角形组成的扇形 |
| GL\_QUADS | 绘制一个或多个四边形 |
| GL\_QUAD\_STRIP | 绘制连续四边形 |

**4.2 第一个 OpenGL 程序**

通常，我们使用工具库（GLUT）创建 OpenGL 应用程序。为啥不用 GL 或者 GLU 库呢？画画之前总得先有一块画布吧，不能直接拿起画笔就开画。前文说过，工具库主要提供窗口相关的函数，有了窗口，就相当于有了画布，而核心库和实用库，就好比各式各样的画笔、颜料。使用工具库（GLUT）创建 OpenGL 应用程序只需要四步（当然，前提是你需要先准备好绘图函数，并给它取一个合适的名字）：

1. 初始化glut库
2. 创建glut窗口
3. 注册绘图的回调函数
4. 进入glut主循环

OK，铺垫了这么多之后，我们终于开始第一个 OpenGL 应用程序了：绘制三维空间的世界坐标系，在坐标原点的后方（z轴的负半区）画一个三角形。代码如下：

# -\*- coding: utf-8 -\*-

# -------------------------------------------

# quidam\_01.py 三维空间的世界坐标系和三角形

# -------------------------------------------

from OpenGL.GL import \*

from OpenGL.GLUT import \*

def draw():

# ---------------------------------------------------------------

glBegin(GL\_LINES) # 开始绘制线段（世界坐标系）

# 以红色绘制x轴

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.8, 0.0, 0.0) # 设置x轴顶点（x轴负方向）

glVertex3f(0.8, 0.0, 0.0) # 设置x轴顶点（x轴正方向）

# 以绿色绘制y轴

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.0, -0.8, 0.0) # 设置y轴顶点（y轴负方向）

glVertex3f(0.0, 0.8, 0.0) # 设置y轴顶点（y轴正方向）

# 以蓝色绘制z轴

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, 0.0, -0.8) # 设置z轴顶点（z轴负方向）

glVertex3f(0.0, 0.0, 0.8) # 设置z轴顶点（z轴正方向）

glEnd() # 结束绘制线段

# ---------------------------------------------------------------

glBegin(GL\_TRIANGLES) # 开始绘制三角形（z轴负半区）

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.5, -0.366, -0.5) # 设置三角形顶点

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.5, -0.366, -0.5) # 设置三角形顶点

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, 0.5, -0.5) # 设置三角形顶点

glEnd() # 结束绘制三角形

# ---------------------------------------------------------------

glFlush() # 清空缓冲区，将指令送往硬件立即执行

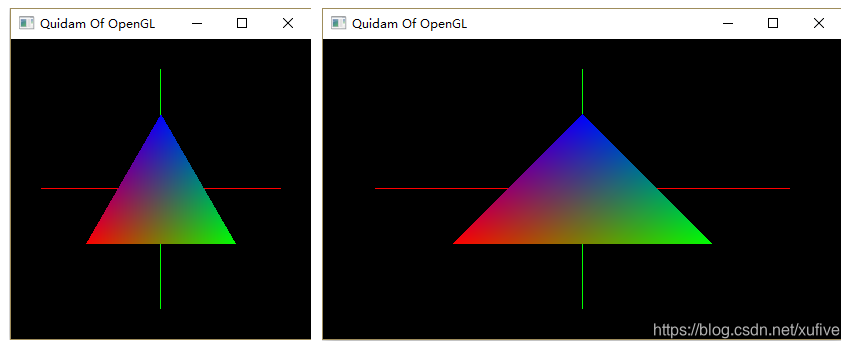
if \_\_name\_\_ == "\_\_main\_\_":

glutInit() # 1. 初始化glut库

glutCreateWindow('Quidam Of OpenGL') # 2. 创建glut窗口

glutDisplayFunc(draw) # 3. 注册回调函数draw()

glutMainLoop() # 4. 进入glut主循环

运行代码，我这里显示结果如下面左图所示。如果尝试运行这段代码出错的话，我猜应该是 pyopengl 安装出现了问题，建议返回到前面重读 pyopengl 的安装。  
  
短暂的激动之后，你可能会尝试画一些其他的线段，变换颜色或者透明度，甚至绘制多边形。很快你会发现，我们的第一个程序有很多问题，比如：

1. 窗口的标题不能使用中文，否则会显示乱码
2. 窗口的初始大小和位置无法改变
3. 改变窗口的宽高比，三角形宽高比也会改变（如上面右图所示）
4. 三角形不应该遮挡坐标轴
5. 改变颜色的透明度无效
6. 不能缩放旋转

没关系，除了第1个问题我不知道怎么解决（貌似无解），其他问题都不是事儿。和我们的代码相比，一个真正实用的 OpenGL 程序，还有许多工作要做：

* 设置初始显示模式
* 初始化画布
* 绘图函数里面需要增加：
  + 清除屏幕及深度缓存
  + 投影设置
  + 模型试图设置
* 绑定鼠标键盘的事件函数

**4.3 设置初始显示模式**

初始化 glut 库的时候，我们一般都要用 glutInitDisplayMode() 来设置初始的显示模式，它的参数可以是下表中参数的组合。

| **参数** | **说明** |
| --- | --- |
| GLUT\_RGB | 指定RGB颜色模式的窗口 |
| GLUT\_RGBA | 指定RGBA 颜色模式的窗口 |
| GLUT\_INDEX | 指定颜色索引模式的窗口 |
| GLUT\_SINGLE | 指定单缓存窗口 |
| GLUT\_DOUBLE | 指定双缓存窗口 |
| GLUT\_ACCUM | 窗口使用累加缓存 |
| GLUT\_ALPHA | 窗口的颜色分量包含 alpha 值 |
| GLUT\_DEPTH | 窗口使用深度缓存 |
| GLUT\_STENCIL | 窗口使用模板缓存 |
| GLUT\_MULTISAMPLE | 指定支持多样本功能的窗口 |
| GLUT\_STEREO | 指定立体窗口 |
| GLUT\_LUMINANCE | 窗口使用亮度颜色模型 |

使用双缓存窗口，可以避免重绘时产生抖动的感觉。我一般选择 GLUT\_DOUBLE | GLUT\_ALPHA | GLUT\_DEPTH 作为参数来设置初始的显示模式。

**4.4 初始化画布**

开始绘图之前，需要对画布做一些初始化工作，这些工作只需要做一次。比如：

glClearColor(0.0, 0.0, 0.0, 1.0) # 设置画布背景色。注意：这里必须是4个参数

glEnable(GL\_DEPTH\_TEST) # 开启深度测试，实现遮挡关系

glDepthFunc(GL\_LEQUAL) # 设置深度测试函数（GL\_LEQUAL只是选项之一）

如有必要，还可以开启失真校正（反走样）、开启表面剔除等。

**4.5 清除屏幕及深度缓存**

每次重绘之前，需要先清除屏幕及深度缓存。这项操作一般放在绘图函数的开头。

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

**4.5 设置投影**

投影设置也是每次重绘都需要的步骤之一。glOrtho() 用来设置平行投影，glFrustum() 用来设置透视投影。这两个函数的参数相同，都是视景体的 left / right / bottom / top / near / far 六个面。

**视景体的 left / right / bottom / top 四个面围成的矩形，就是视口。near 就是投影面，其值是投影面距离视点的距离，far 是视景体的后截面，其值是后截面距离视点的距离。far 和 near 的差值，就是视景体的深度。视点和视景体的相对位置关系是固定的，视点移动时，视景体也随之移动。**

我个人认为，视景体是 OpengGL 最重要、最核心的概念，它和视口、视点、投影面、缩放、漫游等概念密切关联。只有正确理解了视景体，才能正确设置它的六个参数，才能呈现出我们期望的效果。

为了在窗口宽高比改变时，绘制的对象仍然保持固定的宽高比，一般在做投影变换时，需要根据窗口的宽高比适当调整视景体的 left / right 或者 bottom / top 参数。

假设 view 是视景体，width 和 height 是窗口的宽度和高度，在投影变换之前，需要先声明是对投影矩阵的操作，并将投影矩阵单位化：

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

if width > height:

k = width / height

glFrustum(view [0]\*k, view [1]\*k, view [2], view [3], view [4], view [5])

else:

k = height / width

glFrustum(view [0], view [1], view [2]\*k, view [3]\*k, view [4], view [5])

**4.6 设置视点**

视点是和视景体关联的概念。设置视点需要考虑眼睛在哪儿、看哪儿、头顶朝哪儿，分别对应着eye, lookat 和 eye\_up 三个向量。

gluLookAt(

eye[0], eye[1], eye[2],

look\_at[0], look\_at[1], look\_at[2],

eye\_up[0], eye\_up[1], eye\_up[2]

)

**4.7 设置视口**

视口也是和视景体关联的概念，相对简单一点。

glViewport(0, 0, width, height)

**4.8 设置模型视图**

模型平移、旋转、缩放等几何变换，需要切换到模型矩阵：

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

glScale(1.0, 1.0, 1.0)

**4.9 捕捉鼠标事件、键盘事件和窗口事件**

GLUT 库提供了几个函数帮我们捕捉鼠标事件、键盘事件和窗口事件：

1. glutMouseFunc()  
   该函数捕捉鼠标点击和滚轮操作，返回4个参数给被绑定的事件函数：键（左键/右键/中键/滚轮上/滚轮下）、状态（1/0）、x坐标、y坐标
2. glutMotionFunc()  
   该函数捕捉有一个鼠标键被按下时的鼠标移动给被绑定的事件函数，返回2个参数：x坐标、y坐标
3. glutPassiveMotionFunc()  
   该函数捕捉鼠标移动，返回2个参数给被绑定的事件函数：x坐标、y坐标
4. glutEntryFunc()  
   该函数捕捉鼠标离开或进入窗口区域，返回1个参数给被绑定的事件函数：GLUT\_LEFT 或者 GLUT\_ENTERED
5. glutKeyboardFunc(keydown)  
   该函数捕捉键盘按键被按下，返回3个参数给被绑定的事件函数：被按下的键，x坐标、y坐标
6. glutReshapeFunc()  
   该函数捕捉窗口被改变大小，返回2个参数给被绑定的事件函数：窗口宽度、窗口高度

如果我们需要捕捉这些事件，只需要定义事件函数，注册相应的函数就行：

def reshape(width, height):

pass

def mouseclick(button, state, x, y):

pass

def mousemotion(x, y):

pass

def keydown(key, x, y):

pass

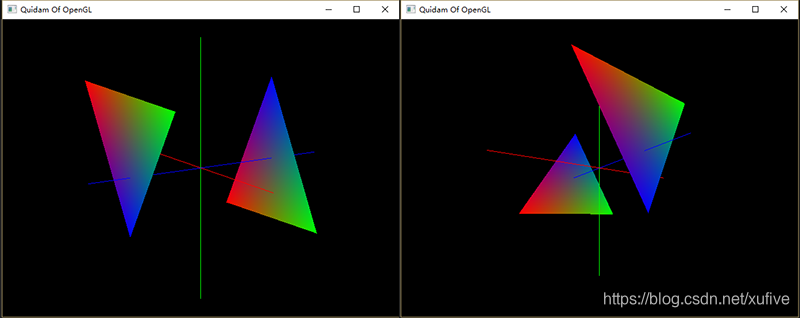
glutReshapeFunc(reshape) # 注册响应窗口改变的函数reshape()

glutMouseFunc(mouseclick) # 注册响应鼠标点击的函数mouseclick()

glutMotionFunc(mousemotion) # 注册响应鼠标拖拽的函数mousemotion()

glutKeyboardFunc(keydown) # 注册键盘输入的函数keydown()

**4.10 综合应用**

是时候把我们上面讲的这些东西完整的演示一下了。下面的代码还是画了世界坐标系，并在原点前后各画了一个三角形。鼠标可以拖拽视点绕参考点旋转（二者距离保持不变），滚轮可以缩放模型。敲击退格键或回车键可以让视点远离或接近参考点。敲击 x/y/z 可以减小参考点对应的坐标值，敲击 X/Y/Z 可以增大参考点对应的坐标值。敲击空格键可以切换投影模式。  
  
上图左是平行投影模式的显示效果，上图右是透视投影模式的显示效果。代码如下：

# -\*- coding: utf-8 -\*-

# -------------------------------------------

# quidam\_02.py 旋转、缩放、改变视点和参考点

# -------------------------------------------

from OpenGL.GL import \*

from OpenGL.GLU import \*

from OpenGL.GLUT import \*

import numpy as np

IS\_PERSPECTIVE = True # 透视投影

VIEW = np.array([-0.8, 0.8, -0.8, 0.8, 1.0, 20.0]) # 视景体的left/right/bottom/top/near/far六个面

SCALE\_K = np.array([1.0, 1.0, 1.0]) # 模型缩放比例

EYE = np.array([0.0, 0.0, 2.0]) # 眼睛的位置（默认z轴的正方向）

LOOK\_AT = np.array([0.0, 0.0, 0.0]) # 瞄准方向的参考点（默认在坐标原点）

EYE\_UP = np.array([0.0, 1.0, 0.0]) # 定义对观察者而言的上方（默认y轴的正方向）

WIN\_W, WIN\_H = 640, 480 # 保存窗口宽度和高度的变量

LEFT\_IS\_DOWNED = False # 鼠标左键被按下

MOUSE\_X, MOUSE\_Y = 0, 0 # 考察鼠标位移量时保存的起始位置

def getposture():

global EYE, LOOK\_AT

dist = np.sqrt(np.power((EYE-LOOK\_AT), 2).sum())

if dist > 0:

phi = np.arcsin((EYE[1]-LOOK\_AT[1])/dist)

theta = np.arcsin((EYE[0]-LOOK\_AT[0])/(dist\*np.cos(phi)))

else:

phi = 0.0

theta = 0.0

return dist, phi, theta

DIST, PHI, THETA = getposture() # 眼睛与观察目标之间的距离、仰角、方位角

def init():

glClearColor(0.0, 0.0, 0.0, 1.0) # 设置画布背景色。注意：这里必须是4个参数

glEnable(GL\_DEPTH\_TEST) # 开启深度测试，实现遮挡关系

glDepthFunc(GL\_LEQUAL) # 设置深度测试函数（GL\_LEQUAL只是选项之一）

def draw():

global IS\_PERSPECTIVE, VIEW

global EYE, LOOK\_AT, EYE\_UP

global SCALE\_K

global WIN\_W, WIN\_H

# 清除屏幕及深度缓存

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

# 设置投影（透视投影）

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

if WIN\_W > WIN\_H:

if IS\_PERSPECTIVE:

glFrustum(VIEW[0]\*WIN\_W/WIN\_H, VIEW[1]\*WIN\_W/WIN\_H, VIEW[2], VIEW[3], VIEW[4], VIEW[5])

else:

glOrtho(VIEW[0]\*WIN\_W/WIN\_H, VIEW[1]\*WIN\_W/WIN\_H, VIEW[2], VIEW[3], VIEW[4], VIEW[5])

else:

if IS\_PERSPECTIVE:

glFrustum(VIEW[0], VIEW[1], VIEW[2]\*WIN\_H/WIN\_W, VIEW[3]\*WIN\_H/WIN\_W, VIEW[4], VIEW[5])

else:

glOrtho(VIEW[0], VIEW[1], VIEW[2]\*WIN\_H/WIN\_W, VIEW[3]\*WIN\_H/WIN\_W, VIEW[4], VIEW[5])

# 设置模型视图

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

# 几何变换

glScale(SCALE\_K[0], SCALE\_K[1], SCALE\_K[2])

# 设置视点

gluLookAt(

EYE[0], EYE[1], EYE[2],

LOOK\_AT[0], LOOK\_AT[1], LOOK\_AT[2],

EYE\_UP[0], EYE\_UP[1], EYE\_UP[2]

)

# 设置视口

glViewport(0, 0, WIN\_W, WIN\_H)

# ---------------------------------------------------------------

glBegin(GL\_LINES) # 开始绘制线段（世界坐标系）

# 以红色绘制x轴

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.8, 0.0, 0.0) # 设置x轴顶点（x轴负方向）

glVertex3f(0.8, 0.0, 0.0) # 设置x轴顶点（x轴正方向）

# 以绿色绘制y轴

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.0, -0.8, 0.0) # 设置y轴顶点（y轴负方向）

glVertex3f(0.0, 0.8, 0.0) # 设置y轴顶点（y轴正方向）

# 以蓝色绘制z轴

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, 0.0, -0.8) # 设置z轴顶点（z轴负方向）

glVertex3f(0.0, 0.0, 0.8) # 设置z轴顶点（z轴正方向）

glEnd() # 结束绘制线段

# ---------------------------------------------------------------

glBegin(GL\_TRIANGLES) # 开始绘制三角形（z轴负半区）

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.5, -0.366, -0.5) # 设置三角形顶点

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.5, -0.366, -0.5) # 设置三角形顶点

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, 0.5, -0.5) # 设置三角形顶点

glEnd() # 结束绘制三角形

# ---------------------------------------------------------------

glBegin(GL\_TRIANGLES) # 开始绘制三角形（z轴正半区）

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.5, 0.5, 0.5) # 设置三角形顶点

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.5, 0.5, 0.5) # 设置三角形顶点

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, -0.366, 0.5) # 设置三角形顶点

glEnd() # 结束绘制三角形

# ---------------------------------------------------------------

glutSwapBuffers() # 切换缓冲区，以显示绘制内容

def reshape(width, height):

global WIN\_W, WIN\_H

WIN\_W, WIN\_H = width, height

glutPostRedisplay()

def mouseclick(button, state, x, y):

global SCALE\_K

global LEFT\_IS\_DOWNED

global MOUSE\_X, MOUSE\_Y

MOUSE\_X, MOUSE\_Y = x, y

if button == GLUT\_LEFT\_BUTTON:

LEFT\_IS\_DOWNED = state==GLUT\_DOWN

elif button == 3:

SCALE\_K \*= 1.05

glutPostRedisplay()

elif button == 4:

SCALE\_K \*= 0.95

glutPostRedisplay()

def mousemotion(x, y):

global LEFT\_IS\_DOWNED

global EYE, EYE\_UP

global MOUSE\_X, MOUSE\_Y

global DIST, PHI, THETA

global WIN\_W, WIN\_H

if LEFT\_IS\_DOWNED:

dx = MOUSE\_X - x

dy = y - MOUSE\_Y

MOUSE\_X, MOUSE\_Y = x, y

PHI += 2\*np.pi\*dy/WIN\_H

PHI %= 2\*np.pi

THETA += 2\*np.pi\*dx/WIN\_W

THETA %= 2\*np.pi

r = DIST\*np.cos(PHI)

EYE[1] = DIST\*np.sin(PHI)

EYE[0] = r\*np.sin(THETA)

EYE[2] = r\*np.cos(THETA)

if 0.5\*np.pi < PHI < 1.5\*np.pi:

EYE\_UP[1] = -1.0

else:

EYE\_UP[1] = 1.0

glutPostRedisplay()

def keydown(key, x, y):

global DIST, PHI, THETA

global EYE, LOOK\_AT, EYE\_UP

global IS\_PERSPECTIVE, VIEW

if key in [b'x', b'X', b'y', b'Y', b'z', b'Z']:

if key == b'x': # 瞄准参考点 x 减小

LOOK\_AT[0] -= 0.01

elif key == b'X': # 瞄准参考 x 增大

LOOK\_AT[0] += 0.01

elif key == b'y': # 瞄准参考点 y 减小

LOOK\_AT[1] -= 0.01

elif key == b'Y': # 瞄准参考点 y 增大

LOOK\_AT[1] += 0.01

elif key == b'z': # 瞄准参考点 z 减小

LOOK\_AT[2] -= 0.01

elif key == b'Z': # 瞄准参考点 z 增大

LOOK\_AT[2] += 0.01

DIST, PHI, THETA = getposture()

glutPostRedisplay()

elif key == b'\r': # 回车键，视点前进

EYE = LOOK\_AT + (EYE - LOOK\_AT) \* 0.9

DIST, PHI, THETA = getposture()

glutPostRedisplay()

elif key == b'\x08': # 退格键，视点后退

EYE = LOOK\_AT + (EYE - LOOK\_AT) \* 1.1

DIST, PHI, THETA = getposture()

glutPostRedisplay()

elif key == b' ': # 空格键，切换投影模式

IS\_PERSPECTIVE = not IS\_PERSPECTIVE

glutPostRedisplay()

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

glutInit()

displayMode = GLUT\_DOUBLE | GLUT\_ALPHA | GLUT\_DEPTH

glutInitDisplayMode(displayMode)

glutInitWindowSize(WIN\_W, WIN\_H)

glutInitWindowPosition(300, 200)

glutCreateWindow('Quidam Of OpenGL')

init() # 初始化画布

glutDisplayFunc(draw) # 注册回调函数draw()

glutReshapeFunc(reshape) # 注册响应窗口改变的函数reshape()

glutMouseFunc(mouseclick) # 注册响应鼠标点击的函数mouseclick()

glutMotionFunc(mousemotion) # 注册响应鼠标拖拽的函数mousemotion()

glutKeyboardFunc(keydown) # 注册键盘输入的函数keydown()

glutMainLoop() # 进入glut主循环

**4.11 小结**

虽然还有很多领域需要我们继续探索，比如灯光、材质、雾化、拾取等，但那不是奇幻之旅的目标。奇幻之旅仅仅是帮助读者建立 OpenGL 的基本概念。至此，我们基本完成了任务。

**5 加速渲染**

实际应用 OpenGL 绘制三维图像时，往往需要处理数以万计的顶点，有时甚至是百万级、千万级。我们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。这个机制地实现，依赖于顶点缓冲区对象（Vertex Buffer Object），简称VBO。

尽管 VBO 是显卡的扩展，其实没有用到GPU运算，也就是说 VBO 不用写着色语言，直接用opengl函数就可以调用，主要目的是用于加快渲染的速。

VBO 将顶点信息放到 GPU 中，GPU 在渲染时去缓存中取数据，二者中间的桥梁是 GL-Context。GL-Context 整个程序一般只有一个，所以如果一个渲染流程里有两份不同的绘制代码，GL-context 就负责在他们之间进行切换。这也是为什么要在渲染过程中，在每份绘制代码之中会有 glBindbuffer、glEnableVertexAttribArray、glVertexAttribPointer。如果把这些都放到初始化时候完成，使用一种结构记录该次绘制所需要的所有 VBO 所需信息，把它保存到 VBO特定位置，绘制的时候直接在这个位置取信息绘制，会简化渲染流程、提升渲染速度。这就是 VAO 概念产生的初衷。

VAO 的全名是 Vertex Array Object，首先，它不是 Buffer-Object，所以不用作存储数据；其次，它针对“顶点”而言，也就是说它跟“顶点的绘制”息息相关。VAO 记录的是一次绘制中所需要的信息，这包括“数据在哪里 glBindBuffer”、“数据的格式是怎么样的 glVertexAttribPointer”、shader-attribute 的 location 的启用 glEnableVertexAttribArray。

根据我查到的资料，几乎所有的显卡都支持 VBO，但不是所有的显卡都支持 VAO，而 VAO 仅仅是优化了 VBO 的使用方法，对于加速并没有实质性的影响，因此本文只讨论 VBO 技术。

**5.1 创建顶点缓冲区对象（VBO）**

假定画一个六面体，顶点是这样的：

# 六面体数据

# ------------------------------------------------------

# v4----- v5

# /| /|

# v0------v1|

# | | | |

# | v7----|-v6

# |/ |/

# v3------v2

# 顶点集

vertices = np.array([

-0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, -0.5, 0.5, -0.5, -0.5, 0.5, # v0-v1-v2-v3

-0.5, 0.5, -0.5, 0.5, 0.5, -0.5, 0.5, -0.5, -0.5, -0.5, -0.5, -0.5 # v4-v5-v6-v7

], dtype=np.float32)

# 索引集

indices = np.array([

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

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

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

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

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

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

], dtype=np.int)

在GPU上创建VBO如下：

from OpenGL.arrays import vbo

vbo\_vertices = vbo.VBO(vertices)

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

创建 顶点 VBO 时，默认 target=GL\_ARRAY\_BUFFER， 而创建索引 VBO 时，target=GL\_ELEMENT\_ARRAY\_BUFFER，因为顶点的数据类型是 np.float32，索引的数据类型是np.int。

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

vertices = np.array([

0.3, 0.6, 0.9, -0.35, 0.35, 0.35, # c0-v0

0.6, 0.9, 0.3, 0.35, 0.35, 0.35, # c1-v1

0.9, 0.3, 0.6, 0.35, -0.35, 0.35, # c2-v2

0.3, 0.9, 0.6, -0.35, -0.35, 0.35, # c3-v3

0.6, 0.3, 0.9, -0.35, 0.35, -0.35, # c4-v4

0.9, 0.6, 0.3, 0.35, 0.35, -0.35, # c5-v5

0.3, 0.9, 0.9, 0.35, -0.35, -0.35, # c6-v6

0.9, 0.9, 0.3, -0.35, -0.35, -0.35 # c7-v7

], dtype=np.float32)

**5.2 分离顶点混合数组**

使用 glInterleavedArrays() 函数可以从顶点混合数组中分离顶点、颜色、法线和纹理。比如，对只包含顶点信息的顶点混合数组：

vbo\_indices.bind()

glInterleavedArrays(GL\_V3F, 0, None)

如果顶点混合数组包含了颜色和顶点信息：

vbo\_indices.bind()

glInterleavedArrays(GL\_C3F\_V3F, 0, None)

glInterleavedArrays() 函数第一个参数总共有14个选项，分别是：

* GL\_V2F
* GL\_V3F
* GL\_C4UB\_V2F
* GL\_C4UB\_V3F
* GL\_C3F\_V3F
* GL\_N3F\_V3F
* GL\_C4F\_N3F\_V3F
* GL\_T2F\_V3F
* GL\_T4F\_V4F
* GL\_T2F\_C4UB\_V3F
* GL\_T2F\_C3F\_V3F
* GL\_T2F\_N3F\_V3F
* GL\_T2F\_C4F\_N3F\_V3F
* GL\_T4F\_C4F\_N3F\_V4F

**5.3 使用顶点缓冲区对象（VBO）**

使用glDrawElements() 等函数绘制前，需要先绑定顶点数据集和索引数据集，然后使用glInterleavedArrays() 分理出顶点、颜色、法线等数据。

vbo\_indices.bind()

glInterleavedArrays(GL\_V3F, 0, None)

vbo\_indices.bind()

glDrawElements(GL\_QUADS, int(vbo\_indices .size/4), GL\_UNSIGNED\_INT, None)

vbo\_indices.unbind()

vbo\_indices.unbind()

**6 致谢**

写作过程中，我参考了很多资料，包括纸质书籍和网页，列写于此，一并致谢！

1. 《OpenGL编程精粹》杨柏林 陈根浪 徐静 编著
2. [Opengl开发库介绍](https://blog.csdn.net/yyyuhan/article/details/2045009)
3. [OpenGL的API函数使用手册](https://www.cnblogs.com/1024Planet/p/5764646.html)
4. [glut处理鼠标事件](https://blog.csdn.net/jacky_chenjp/article/details/69396540/)
5. [Learn OpenGL](https://learnopengl-cn.github.io/)